Organisation des Ordinateurs

Bernard Boigelot
Université de Liège

2020

Table des matières

Avant-propos

Les ordinateurs sont les machines les plus complexes et sans doute les plus fascinantes jamais construites. Ce cours fournit une première introduction à leurs principes de fonctionnement. Il vient en complément aux cours d'algorithmique et de programmation, dans lesquels on étudie comment développer des programmes permettant à l'ordinateur de résoudre des problèmes. Dans ce cours, nous nous intéresserons aux mécanismes qui permettent à un ordinateur d'exécuter de tels programmes.

Ces notes de cours sont structurées en plusieurs chapitres :

— Le chapitre 1 introduit la notion d'information et explique comment celle-ci peut être manipulée par les circuits d'un ordinateur.

— Le chapitre 2 étudie la représentation des données élémentaires amenées à être traitées par un ordinateur, c'est-à-dire les nombres entiers et réels, ainsi que les textes.

— Le chapitre 3 décrit l'architecture d'un processeur et les mécanismes permettant à celui-ci d'exécuter des programmes.

— Le chapitre 4 traite de la programmation en langage d'assemblage, et établit des liens entre les mécanismes étudiés dans ce cours et la programmation dans des langages de plus haut niveau.

Chacun de ces chapitres se termine par une série d'exercices, qui seront explorés plus en détail lors des séances de travaux pratiques.

Certaines notions qui ne sont abordées que de façon superficielle dans ce cours sont approfondies dans d'autres enseignements. Par exemple, la conception de circuits électroniques capables de traiter l'information est étudiée dans *Digital Electronics*. Les mécanismes permettant à un processeur d'exécuter un ou plusieurs programmes font l'objet des cours *Computation Structures* et *Operating Systems*. L'utilisation de l'outil informatique pour résoudre des problèmes numériques est abordée dans *Analyse Numérique*. Cette liste n'est pas exhaustive.

Chapitre 1

Le traitement de l'information

1.1 Qu'est-ce qu'un ordinateur ?

L'informatique est omniprésente dans la vie quotidienne. Lorsqu'on évoque le terme *ordinateur*, le grand public pense naturellement à ceux que l'on emploie pour effectuer des tâches bureautiques, naviguer sur Internet ou envoyer du courrier électronique. On sait également que des ordinateurs servent à gérer les stocks et la comptabilité des entreprises, à réaliser des simulations permettant par exemple de prévoir la météo, et à résoudre des problèmes scientifiques complexes. Enfin, personne n'ignore que les *smartphones*, devenus indispensables à beaucoup de personnes, sont essentiellement des ordinateurs de poche.

Ces exemples d'ordinateurs ne constituent cependant pas l'application la plus répandue de l'informatique. Des ordinateurs sont en effet aussi intégrés, sous la forme de *systèmes enfouis (embedded systems)*, ou *embarqués*, dans des dispositifs plus complexes où ils restent en général invisibles. Une voiture moderne, par exemple, comprend plusieurs dizaines de microprocesseurs chargés de gérer les paramètres de fonctionnement du moteur, de contrôler le système de freinage, de commander les accessoires comme la radio, les phares ou les lève-vitres, ... De nos jours, l'informatique est ainsi devenue un composant indispensable des appareils électroménagers, audio et vidéo, des caméras et des appareils photo, des jouets, des appareils de mesure, des système de contrôle industriels, ...

Une question intéressante est dès lors de caractériser précisément ce que l'on entend par *ordinateur*. Un point commun entre tous les types de systèmes informatiques que nous avons cités est que ceux-ci *traitent des données*. La nature de ces données diffère selon les applications ; il peut s'agir du contenu d'une page WWW pour un navigateur Internet, de la température de la mer à un ensemble de points de mesure pour un simulateur météo, de la position de la pédale d'accélération pour le contrôleur du moteur d'une voiture, ou du code du programme sélectionné pour un lave-linge. Notons que lorsqu'un ordinateur est employé pour effectuer un calcul ou résoudre un problème, on peut également considérer qu'il s'agit d'un traitement de données. Par

exemple, résoudre l'équation du second degré

$$ax^2 + bx + c = 0$$

correspond à traiter les données d'entrée (a, b, c) de façon à obtenir la ou les solutions de cette équation, ou bien une information signalant que ces solutions n'existent pas ou sont universelles.

Une autre caractéristique fondamentale des systèmes informatiques est qu'ils fonctionnent *en suivant un programme*. Par exemple, pour déterminer quelle quantité de carburant injecter à quel moment dans quel cylindre, le contrôleur du moteur d'une voiture suit un programme qui précise comment ces valeurs doivent être calculées, en fonction de divers paramètres comme le régime actuel du moteur, la position de la pédale d'accélération, la température et la pression atmosphériques ambiantes, … Ce programme constitue le *logiciel (software)* exécuté par cet ordinateur. Il précise en détail quelles opérations de traitement doivent être effectuées et dans quel ordre. Nous sommes maintenant à même de formuler notre définition d'un ordinateur.

> Un *ordinateur* est une machine capable de traiter des données, en suivant un programme préétabli.

Le fait qu'un ordinateur se contente d'exécuter mécaniquement les instructions figurant dans un programme, sans faire preuve de la moindre initiative, reste méconnu de beaucoup de personnes. Au cinéma, il n'est pas rare de voir des soi-disant ordinateurs résoudre par eux-mêmes des problèmes complexes, hors de portée de leurs opérateurs humains. (Un exemple célèbre est l'ordinateur *HAL 9000* du film *2001 : A Space Odyssey*.) Il n'est pas impossible qu'un ordinateur possède des capacités de raisonnement et d'intuition, mais pour cela, il faut lui fournir un programme qui détaille l'ensemble des opérations à effectuer pour mettre en œuvre de telles capacités. L'intelligence éventuelle d'un système informatique n'est donc jamais située dans ses composants matériels (*hardware*), mais bien dans son logiciel (*software*).

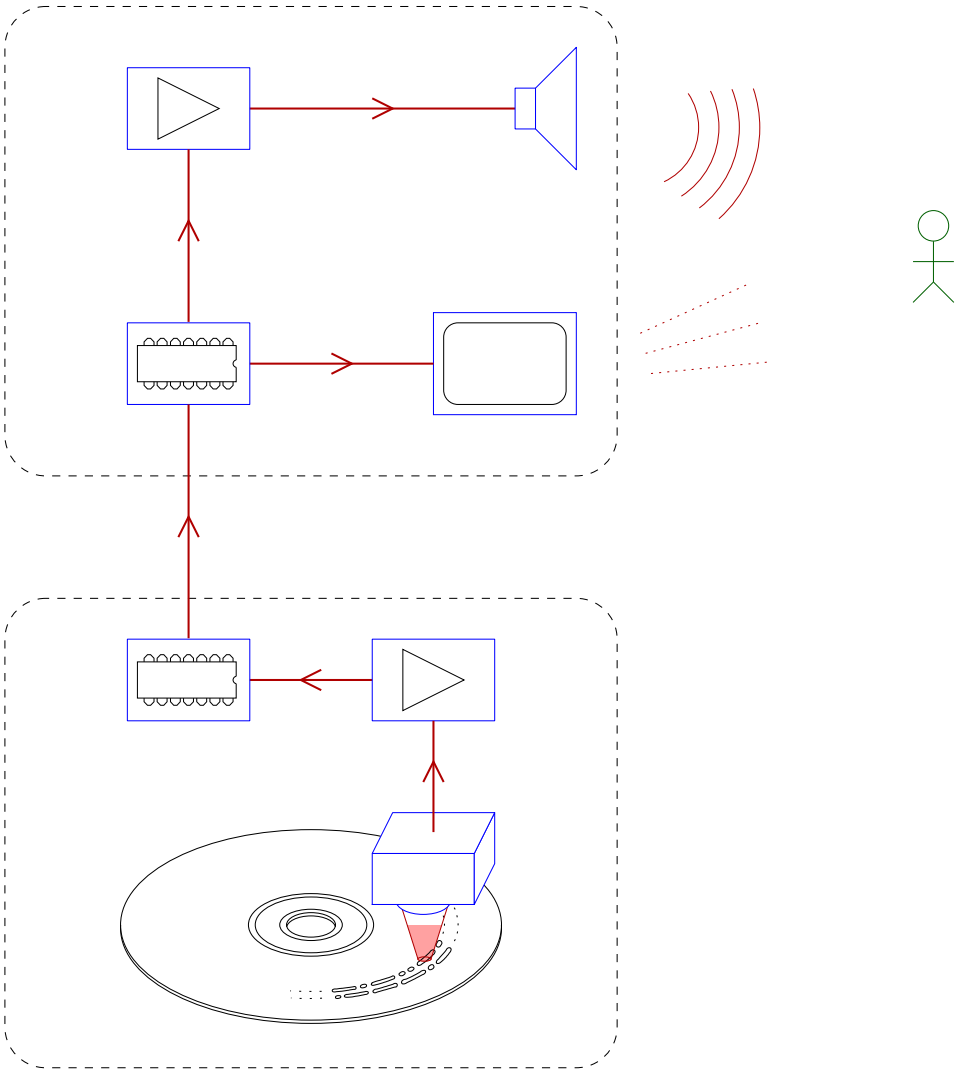
1.2 La notion d'information

Dans ce cours, nous allons étudier les mécanismes permettant à des machines de traiter des données. La première étape consiste à introduire la notion d'*information*, afin de caractériser la nature de ces données et la forme qu'elles prennent au sein des circuits de l'ordinateur.

En physique, on appelle *information* la connaissance qu'un observateur possède de l'état d'un système. Considérons par exemple l'expérience consistant à placer un dé à six faces dans une boîte opaque, et à agiter celle-ci. Si la boîte reste fermée, nous ne connaissons pas l'état du dé ; celui-ci peut prendre n'importe quelle valeur dans l'ensemble $\{1, 2, 3, 4, 5, 6\}$. Nous ne possédons donc aucune autre information sur cet état. Lorsque nous ouvrons la boîte, nous voyons immédiatement dans quelle configuration se trouve le dé, par exemple 3. Cela signifie que nous avons reçu de l'information supplémentaire. L'effet de celle-ci est d'améliorer la connaissance que nous avons de l'état du dé.

Dans cet exemple, c'est en regardant le dé que nous avons reçu de l'information. En d'autres termes, cette information nous a été transmise par des signaux optiques. De façon générale, les signaux véhiculant de l'information peuvent prendre des formes très variées : Toute grandeur physique prenant potentiellement plusieurs valeurs, capable d'être transmise d'un endroit à un autre, et pouvant être mesurée par un observateur, fournit de l'information.

Pour illustrer cette variété, considérons l'exemple d'un lecteur de DVD. Un DVD contient une grande quantité de données, correspondant principalement au contenu de toutes les images et des sons d'un film. Ces données sont représentées par une disposition particulière de petites [1] bosses à la surface d'une ou de plusieurs couches réfléchissantes situées à l'intérieur du disque.



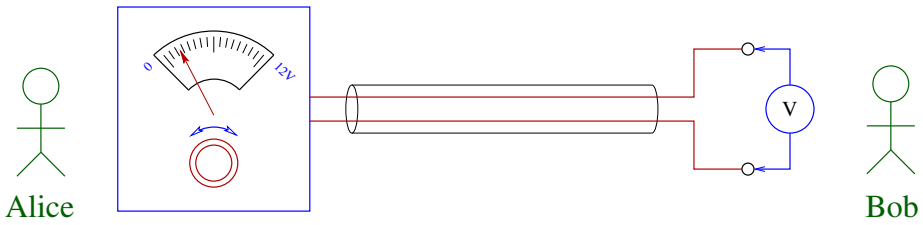
1. Leur taille est d'environ 400 nm, soit $1/2500$ mm, ce qui les rend invisibles à l'œil nu.

Avant d'insérer le DVD dans un lecteur, les données qu'il contient sont inconnues de celui-ci. Pour les obtenir, le lecteur projette un faisceau laser sur une couche réfléchissante du disque. Ce faisceau est réfléchi différemment selon que la surface visée présente une bosse ou non. En mesurant la lumière réfléchie à l'aide d'une photodiode, le lecteur reçoit donc de l'information indiquant la présence ou l'absence d'une bosse. Cette information est donc transmise du disque à la photodiode par un signal lumineux. La photodiode transforme alors ce signal en un courant électrique envoyé au circuit électronique du lecteur. Celui-ci reçoit donc, sous la forme d'un signal de courant, de l'information correspondant à une partie du contenu du disque. Ensuite, ce signal est transformé en une tension électrique par un amplificateur, avant d'être traité par un système informatique enfoui chargé de décoder les sons et les images. Ceux-ci sont alors transmis à la télévision connectée au lecteur, sous la forme de signaux électriques. A l'intérieur de la télévision, l'information est à nouveau traitée et communiquée d'un composant à un autre par des signaux électriques, avant de finalement atteindre la dalle d'affichage des images et le ou les haut-parleurs. La chaîne de transmission de l'information ne s'arrête pas là, puisque les images affichées sont perçues par les yeux du téléspectateur, dont la rétine les transforme en impulsions électriques envoyées au cerveau par le nerf optique. Les sons émis par la télévision sont captés de façon similaire. Comme on le voit, l'information est acheminée depuis le DVD vers les neurones du téléspectateur en empruntant une succession de canaux de communication optiques, électriques et sonores. A chaque étape de ce processus, de l'information est reçue, traitée et émise.

1.3 L'encodage de l'information

Intéressons nous plus en détail à la forme des signaux qui transmettent de l'information. Imaginons un dispositif simple, dans lequel un opérateur (*Alice*) manipule un potentiomètre permettant de placer une tension électrique sur un câble : Si le bouton est tourné en butée vers la droite, cette tension vaut 12 V ; complètement à gauche, elle est de 0 V. Entre ces deux extrêmes, la tension générée dépend linéairement de la position du bouton (en particulier, elle est de 6 V au milieu de l'intervalle).

A l'autre extrémité du câble se trouve un observateur (*Bob*), qui mesure la tension présente sur celui-ci à l'aide d'un voltmètre.



Il est clair que ce système permet à Alice de transmettre de l'information à Bob. En effet, initialement, Bob ne sait pas quelle tension va être produite par Alice. Son incertitude quant à

cette tension est donc complète (du moins, dans l'intervalle [0, 12] V). En revanche, lorsque Bob effectue une mesure, le résultat de celle-ci l'informe de la tension produite par Alice, ce qui réduit donc l'incertitude de Bob quant à cette tension. On peut donc dire que, suite à la mesure qu'il a effectué, Bob a reçu de l'information.

Il serait cependant incorrect d'affirmer que Bob connaît maintenant exactement la tension émise par Alice ! Il est en effet physiquement impossible que le dispositif que nous avons décrit permette à Alice de générer une tension, au câble de propager celle-ci, et à Bob de la mesurer, sans que ces opérations ne souffrent d'imprécisions. En particulier, Alice n'est capable de placer le bouton à la position qu'elle souhaite qu'avec une précision limitée. De plus, la linéarité du potentiomètre entre ses deux butées n'est certainement pas parfaite. La propagation du signal de tension le long du câble est quant à elle affectée par des signaux parasites, produits par exemple par d'autres appareils électroniques situés à proximité. Enfin, le voltmètre utilisé pour la mesure ne possède pas non plus une précision infinie.

Cette situation est inévitable lorsqu'un signal prend ses valeurs dans un domaine dense, comme c'est le cas pour un intervalle de $\mathbb{R}$. On parle dans ce cas d'un signal *continu*. Un signal continu est toujours affecté par des imprécisions qui le perturbent lorsqu'il est émis, transmis et reçu. En soignant la construction d'un dispositif, il y a souvent moyen de réduire l'ampleur de ces imprécisions, mais il n'est jamais possible de les éliminer totalement.

L'imprécision qui affecte un signal continu limite l'information que celui-ci véhicule. Par exemple, supposons que le système de communication que nous avons considéré soit construit de façon à ce que la tension mesurée par Bob ne diffère jamais de plus de 0,1 V de celle émise par Alice. Lorsque Bob mesure une tension de 3,14 V, il sait donc que celle produite par Alice appartient à l'intervalle [3,04, 3,24] V. Si, en revanche, la différence entre les tensions générée et mesurée peut monter jusqu'à 0,5 V, alors une mesure de 3,14 V ne permet de situer la tension émise que dans l'intervalle [2,64, 3,64]. L'incertitude de Bob quant à la tension produite par Alice est donc plus élevée, ce qui indique que Bob reçoit moins d'information. Nous aborderons à la section 1.4 la question de quantifier l'information transmise par un signal.

Pour concevoir les circuits d'un ordinateur, utiliser des signaux continus est problématique, en raison de cette présence inévitable d'imprécisions. On souhaite en effet que les données traitées par un ordinateur ne soient pas entachées d'erreurs. En d'autres termes, lorsqu'un composant A transmet une valeur à un composant B au sein d'un circuit, il est indispensable que la valeur reçue par B soit toujours égale à celle émise par A.

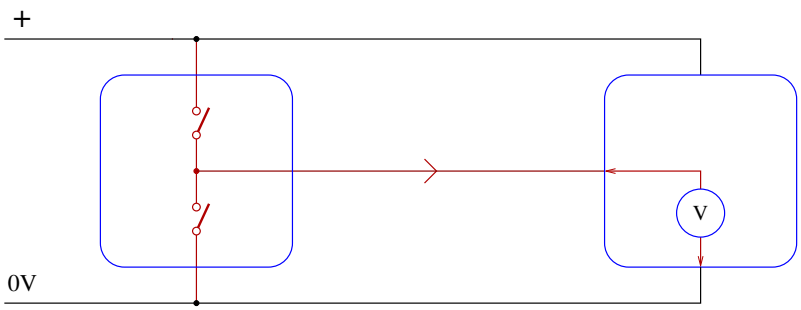
Les signaux continus ne permettant pas de s'affranchir des imprécisions, il est nécessaire de se tourner vers un autre type de signaux, les signaux *discrets*. Un signal discret est un signal qui ne possède qu'un nombre fini de valeurs nominales. Par exemple, nous pourrions modifier le dispositif de communication entre Alice et Bob en remplaçant le potentiomètre par un bouton cranté à 13 positions, permettant de sélectionner la tension à produire dans l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ V. Si ce système est construit de façon à ce que les tensions émise et mesurée ne s'écartent pas l'une de l'autre de plus de 0,25 V, alors Bob est à même d'affirmer, lorsqu'il observe une tension de 3,14 V, que celle choisie par Alice est exactement égale à 3 V.

Dans cet exemple, les 13 valeurs nominales du signal peuvent être fiablement décodées si l'amplitude de l'imprécision reste inférieure à 0,5V, c'est-à-dire la moitié de la distance entre deux valeurs voisines. Si l'imprécision est supérieure à ce seuil, alors le nombre de valeurs nominales du signal devra être réduit si l'on souhaite que celui-ci puisse toujours être correctement décodé. Cela diminuera l'information transmise par ce signal.

Les signaux discrets sont employés dans toutes les applications où des données doivent être transmises fiablement. Par exemple, la présence ou l'absence d'une bosse sur une couche réfléchissante d'un DVD constitue un signal discret à deux valeurs nominales. La télécommande d'une télévision fonctionne en émettant de la lumière infrarouge sujette à de nombreuses perturbations. Pour que le code émis par la télécommande puisse être correctement identifié par la télévision, ce signal lumineux prend la forme d'un train d'impulsions dont la présence ou l'absence représente un signal à deux valeurs. Enfin, les connexions à Internet de type *DSL* fonctionnent en échangeant des signaux électriques dans différentes plages de fréquence via des lignes téléphoniques. Dans chacune de ces plages, des signaux discrets sont utilisés, en choisissant dynamiquement un nombre de valeurs nominales compatible avec l'amplitude observée des imprécisions. Cette amplitude croît généralement avec la longueur de la ligne, ce qui limite l'information pouvant être transmise par ce type de connexion.

Dans les ordinateurs modernes, l'information est le plus souvent représentée à l'aide de signaux discrets *binaires*, c'est-à-dire possédant deux valeurs nominales. Cela n'a pas toujours été le cas ; parmi les premiers ordinateurs, certains comme l'*ENIAC* (1946) employaient des signaux à 10 valeurs nominales. Le choix d'utiliser des signaux binaires présente trois avantages :

— Un tel signal est facile à générer et à décoder. Dans la très grande majorité des cas, il correspond à la tension électrique présente sur un conducteur reliant deux composants d'un circuit. Les deux valeurs nominales de ce signal sont alors définies comme étant la tension nulle (par rapport à une référence donnée appelée la *masse* du circuit), et la *tension d'alimentation* fixe et connue du circuit. Pour émettre un signal sur un conducteur, il suffit alors d'établir une connexion électrique entre celui-ci et la masse ou la ligne d'alimentation, en fonction de la valeur souhaitée. La valeur d'un tel signal peut être décodée en mesurant la tension présente sur le conducteur, et en déterminant si elle est inférieure ou supérieure à un seuil donné.



— La distance séparant les valeurs nominales est la plus grande possible, pour une tension d'alimentation donnée, ce qui rend les circuits robustes par rapport aux perturbations affectant les signaux.

— La manipulation et le traitement de valeurs binaires est simple, grâce à l'*algèbre booléenne*. L'étude de celle-ci sort du cadre de ce cours. Elle est abordée notamment dans *Digital Electronics*, où cette algèbre est appliquée à l'analyse et à la conception de circuits électroniques capables de traiter l'information. Dans ce cours, nous étudierons au chapitre 2 les procédés permettant de représenter des nombres et des textes à l'aide de valeurs binaires.

1.4 La quantité d'information

Nous avons vu dans la section 1.3 que les signaux peuvent transporter plus ou moins d'information. Nous allons à présent chercher à définir précisément la quantité d'information associée à un signal. Cette définition doit obéir à deux principes :

— Plus une valeur est inattendue, plus elle apporte d'information. En d'autres termes, la quantité d'information associée à un signal est d'autant plus élevée que la probabilité de le recevoir est faible. Il s'agit d'une conséquence de la discussion menée dans la section 1.2 : Si la probabilité de recevoir une valeur donnée d'un signal est élevée, alors cette valeur ne permet de lever que de façon limitée l'incertitude que l'on a de l'état de l'émetteur.

  Exemple : Considérons à nouveau l'expérience discutée dans la section 1.2, où nous avions enfermé un dé à six faces dans une boîte. Lorsque nous ouvrons celle-ci et observons la valeur du dé, nous recevons une certaine quantité $q_{dé}$ d'information. Si nous réalisons la même expérience avec une pièce de monnaie, dont les deux valeurs nominales sont *pile* et *face*, la quantité d'information reçue vaut alors $q_{pièce}$.

  Si le dé n'est pas truqué, alors la probabilité d'observer une valeur particulière, quelle qu'elle soit, vaut $p_{dé} = \frac{1}{6}$. Pour la pièce, on a $p_{pièce} = \frac{1}{2}$.

  On a $p_{dé} < p_{pièce}$, ce qui entraîne $q_{dé} > q_{pièce}$. Un lancer de dé fournit donc plus d'information qu'un lancer de pièce. □

— Si une donnée est transmise par un signal, ou par une combinaison équivalente de plusieurs signaux, la quantité d'information totale reçue reste la même. En d'autres termes, la quantité d'information n'est pas affectée par la forme particulière que l'on donne aux signaux.

  Exemple : Effectuons trois lancers successifs de la pièce de monnaie. Si nous souhaitons en communiquer les résultats à quelqu'un, plusieurs solutions sont possibles. Une première option serait d'envoyer trois signaux binaires successifs, correspondant à l'état des

trois pièces [2]. Dans ce cas, la quantité totale d'information reçue vaut $3\,q_{pièce}$.

Une autre possibilité serait de générer un signal à huit valeurs, correspondant aux huit combinaisons possibles de valeur des trois pièces : (*pile*, *pile*, *pile*), (*pile*, *pile*, *face*), (*pile*, *face*, *pile*), (*pile*, *face*, *face*), (*face*, *pile*, *pile*), (*face*, *pile*, *face*), (*face*, *face*, *pile*) et (*face*, *face*, *face*). Chacune de ces combinaisons possède la même probabilité se se produire. On a donc dans ce cas $p_{troispièces} = \frac{1}{8}$, et la quantité d'information $q_{troispièces}$ correspondante doit être égale au résultat $3\,q_{pièce}$ obtenu dans le premier cas. □

Une définition satisfaisant ces deux principes est la suivante : La quantité d'information fournie par un signal présentant une probabilité p d'être reçu est égale à

$$\log_2 \frac{1}{p}.$$

En effet, cette expression est bien décroissante en p. De plus, une combinaison de signaux indépendants de probabilités respectives $p_1, p_2, \ldots, p_k$ présente la probabilité $p_1 p_2 \ldots p_k$ d'être reçue. On a bien dans ce cas

$$\log_2 \frac{1}{p_1 p_2 \ldots p_k} = \log_2 \frac{1}{p_1} + \log_2 \frac{1}{p_2} + \cdots + \log_2 \frac{1}{p_k}.$$

La quantité d'information fournie par cette combinaison de signaux est donc bien identique à celle d'un seul signal transmettant la même donnée.

L'unité dans laquelle la quantité d'information s'exprime est le *bit (b)*, une abréviation de *binary digit (chiffre binaire)*. En effet, dans le cas d'un signal binaire dont les deux valeurs sont considérées équiprobables (donc de probabilité $p = \frac{1}{2}$), on a

$$\log_2 \frac{1}{p} = \log_2 2 = 1.$$

De façon plus générale, si un signal possède n valeurs équiprobables, alors la quantité d'information qu'il contient est égale à

$$\log_2 \frac{1}{\frac{1}{n}} = \log_2 n \text{ bits}.$$

D'autres unités ont également employées. Un *octet (byte, B)* représente 8 bits d'information. Un *nibble* est un demi-octet, soit 4 bits. Les préfixes K (*kilo*), M (*mega*), G (*giga*), T (*tera*), P (*peta*), … ont une signification qui diffère suivant le contexte. En informatique, pour la plupart des applications, ils correspondent à des facteurs égaux à une puissance de deux, sur base de l'approximation $2^{10} = 1024 \approx 1000$: On a ainsi K = 2^{10}, M = 2^{20}, G = 2^{30}, T = 2^{40}, P =

2. Dans cette expérience, on ne se contente pas de compter le nombre de pièces pile et le nombre de pièces face, mais on souhaite connaître le résultat individuel de chaque lancer.

2^{50}, … Les fabricants de supports de données interprètent cependant ces préfixes de façon plus traditionnelle, car cela leur est plus favorable : K= 10^3, M = 10^6, G = 10^9, T = 10^{12}, P = 10^{15}, … La capacité d'un disque dur annoncé comme capable de contenir 4 TB ne s'élève ainsi en réalité qu'à

$$\frac{4 \times 10^{12}}{2^{40}} \approx 3{,}64 \text{ TB},$$

soit environ 9% de moins.

1.5 Exercices résolus

1. Quelle est la quantité d'information fournie par les expériences de lancer de dé et de pièces discutées à la section 1.4 ?

   Solution : Dans le cas d'un dé non truqué, il y a six valeurs équiprobables, donc la quantité d'information apportée par un lancer vaut

   $$\log_2 6 \approx 2{,}58 \text{ bits}.$$

   Pour une pièce non biaisée, chaque lancer fournit 1 bit d'information, donc les résultats de trois lancers successifs totalisent 3 bits. Il s'agit bien de la même quantité d'information que celle d'un signal possédant 8 valeurs équiprobables, puisque $\log_2 8 = 3$.

2. On construit un système de télex simple en disposant un émetteur capable de générer une tension comprise entre 0 V et 1 V à une extrémité d'un câble, et un récepteur mesurant la tension présente à l'autre extrémité. Ce système permet de transmettre des lettres grâce au codage suivant : A = 0 V, B = 0, 04 V, C = 0, 08 V, …, Z = 1 V.

   Quelle est la quantité d'information transmise par un signal, dans chacun des cas suivants ?

   (a) Le système est employé pour envoyer un texte.

   (b) Le système est utilisé pour transmettre un mot de passe dont les lettres sont aléatoires.

   (c) Le niveau de bruit affectant le canal de communication ne permet de distinguer fiablement que les tensions émises qui sont supérieures et inférieures à 0,5 V.

   Solution :

   (a) Pour pouvoir répondre à cette question, il faut connaître la probabilité de recevoir une valeur, qui dépend du type de texte transmis.

   Par exemple, un texte aléatoire rédigé en français contient environ 17% de lettres E, mais seulement 0,12% de lettres Z. La quantité d'information apportée par une lettre E vaut donc

   $$\log_2 \frac{1}{0{,}17} \approx 2{,}56 \text{ bits},$$

et celle d'un Z est égale à

$$\log_2 \frac{1}{0{,}0012} \approx 9{,}70 \text{ bits},$$

à condition que chaque signal puisse être décodé fiablement.

Une lettre Z fournit donc presque quatre fois plus d'information qu'une lettre E. Cette observation peut être mise à profit afin d'encoder efficacement les textes, en choisissant la taille des symboles en fonction de la quantité d'information qui y est associée. C'est pour cette raison que dans le code morse, la lettre Z est représentée par un symbole (▬ ▬ . .) plus long que celui de la lettre E (.). Le même procédé est employé par les outils de compression de fichiers.

(b) Si le texte transmis est composé de lettres tirées aléatoirement avec une distribution de probabilité uniforme sur l'intervalle $[\mathrm{A}, \mathrm{Z}]$, et si chaque signal peut être décodé fiablement, alors la quantité d'information de chaque lettre est identique et vaut

$$\log_2 26 \approx 4{,}70 \text{ bits}.$$

(c) Si seules deux valeurs du signal peuvent être distinguées, et si celles-ci sont équiprobables, alors la quantité d'information fournie par chaque valeur vaut

$$\log_2 2 = 1 \text{ bit}.$$

Cela montre bien que l'augmentation du niveau de bruit sur un canal de communication a pour effet de réduire la quantité d'information que ce canal est capable de transmettre.

1.6 Exercices supplémentaires

1. Le clavier d'un téléphone emploie des signaux discrets pour transmettre les chiffres de 0 à 9.
   (a) Quelle quantité d'information est-elle véhiculée par un signal représentant un chiffre, si chacun d'entre eux possède la même probabilité d'être transmis ?
   (b) Les chiffres qui composent un numéro de téléphone sont transmis successivement. Si l'on sait qu'un numéro de téléphone ne peut jamais commencer par le chiffre 0 et qu'il comporte exactement 4 chiffres, quelle est la quantité totale d'information fournie par un numéro ?
   (c) Si un numéro de téléphone était au contraire transmis par un seul signal, représentant les numéros complets par des valeurs équiprobables, quelle serait la quantité d'information fournie par un numéro ?

2. Un DVD simple couche représente l'information par des bosses imprimées sur une surface réfléchissante, prenant la forme d'une couronne de 24 mm de rayon intérieur et de 58 mm de rayon extérieur. Les bosses sont disposées le long d'un sillon enroulé en spirale, avec une séparation latérale de 740 nm entre les spires. Les données stockées sur le disque sont encodées grâce au procédé suivant :

   — Chaque octet de données est transformé en une séquence de 16 valeurs binaires, en suivant une table de conversion figurant dans la spécification du format DVD.

   — Chaque valeur binaire ainsi obtenue se traduit par la présence ou à l'absence d'une bosse à un endroit donné du sillon, sur une longueur de 133,3 nm.

   Calculer la capacité de stockage de données, en octets, d'un DVD simple couche. Si le résultat obtenu ne correspond pas à la capacité annoncée par les fabricants de DVD (4,7 GB), expliquer l'origine de la différence observée.

3. Une ligne de télécommunications est composée de huit canaux parallèles, dans lesquels on peut émettre des signaux discrets d'amplitude comprise entre 0 V et 2 V, à la fréquence de 10^7 signaux par canal par seconde. Des mesures ont été effectuées afin de connaître l'amplitude des parasites affectant chaque canal, c'est-à-dire la différence d'amplitude maximale entre les signaux émis et reçus sur ce canal. Ces mesures ont retourné les valeurs suivantes (en millivolts) :

   [3,01, 5,56, 4,81, 4,75, 5,16, 3,84, 3,91, 3,82]

   Quelle est la capacité de transmission maximale, en bits par seconde, de cette ligne ?

4. *(Examen de première session, 2018)* En français, la probabilité qu'une lettre prise au hasard dans un texte soit un "O", un "C", un "T" ou un "E" est respectivement égale, approximativement, à 5,02%, 3,18%, 5,92% et 12,20%. Par souci de simplicité, on considère que ces probabilités ne dépendent ni de la place des lettres dans les mots, ni de la nature des lettres voisines.

   Sous ces hypothèses, on demande de calculer la quantité d'information contenue dans le mot "OCTET".

5. *(Examen de seconde session, 2018)* Une bande magnétique mesurant 154 m est composée de 1200 pistes parallèles. Sur chacune de ces pistes, l'information est représentée par l'orientation de domaines magnétiques disposés séquentiellement. Chaque domaine mesure 6,8 μm de longueur et possède quatre orientations possibles. On demande de calculer la quantité d'information totale mémorisée par une telle bande magnétique.

6. *(Examen de première session, 2019)* En Belgique, un numéro de téléphone mobile est formé

   — d'un préfixe de la forme $04c_1c_2$, où c_1 est un chiffre de 5 à 9 et c_2 un chiffre quelconque, et
   — d'un suffixe composé de 6 chiffres, dont le premier est non nul.

   (a) En supposant que tous les numéros de téléphone mobile valables sont équiprobables, calculez la quantité d'information contenue dans un numéro.

   (b) Un opérateur téléphonique gère 2 millions de numéros de téléphone mobile belges, regroupés dans une base de données qui associe à chaque numéro le code postal de son titulaire. En sachant qu'il y a 2825 codes postaux distincts, supposés équiprobables et indépendants des numéros de téléphone, calculez la quantité de mémoire nécessaire au stockage de cette base de données.

7. *(Examen de première session, 2019)* Quelle quantité d'information un disque dur vendu comme possédant une capacité de 1 TB permet-il de mémoriser ?

8. *(Examen de seconde session, 2019)* Une machine est constituée de deux récipients opaques R_1 et R_2 entre lesquels sont réparties mille billes identiques. Chaque bille a une probabilité égale de se trouver dans R_1 ou dans R_2, indépendamment des autres billes. La machine comprend également une balance de précision située sous R_1, affichant en permanence le nombre de billes que ce récipient contient.

   Si la balance affiche 0, quelle quantité d'information fournit-elle sur l'état de la machine ?

9. *(Examen de seconde session, 2019)* Une station météorologique est équipée de deux capteurs : un thermomètre affichant la température par pas de 0,5°, et un pluviomètre renseignant la quantité de précipitations en millimètres avec un chiffre après la virgule. On suppose que la température est indépendante des précipitations, que la température affichée suit une distribution uniforme entre $-20°$ et 40°, et que la valeur renseignée par le pluviomètre suit une distribution uniforme entre 0 et 100 mm.

   Quelle quantité de mémoire, exprimée en octets, est-elle nécessaire pour enregistrer le bulletin météorologique de cette station ?

Chapitre 2

La représentation des données

2.1 Introduction

Au chapitre précédent, nous avons vu que les ordinateurs représentent l'information qu'ils manipulent à l'aide de signaux discrets binaires. Les deux valeurs nominales de ces signaux sont notées 0 et 1. Il ne s'agit que d'une convention arbitraire. Pour certaines applications, cette notation peut être différente ; par exemple, lorsqu'il s'agit d'établir un lien avec la logique, les deux valeurs sont appelées *Faux (False)* et *Vrai (True)*. En électronique numérique, lorsqu'on souhaite faire directement référence au niveau de tension d'un signal, on utilise parfois les termes *Bas (Low, L)* et *Haut (High, H)*. Ce choix n'a pas d'importance, tant qu'il est appliqué de façon cohérente à tous les composants qui s'échangent mutuellement de l'information.

Dans ce chapitre, nous allons étudier les procédés qui permettent de représenter à l'aide des valeurs binaires 0 et 1 des données plus complexes, comme des nombres et des textes.

2.2 Les nombres entiers non signés

Le premier type de données que nous allons considérer est celui des entiers non négatifs, qui sont indispensables à toutes les applications de l'informatique. La plupart des langages de programmation permettent de définir des *variables* capables de retenir une valeur entière. Ces variables sont généralement de taille fixée, c'est-à-dire qu'elles correspondent à un certain nombre n de bits dans la mémoire de l'ordinateur. Le problème qui nous intéresse est celui d'encoder les entiers naturels sous la forme de n valeurs binaires. Le procédé choisi pour cet encodage doit permettre d'effectuer des opérations de traitement d'entiers (comme par exemple des additions ou des multiplications) directement sur base de ces valeurs binaires.

2.2.1 La notation positionnelle

Pour représenter un entier non négatif à l'aide des seuls symboles 0 et 1, on peut utiliser un procédé analogue à celui qui nous employons dans la vie de tous les jours pour exprimer les nombres. Lorsque nous écrivons un nombre entier, par exemple 123, nous appliquons la *notation positionnelle en base 10*, basée sur les principes suivants :

— Chaque symbole de l'écriture du nombre est un *chiffre*, appartenant à l'ensemble {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

— Le *poids* de chaque chiffre dans la valeur du nombre est une puissance de 10, déterminée sur base de la *position* de ce chiffre : Le chiffre le plus à droite possède la position 0 et le poids 10^0, le chiffre immédiatement à sa gauche la position 1 et le poids 10^1, et ainsi de suite jusqu'au chiffre le plus à gauche qui possède la position $n-1$ et le poids 10^{n-1}, où n est le nombre de chiffres écrits.

Pour le nombre 123, on obtient bien

poids :	10^2	10^1	10^0
position :	2	1	0
	1	2	3

$$\begin{aligned} & 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \\ = \; & 1 \times 100 + 2 \times 10 + 3 \times 1 \\ = \; & 123. \end{aligned}$$

Cette notation se généralise à n'importe quelle *base* entière $r > 1$:

— Les chiffres utilisables appartiennent alors à l'ensemble $\{0, 1, \ldots, r-1\}$.

— Le poids du chiffre situé à une position k de l'écriture d'un nombre est égal à r^k.

Dans le cas où la base r est égale à 2, on parle de représentation *binaire* des nombres. Dans cette représentation, le nombre 123 s'écrit 1111011. En effet, on a

poids :	2^6	2^5	2^4	2^3	2^2	2^1	2^0
position :	6	5	4	3	2	1	0
	1	1	1	1	0	1	1

$$\begin{aligned} & 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ = \; & 64 + 32 + 16 + 8 + 2 + 1 \\ = \; & 123. \end{aligned}$$

De façon générale, le nombre encodé par la suite de bits $b_{n-1}b_{n-2}\dots b_0$, avec $n > 0$, est égal à

$$\sum_{i=0}^{n-1} 2^i\, b_i. \tag{2.1}$$

Cette expression permet de calculer la valeur du nombre non signé représenté par une suite de bits donnée. Pour effectuer l'opération réciproque, c'est-à-dire déterminer la suite de bits qui représente un nombre donné, on peut procéder de la façon suivante. On remarque tout d'abord que le bit situé à la position 0, appelé *bit de poids faible* [1] est particulier : C'est le seul à posséder un poids impair. Par conséquent, la somme

$$\sum_{i=0}^{n-1} 2^i\, b_i.$$

possède toujours la même parité que b_0. En effet, la somme de n'importe quelle combinaison de nombre pairs est nécessairement paire. Pour calculer le bit de poids faible b_0 de la représentation $b_{n-1}b_{n-2}\dots b_0$ d'un nombre v, il suffit donc de déterminer si v est pair (ce qui entraîne $b_0 = 0$) ou bien impair ($b_0 = 1$).

Pour calculer ensuite la valeur de b_1, le plus simple consiste à examiner le la même façon le nombre représenté par la suite de bits $b_{n-1}b_{n-2}\dots b_1$, obtenue en retirant b_0. Notons v' ce nombre. On a

$$v' = \sum_{i=1}^{n-1} 2^{i-1}\, b_i.$$

étant donné que le poids de chaque bit a été divisé par 2, et que b_0 n'est plus présent. On a donc

$$v = 2v' + b_0,$$

ce qui signifie que v' correspond à la division entière de v par 2. Cette propriété peut s'écrire

$$v' = \left\lfloor \frac{v}{2} \right\rfloor,$$

où $\lfloor\cdot\rfloor$ est l'opérateur d'*arrondi vers le bas*, qui retourne le plus grand nombre entier inférieur ou égal à son opérande. L'intérêt de cette manipulation est que b_1 est maintenant le bit de poids faible de la représentation $b_{n-1}b_{n-2}\dots b_1$ de v'. Sa valeur correspond donc à la parité de v'.

Ensuite, la même procédure peut être continuée pour obtenir successivement b_2, b_3, … et b_{n-1}. En résumé, l'algorithme permettant de calculer la représentation binaire d'un nombre entier $v \geq 0$ est le suivant :

1. Si v est pair, afficher 0. Sinon, afficher 1.

1. De même, celui situé à la position $n - 1$ est appelé *bit de poids fort*.

2. Remplacer v par $\left\lfloor \frac{v}{2} \right\rfloor$.

3. Si $v \neq 0$, recommencer à l'étape 1.

Cette procédure construit la représentation de v de la droite vers la gauche, c'est-à-dire du bit de poids faible vers le bit de poids fort. Le résultat correspond à la plus courte représentation binaire de v ; cette représentation peut toujours être étendue en la préfixant d'un nombre quelconque de *zéros de tête*.

Exemple : Pour le nombre 123, l'algorithme s'exécute de la façon suivante :

v	*parité*		*bit généré*
123	impair	$\longrightarrow$	1
61	impair	$\longrightarrow$	1
30	pair	$\longrightarrow$	0
15	impair	$\longrightarrow$	1
7	impair	$\longrightarrow$	1
3	impair	$\longrightarrow$	1
1	impair	$\longrightarrow$	1
0			

La représentation binaire non signée la plus courte de ce nombre est donc égale à 1111011. □

Les nombres qui possèdent une représentation binaire non signée sur n bits, avec $n \geq 1$, sont ceux qui appartiennent à l'intervalle

$$[0, 2^n - 1].$$

En effet, ces nombres sont ceux pour lesquels l'algorithme de construction de la représentation la plus courte se termine après avoir généré au plus n bits. La représentation sur n bits de chaque nombre appartenant à cet intervalle est unique.

Dans les applications informatiques, on est souvent confronté à des représentations binaires de nombres dont la taille n est égale à 8, 16 et 32. Ces représentations permettent donc de traiter des nombres inférieurs ou égaux respectivement à 255, 65535 et 4294967295.

2.2.2 La représentation hexadécimale

La notation positionnelle n'est pas limitée aux bases $r = 2$ (binaire) et $r = 10$ (décimale). Le choix de la base $r = 16$ (*hexadécimale*) conduit à une représentation des nombres très utilisée en informatique.

La base hexadécimale nécessite 16 chiffres. Par convention, ceux-ci sont notés [2], par ordre croissant de valeur : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F.

2. Les six lettres A, B, C, D, E et F peuvent indifféremment s'écrire en minuscules.

Exemple : En représentation hexadécimale, le nombre 1234 se note 4D2. En effet, on a

poids :	16^2	16^1	16^0
position :	2	1	0
	4	D	2

$$\begin{aligned} & 4 \times 16^2 + 13 \times 16^1 + 2 \times 16^0 \\ &= 4 \times 256 + 13 \times 16 + 2 \times 1 \\ &= 1234. \end{aligned}$$

□

L'intérêt de la représentation hexadécimale est qu'elle peut très facilement être convertie en une représentation binaire, et réciproquement. Elle est cependant plus lisible que le binaire ; pour cette raison, elle est souvent employée lorsqu'un programme doit faire explicitement référence à la représentation interne des données.

Cette conversion exploite l'égalité $16 = 2^4$, qui implique que chaque chiffre hexadécimal correspond à une séquence de quatre bits selon la table de correspondance suivante :

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Par exemple, pour convertir 4D2 de l'hexadécimal vers le binaire, il suffit d'assembler les séquences $\boxed{0100}$, $\boxed{1101}$ et $\boxed{0010}$ correspondant respectivement aux chiffres 4, D et 2. On obtient alors $\boxed{010011010010}$. (Il ne s'agit pas de la représentation la plus courte ; pour obtenir celle-ci, il ne faut pas oublier d'ôter le zéro de tête.) Cette conversion est correcte car on a bien

$$\begin{aligned} & 4 \times 16^2 + 13 \times 16^1 + 2 \times 16^0 \\ &= 2^2 \times 2^8 + (2^3 + 2^2 + 2^0) \times 2^4 + 2^1 \times 2^0 \\ &= 2^{10} + 2^7 + 2^6 + 2^4 + 2^1. \end{aligned}$$

En pratique, lorsqu'on écrit un nombre en notation positionnelle, il est important de préciser la base utilisée si celle-ci n'est pas évidente d'après le contexte. Il existe plusieurs moyens de le faire :

— En accompagnant chaque nombre d'un indice indiquant la base dans laquelle il est représenté. Par exemple :

$$1234_{10} = 4D2_{16} = 10011010010_2.$$

— En ajoutant à l'écriture d'un nombre le suffixe *d* pour la base décimale, *h* pour hexadécimale et *b* pour binaire :

$$1234d = 4D2h = 10011010010b.$$

— En préfixant les nombres hexadécimaux de 0x et les nombres binaires de 0b :

$$1234 = 0x4D2 = 0b10011010010.$$

2.2.3 Les opérations sur les entiers non signés

Au sein des circuits d'un ordinateur, les nombres sont représentés en notation binaire, et les opérations arithmétiques impliquant des nombres sont effectués sur base de cette représentation. Les opérations de conversion vers et depuis la base 10 évoquées à la section 2.2.1 ne sont donc effectuées que lors d'entrées/sorties, c'est-à-dire lorsqu'un programme doit traiter une valeur entrée par un opérateur humain, ou produire un résultat compréhensible par celui-ci.

Nous allons à présent étudier comment effectuer des opérations arithmétiques sur des entiers représentés en notation binaire non signée. Commençons par le cas d'une addition de deux nombres. Nous supposons que chaque opérande ainsi que le résultat de l'opération sont représentés à l'aide de n bits, avec $n \geq 1$.

La procédure consiste à additionner un par un les bits de la représentation des opérandes, du bit de poids faible vers le bit de poids fort. Si le résultat est supérieur à 1, alors il donne naissance à un *report* vers la gauche, qui sera pris en compte à l'étape suivante. Les différents cas de figure possibles sont les suivants, les reports étant entourés d'un rectangle :

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline \boxed{1}\ 0 \end{array}$$

$$\begin{array}{r} \boxed{1} \\ 0 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} \boxed{1} \\ 0 \\ +\ 1 \\ \hline \boxed{1}\ 0 \end{array} \qquad \begin{array}{r} \boxed{1} \\ 1 \\ +\ 0 \\ \hline \boxed{1}\ 0 \end{array} \qquad \begin{array}{r} \boxed{1} \\ 1 \\ +\ 1 \\ \hline \boxed{1}\ 1 \end{array}$$

Exemple : Calcul de la somme 123 + 456 sur 10 bits :

$$\begin{array}{cccccccccc}
 & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & & & & \\
 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
+ & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}$$

□

Pour que cet algorithme d'addition produise un résultat correct, il est nécessaire de disposer d'un nombre de bits suffisant pour représenter ce résultat. Dans l'exemple précédent, chaque opérande appartient à l'intervalle $[0, 2^9 - 1]$, et est donc représentable sur 9 bits. Leur somme est donc inférieure ou égale à $2^{10} - 2$, donc représentable sur 10 bits.

De façon générale, lorsqu'on calcule sur n bits la somme de deux nombres, il arrive que le résultat ne soit pas représentable sur n bits. Cette situation correspond à la présence d'un report à la position n, qui ne peut pas être pris en compte dans le résultat calculé. Cela signifie que la différence entre celui-ci et le résultat correct est égale à 2^n. En d'autres termes, lorsqu'on effectue des additions sur n bits en ignorant les reports aux positions plus grandes que $n - 1$, on travaille dans une arithmétique *modulo* 2^n.

Exemple : Calcul de la somme 123 + 197 sur 8 bits :

$$\begin{array}{cccccccccc}
 & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & & \\
 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
+ & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Le résultat est la représentation binaire non signée sur 8 bits du nombre 64, qui est bien égal modulo 256 à 123 + 197 = 320. □

Pour multiplier deux nombres entiers non signés à partir de leur représentation binaire, la procédure est similaire à celle d'un calcul écrit : On énumère d'abord chaque bit de la deuxième opérande (le *multiplicateur*), du bit de poids faible vers le bit de poids fort. On multiplie la première opérande (le *multiplicande*) par ce bit, de façon à obtenir un *produit partiel* que l'on place à des positions successives décalées vers la gauche. Ensuite, on additionne l'ensemble des produits partiels.

En binaire, ces opérations sont faciles à effectuer, car multiplier un nombre par 0 produit un résultat nul, et le multiplier par 1 le laisse inchangé. Il n'est donc pas nécessaire de mémoriser des tables de multiplication !

Exemple : Calcul du produit 12×34 :

$$
\begin{array}{cccccccccc}
 & & & & & & 1 & 1 & 0 & 0 \\
\times & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
 & & & & & & 0 & 0 & 0 & 0 \\
 & & & & & 1 & 1 & 0 & 0 & \\
 & & & & 0 & 0 & 0 & 0 & & \\
 & & & 0 & 0 & 0 & 0 & & & \\
 & & 0 & 0 & 0 & 0 & & & & \\
+ & 1 & 1 & 0 & 0 & & & & & \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0
\end{array}
$$

□

Remarques :

— En général, le produit de deux nombres représentés respectivement sur n_1 et n_2 bits requiert $n_1 + n_2$ bits. Si le nombre de bits n choisi pour représenter le résultat est inférieur à cette valeur, alors seuls les n bits de poids faible de celle-ci sont calculés, ce qui correspond à effectuer la multiplication en arithmétique modulo 2^n.

— Lorsqu'on additionne les produits partiels, il peut arriver que la somme des bits à une position donnée devienne supérieure à 3, ce qui peut conduire à plusieurs reports simultanés, ou a des reports vers des positions plus lointaines.

Par exemple, si la somme des bits à la position k est égale à 6, alors le résultat est noté 0 et deux reports sont effectués aux positions $k + 1$ et $k + 2$, car 6 admet la représentation binaire 110 :

$$
\begin{array}{cc}
\boxed{1}\,\boxed{1} & \\
 & 1 \\
 & 1 \\
 & 1 \\
 & 1 \\
 & 1 \\
+ & 1 \\
\hline
 & 0
\end{array}
$$

□

2.3 Les nombres entiers signés

Il existe plusieurs façons de représenter en binaire les nombres entiers pouvant être positifs ou négatifs. Nous allons étudier trois solutions appelées *valeur signée*, *complément à un* et *complément à deux*. Ces trois types de représentation possèdent les caractéristiques communes suivantes :

— Le bit de poids fort correspond toujours au signe du nombre représenté, et est dès lors appelé *bit de signe*. Celui-ci vaut 0 pour les nombres positifs et 1 pour les nombres négatifs. Le cas du nombre zéro est particulier et sera discuté séparément pour chaque représentation.

— Si la représentation d'un nombre commence par un bit de signe égal à 0, alors elle est identique à la représentation binaire non signée du même nombre.

Il est indispensable, lorsqu'on manipule des nombres encodés en binaire, de connaître le système de représentation utilisé. Par exemple, l'octet 11001000 représente le nombre 200 selon l'encodage non signé, mais trois autres nombres, négatifs, par valeur signée, complément à un et complément à deux. Seules les représentations commençant par un bit égal à 0 définissent les mêmes nombres dans les quatre systèmes.

2.3.1 La représentation par valeur signée

La représentation par *valeur signée* est celle qui ressemble le plus au procédé habituellement utilisé pour noter les nombres signés en base décimale. Elle consiste à encoder un nombre négatif en préfixant sa valeur absolue d'un symbole particulier. En base 10, nous utilisons le symbole "−" : Le nombre −42 s'écrit en plaçant ce symbole avant l'écriture positionnelle 42 de sa valeur absolue.

Les représentations binaires utilisées en informatique ne fournissent que les deux symboles 0 et 1. Plutôt que d'introduire un symbole supplémentaire, il est plus simple d'utiliser le bit de signe et de faire suivre celui-ci de la représentation non signée de la valeur absolue du nombre.

Exemple : La représentation par valeur signée sur 8 bits du nombre −42 est égale à 10101010. En effet :

— Ce nombre est négatif, donc le bit de signe est égal à 1.

— La valeur absolue 42 du nombre possède une représentation non signée sur 7 bits égale à 0101010. □

Nous allons à présent chercher à déterminer le nombre v représenté par la suite de bits $b_{n-1}b_{n-2}\dots b_0$. Il y a deux cas à considérer :

— *Si* $b_{n-1} = 0$: On a $v = \sum_{i=0}^{n-2} 2^i\, b_i$.

— *Si* $b_{n-1} = 1$: On a $v = -\sum_{i=0}^{n-2} 2^i\, b_i$.

On obtient donc dans le deux cas la même somme (correspondant à la valeur absolue du nombre) multipliée par un facteur égal à 1 si $b_{n-1} = 0$ et à -1 si $b_{n-1} = 1$. Il est donc facile d'unifier ces deux expressions en une seule, valable quel que soit le bit de signe :

$$v = (1 - 2b_{n-1}) \sum_{i=0}^{n-2} 2^i \, b_i. \tag{2.2}$$

Si l'on dispose de n bits, la représentation par valeur signée permet d'encoder

— tous les nombres de l'intervalle $[0, 2^{n-1} - 1]$ avec un bit de signe égal à 0, et

— tous les nombres de l'intervalle $[-2^{n-1} + 1, 0]$ avec un bit de signe égal à 1.

L'ensemble des nombres représentables forme donc en définitive l'intervalle

$$[-2^{n-1} + 1, 2^{n-1} - 1].$$

On voit que le nombre zéro possède deux encodages sur n bits : $0000\ldots0$ (*zéro positif*) et $1000\ldots0$ (*zéro négatif*). Cette propriété peut être exploitée par certaines applications, par exemple pour préserver le signe d'une quantité arrondie vers un entier. Néanmoins, dans la plupart des cas, l'existence de deux représentations de zéro pour les nombres entiers est une difficulté qui complique les calculs arithmétiques.

Un autre inconvénient de la représentation par valeur signée est que, pour effectuer une addition de deux nombres, il faut implémenter deux opérations différentes : Additionner les valeurs absolues lorsque les deux nombres sont de même signe, et les soustraire lorsqu'ils sont de signe différent. Les représentations que nous allons introduire dans les sections 2.3.2 et 2.3.3 ont pour objectif de pallier ce défaut.

2.3.2 La représentation par complément à un

La représentation par complément à un est une variante de la représentation par valeur signée : Lorsque le bit de signe est égal à 1, plutôt que de faire suivre ce bit par l'encodage non signé de la valeur absolue du nombre, on *complémente* cet encodage, c'est-à-dire que l'on y remplace les bits égaux à 0 par des bits égaux à 1, et vice-versa.

Exemple : La représentation par complément à un sur 8 bits du nombre -42 vaut 11010101. En effet :

— Ce nombre est négatif, donc le bit de signe est égal à 1.

— La valeur absolue 42 du nombre possède la représentation non signée sur 7 bits 0101010, qui se complémente en 1010101. □

Comme dans la section précédente, nous allons à présent chercher à dériver une expression pour le nombre v représenté par la suite de bits $b_{n-1}b_{n-2}\ldots b_0$.

— *Si* $b_{n-1} = 0$: On a $v = \sum_{i=0}^{n-2} 2^i \, b_i = \sum_{i=0}^{n-1} 2^i \, b_i$.

— *Si* $b_{n-1} = 1$: Pour obtenir la représentation non signée sur $n-1$ bits de $|v|$, il faut complémenter chaque bit b_i pour $i \in [0, n-2]$, c'est-à-dire remplacer b_i par $1 - b_i$. On a donc

$$\begin{aligned} |v| &= \sum_{i=0}^{n-2} 2^i (1 - b_i) \\ &= \sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} 2^i b_i \\ &= \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 2^i b_i \\ &= 2^n - 1 - \sum_{i=0}^{n-1} 2^i b_i \end{aligned}$$

et donc $v = -2^n + 1 + \sum_{i=0}^{n-1} 2^i b_i$.

Contrairement au cas de la valeur signée, le terme $\sum_{i=0}^{n-1} 2^i b_i$ apparaît maintenant avec le même signe dans les expressions relatives à $b_{n-1} = 0$ et $b_{n-1} = 1$. La seule différence entre ces expressions réside dans un terme additionnel $-2^n + 1$ présent pour les nombres négatifs. Il est donc facile d'unifier les deux expressions :

$$v = (-2^n + 1) b_{n-1} + \sum_{i=0}^{n-1} 2^i b_i. \tag{2.3}$$

L'ensemble des nombres représentables à l'aide de n bits est inchangé par rapport à la représentation par valeur signée ; il s'agit toujours de l'intervalle

$$[-2^{n-1} + 1, 2^{n-1} - 1].$$

Le nombre zéro possède toujours deux représentations, qui prennent maintenant la forme $0000\ldots0$ (*zéro positif*) et $11111\ldots1$ (*zéro négatif*). Remarquons que pour calculer l'opposé d'un nombre, il suffit de complémenter l'ensemble des bits de sa représentation. Les zéros positif et négatif constituent un cas particulier d'application de cette règle.

Examinons maintenant comment calculer la somme de deux nombres entiers signés représentés par complément à un. L'idée consiste à essayer de réutiliser l'algorithme développé dans la section 2.2.3 pour la représentation non signée, en étudiant le résultat qu'il fournit lorsqu'on l'applique à des représentations par complément à un. Nous utiliserons les notations suivantes. Soit $w = b_{n-1} b_{n-2} \ldots b_0$ une suite de n bits, avec $n \geq 1$. Le nombre dont w est une représentation non signée est noté $[w]_{ns}$. De même, le nombre dont w est une représentation par complément à un est noté $[w]_{c_1}$. On a ainsi, par exemple,

$$\begin{aligned}[11010101]_{ns} &= 213\\ [11010101]_{c_1} &= -42.\end{aligned}$$

Comme le montrent les expressions (2.1) et (2.3), les deux interprétations $[w]_{ns}$ et $[w]_{c_1}$ sont liées par la propriété suivante :

— Si $b_{n-1} = 0$, alors $[w]_{ns} = [w]_{c_1}$.

— Si $b_{n-1} = 1$, alors $[w]_{ns} = [w]_{c_1} + 2^n - 1$.

Cela signifie que les représentations non signées et par complément à un des nombres sont égales à un certain décalage près. Ce décalage est nul pour un bit de signe égal à 0, et égal à $2^n - 1$ pour un bit de signe égal à 1.

Lorsqu'on applique l'algorithme d'addition de la section 2.2.3 aux deux suites de bits $w = b_{n-1}b_{n-2}\dots b_0$ et $w' = b'_{n-1}b'_{n-2}\dots b'_0$, on obtient un résultat $w'' = b''_{n-1}b''_{n-2}\dots b''_0$ tel que

$$\begin{cases} [w'']_{ns} = [w]_{ns} + [w']_{ns} & \text{si aucun report n'est apparu à la position } n,\\ [w'']_{ns} = [w]_{ns} + [w']_{ns} - 2^n & \text{si un report est apparu à la position } n.\end{cases}$$

(En effet, dans le deuxième cas, ce report est ignoré, ce qui retire 2^n au résultat.)

Notre objectif consiste à déterminer la relation qui lie $[w]_{c_1}$, $[w']_{c_1}$ et $[w'']_{c_1}$. Nous savons que ces nombres sont respectivement égaux à $[w]_{ns}$, $[w']_{ns}$ et $[w'']_{ns}$ à un certain décalage près, ce décalage dépendant du bit de signe de w, w' et w''. Par conséquent, la somme $[w]_{c_1} + [w']_{c_1}$, qui est celle que nous souhaitons calculer, est égale à $[w'']_{c_1}$ à un certain décalage près. La question consiste à calculer précisément la valeur de ce décalage, afin de pouvoir si nécessaire le corriger en vue d'obtenir le bon résultat. Plusieurs cas de figure sont à examiner :

— *Si* $b_{n-1} = 0$, $b'_{n-1} = 0$ *et* $b''_{n-1} = 0$: Dans ce cas, on a $[w]_{ns} = [w]_{c_1}$, $[w']_{ns} = [w']_{c_1}$ et $[w'']_{ns} = [w'']_{c_1}$. Aucun report n'a pu être produit à la position n. Cela entraîne

$$\begin{aligned}[w'']_{c_1} &= [w'']_{ns}\\ &= [w]_{ns} + [w']_{ns}\\ &= [w]_{c_1} + [w']_{c_1}.\end{aligned}$$

L'opération retourne donc le résultat attendu.

— *Si* $b_{n-1} = 0$, $b'_{n-1} = 0$ *et* $b''_{n-1} = 1$: Nous avons alors $[w]_{ns} = [w]_{c_1} \geq 0$ et $[w']_{ns} = [w']_{c_1} \geq 0$. La somme de ces deux nombres est supérieure ou égale à 2^{n-1}, puisque le bit de poids fort b''_{n-1} du résultat est égal à 1. On en déduit que cette somme n'est pas représentable sur n bits par la méthode du complément à un. Il s'agit donc d'un *dépassement arithmétique (arithmetic overflow)* : Le nombre de bits choisi pour effectuer l'opération n'est pas suffisant pour en représenter le résultat.

— *Si* $b_{n-1} = 0$, $b'_{n-1} = 1$ *et* $b''_{n-1} = 0$ *:* Alors, on a $[w]_{ns} = [w]_{c_1}$, $[w']_{ns} = [w']_{c_1} + 2^n - 1$ et $[w'']_{ns} = [w'']_{c_1}$. Un report a nécessairement été produit à la position n. On a donc

$$\begin{aligned}[w'']_{c_1} &= [w'']_{ns} \\ &= [w]_{ns} + [w']_{ns} - 2^n \\ &= [w]_{c_1} + [w']_{c_1} - 1.\end{aligned}$$

L'opération retourne donc le résultat attendu moins un.

— *Si* $b_{n-1} = 0$, $b'_{n-1} = 1$ *et* $b''_{n-1} = 1$: Alors, on a $[w]_{ns} = [w]_{c_1}$, $[w']_{ns} = [w']_{c_1} + 2^n - 1$ et $[w'']_{ns} = [w'']_{c_1} + 2^n - 1$. Il n'est pas possible qu'un report soit apparu à la position n. On a donc

$$\begin{aligned}[w'']_{c_1} &= [w'']_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^n + 1 \\ &= [w]_{c_1} + [w']_{c_1}.\end{aligned}$$

Le résultat de l'opération est donc correct.

— *Si* $b_{n-1} = 1$, $b'_{n-1} = 0$ *:* Ce cas de figure est similaire aux deux cas précédents, en permutant les deux opérandes w et w'.

— *Si* $b_{n-1} = 1$, $b'_{n-1} = 1$ *et* $b''_{n-1} = 0$: Dans ce cas $[w]_{ns} = [w]_{c_1} + 2^n - 1$ et $[w']_{ns} = [w']_{c_1} + 2^n - 1$, avec $[w]_{c_1} \leq 0$ et $[w']_{c_1} \leq 0$. L'addition $[w]_{ns} + [w']_{ns}$ produit un report à la position n, et son résultat possède un bit de poids fort b''_{n-1} qui est nul. On a donc

$$[w]_{ns} + [w']_{ns} \leq 2^n + 2^{n-1} - 1.$$

En remplaçant les expressions de $[w]_{ns}$ et $[w']_{ns}$, on obtient

$$[w]_{ns} + [w']_{ns} = [w]_{c_1} + [w']_{c_1} + 2^{n+1} - 2$$

qui entraîne

$$\begin{aligned}[w]_{c_1} + [w']_{c_1} &\leq 2^n + 2^{n-1} - 1 - 2^{n+1} + 2 \\ &= -2^{n-1} + 1.\end{aligned}$$

Les nombres représentables par complément à un à l'aide de n bits sont supérieurs ou égaux à $-2^{n-1} - 1$. La somme $[w]_{c_1} + [w']_{c_1}$ n'est donc représentable que dans le cas particulier où elle est égale à $-2^{n-1} + 1$; toutes les autres situations correspondent à un dépassement arithmétique. Dans un premier temps, nous ne considérerons pas ce cas particulier d'un résultat égal à $-2^{n-1} + 1$. Nous verrons dans un deuxième temps comment il peut être traité.

— *Si* $b_{n-1} = 1$, $b'_{n-1} = 1$ *et* $b''_{n-1} = 1$: On a alors $[w]_{ns} = [w]_{c_1} + 2^n - 1$, $[w']_{ns} = [w']_{c_1} + 2^n - 1$ et $[w'']_{ns} = [w'']_{c_1} + 2^n - 1$. Un report a nécessairement été effectué à la position n. On a

donc

$$\begin{aligned}[w'']_{c_1} &= [w'']_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^{n+1} + 1 \\ &= [w]_{c_1} + [w']_{c_1} - 1.\end{aligned}$$

Le résultat est donc celui qui était attendu moins un.

Nous pouvons donc résumer cette longue discussion par la propriété suivante. Si l'on additionne deux nombres de n bits représentés par complément à un par le même algorithme que celui développé pour les nombres non signés :

- Si les deux opérandes présentent le même bit de signe, et si celui-ci diffère de celui du résultat, alors un dépassement arithmétique s'est produit.
- Sinon :
  - Si aucun report n'est apparu à la position n, alors le résultat est correct.
  - Si un tel report est apparu, alors le résultat est trop petit d'une unité.

Cette propriété mène à un algorithme relativement simple pour additionner deux nombres de n bits représentés par complément à un :

1. Additionner ces deux nombres sur n bits comme s'ils étaient représentés de façon non signée.
2. Si un report est apparu à la position n lors de l'opération précédente, effectuer une deuxième opération d'addition afin d'ajouter 1 au résultat.
3. Détecter un dépassement arithmétique éventuel en examinant le bit de signe des opérandes et du résultat.

Remarque : Le test de dépassement arithmétique est effectué *après* la correction éventuelle du résultat afin de traiter correctement le cas particulier d'un résultat égal à $-2^{n-1} + 1$, que nous avons négligé dans un premier temps. En effet, dans ce cas, le résultat ne devient représentable qu'après avoir appliqué la correction. Cette situation particulière correspond à l'exemple 7 ci-dessous.

□

Exemples : Nous allons illustrer les différentes situations pouvant se présenter grâce à quelques exemples. Pour chacun d'entre eux, les interprétations non signée et par complément à un des séquences de bits manipulées sont indiquées, ainsi que leur différence. Le nombre de bits n est fixé à 8 pour tous ces exemples.

1. Calcul de 12 + 34 : $b_{n-1} = 0$ et $b'_{n-1} = 0$.

									n. sign.	compl. un	diff.
	0	0	0	0	1	1	0	0	12	**12**	0
+	0	0	1	0	0	0	1	0	34	**34**	0
	0	0	1	0	1	1	1	0	46	**46**	0

2. Calcul de 120 + 34 : $b_{n-1} = 0$ et $b'_{n-1} = 0$, dépassement arithmétique.

	$\boxed{1}$	$\boxed{1}$							n. sign.	compl. un	diff.
	0	1	1	1	1	0	0	0	120	**120**	0
+	0	0	1	0	0	0	1	0	34	**34**	0
	1	0	0	1	1	0	1	0	154	~~**101**~~	255

3. Calcul de 34 + (−12) : $b_{n-1} = 0$ et $b'_{n-1} = 1$, résultat positif.

$\boxed{1}$	$\boxed{1}$	$\boxed{1}$				$\boxed{1}$			n. sign.	compl. un	diff.
	0	0	1	0	0	0	1	0	34	**34**	0
+	1	1	1	1	0	0	1	1	243	**−12**	255
	0	0	0	1	0	1	0	1	21	21	0
+	0	0	0	0	0	0	0	1	1	1	0
	0	0	0	1	0	1	1	0	22	**22**	0

4. Calcul de 12 + (−34) : $b_{n-1} = 0$ et $b'_{n-1} = 1$, résultat négatif.

			$\boxed{1}$	$\boxed{1}$	$\boxed{1}$				n. sign.	compl. un	diff.
	0	0	0	0	1	1	0	0	12	**12**	0
+	1	1	0	1	1	1	0	1	221	**−34**	255
	1	1	1	0	1	0	0	1	233	**−22**	255

5. Calcul de (−34) + (−12) : $b_{n-1} = 1$ et $b'_{n-1} = 1$.

$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$		n. sign.	compl. un	diff.
	1	1	0	1	1	1	0	1	221	**−34**	255
+	1	1	1	1	0	0	1	1	243	**−12**	255
	1	1	0	1	0	0	0	0	208	−47	255
+	0	0	0	0	0	0	0	1	1	1	0
	1	1	0	1	0	0	0	1	209	**−46**	255

6. Calcul de $(-34) + (-120)$: $b_{n-1} = 1$ et $b'_{n-1} = 1$, dépassement arithmétique.

									n. sign.	compl. un	diff.
$\boxed{1}$			$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$				
	1	1	0	1	1	1	0	1	221	**−34**	255
+	1	0	0	0	0	1	1	1	135	**−120**	255
	0	1	1	0	0	1	0	0	100	100	0
+	0	0	0	0	0	0	0	1	1	1	0
	0	1	1	0	0	1	0	1	101	~~**101**~~	0

7. Calcul de $(-120)+(-7)$: $b_{n-1} = 1$ et $b'_{n-1} = 1$, cas particulier d'un résultat égal à $-2^{n-1}+1$.

									n. sign.	compl. un	diff.
$\boxed{1}$											
	1	0	0	0	0	1	1	1	135	**−120**	255
+	1	1	1	1	1	0	0	0	248	**−7**	255
	0	1	1	1	1	1	1	1	127	127	0
+	0	0	0	0	0	0	0	1	1	1	0
	1	0	0	0	0	0	0	0	128	**−127**	255

□

Par rapport à la valeur signée, la représentation par complément à un présente l'avantage de pouvoir additionner des nombres positifs et négatifs à l'aide d'un seul algorithme de calcul, identique à celui développé pour les nombres non signés. Cette représentation possède cependant deux inconvénients. Premièrement, dans certains cas, une correction doit être appliquée au résultat de l'opération, ce qui requiert une opération d'addition supplémentaire. Ensuite, tout comme la valeur signée, la représentation par complément à un définit deux représentations distinctes de zéro. Dans la section suivante, nous allons introduire une variante de cette représentation qui ne possède pas ces deux défauts.

2.3.3 La représentation par complément à deux

L'idée à la base de la représentation par complément à deux est de décaler la représentation des nombres négatifs d'une unité par rapport au complément à un, de façon à ne plus avoir besoin de terme correctif lorsqu'on effectue des additions. Dans ce système, la représentation sur n bits du nombre v est définie comme

— la représentation non signée de v sur n bits si $v \geq 0$,

— la représentation par complément à un de $v + 1$ sur n bits si $v < 0$. (Si $v + 1$ est nul, on en choisit la représentation négative.)

Exemples :

— La représentation par complément à deux sur 8 bits du nombre -42 vaut 11010110. En

effet, ce nombre est négatif, donc la représentation recherchée est celle de -41 par complément à un, qui est bien égale à 11010110.

— La représentation par complément à deux sur n bits du nombre -1 vaut $111\ldots1$. En effet, cette représentation correspond au zéro négatif du complément à un. □

Pour déterminer l'entier v encodé par la suite de bits $w = b_{n-1}b_{n-2}\ldots b_0$, deux cas sont à considérer :

— *Si* $b_{n-1} = 0$: Alors, le résultat est identique à celui d'une représentation non signée, c'est-à-dire

$$v = \sum_{i=0}^{n-1} 2^i\, b_i.$$

— *Si* $b_{n-1} = 1$: Dans ce cas, la représentation est celle de $v + 1$ par complément à un. En utilisant l'expression (2.3), on obtient donc

$$v + 1 = -2^n + 1 + \sum_{i=0}^{n-1} 2^i\, b_i,$$

c'est-à-dire

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

On a donc de façon générale

$$v = -2^n\, b_{n-1} + \sum_{i=0}^{n-1} 2^i\, b_i. \tag{2.4}$$

Quel que soit le bit de signe, on remarque que les nombres encodés par w dans les représentations non signée et par complément à deux sont égaux *modulo* 2^n. Formellement, on a

$$[w]_{ns} =_{2^n} [w]_{c_2}, \tag{2.5}$$

où $[w]_{c_2}$ est le nombre encodé par la suite de bits w en complément à deux, et "$=_m$" dénote l'"égalité modulo m.

L'ensemble des nombres représentables par complément à deux à l'aide de n bits forme l'intervalle

$$[-2^{n-1}, 2^{n-1} - 1].$$

Par rapport au complément à un, on remarque que les nombres négatifs sont décalés d'une unité. De plus, le nombre zéro ne possède plus qu'une seule représentation, entièrement composée de bits nuls.

La représentation par complément à deux présente d'autres propriétés intéressantes :

— *On peut étendre la représentation d'un nombre vers davantage de bits en en répétant le bit de signe.*

Cette propriété est évidente dans le cas d'un nombre positif ou nul, car ajouter un zéro en tête de sa représentation n'influence clairement pas ce nombre.

Dans le cas d'un nombre négatif, si l'on passe de $b_{n-1}b_{n-2}\dots b_0$ à $b_n b_{n-1}b_{n-2}\dots b_0$ en posant $b_n = b_{n-1} = 1$, le nombre représenté devient égal à

$$\begin{aligned} -2^{n+1} + \sum_{i=0}^{n} 2^i\, b_i &= -2^{n+1} + 2^n + \sum_{i=0}^{n-1} 2^i\, b_i \\ &= -2^n + \sum_{i=0}^{n-1} 2^i\, b_i \end{aligned}$$

qui correspond bien à la valeur de départ.

— *La représentation d'un nombre v se termine par k bits nuls si et seulement si v est divisible par 2^k. En particulier, v est pair si et seulement si son bit de poids faible est nul.*

En effet, cette propriété est vraie pour la représentation non signée, et l'est donc également pour le complément à deux, puisque les nombres correspondants ne diffèrent que d'un multiple entier de 2^n, où n est le nombre de bits utilisés. Ces nombres diffèrent donc également d'un multiple entier de 2^k pour $k \leq n$.

— *Pour calculer l'opposé d'un nombre, il suffit de complémenter l'ensemble des bits de sa représentation, et d'ajouter ensuite* 1 *au résultat.*
Considérons le nombre v encodé par la suite de bits $b_{n-1}b_{n-2}\dots b_0$. Si l'on complémente l'ensemble de ces bits, le nombre représenté devient égal modulo 2^n à

$$\sum_{i=0}^{n-1} 2^i\,(1 - b_i)$$

Soit v' le nombre obtenu en ajoutant 1 à cette représentation. On a donc bien

$$\begin{aligned} v' &=_{2^n} 1 + \sum_{i=0}^{n-1} 2^i\,(1 - b_i) \\ &=_{2^n} 1 + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 2^i\, b_i \\ &=_{2^n} 2^n - \sum_{i=0}^{n-1} 2^i\, b_i \\ &=_{2^n} -\sum_{i=0}^{n-1} 2^i\, b_i \\ &=_{2^n} -v. \end{aligned}$$

La somme de deux nombres peut facilement être calculée à partir de leur représentation par complément à deux. Considérons, de la même manière que dans la section 2.3.2, deux suites de bits $w = b_{n-1}b_{n-2}\dots b_0$ et $w' = b'_{n-1}b'_{n-2}\dots b'_0$ dont l'addition non signée produit le résultat $w'' = b''_{n-1}b''_{n-2}\dots b''_0$. Dans la section 2.2.3, on a établi

$$[w'']_{ns} =_{2^n} [w]_{ns} + [w']_{ns}.$$

En introduisant (2.5), on obtient alors

$$[w'']_{c_2} =_{2^n} [w]_{c_2} + [w']_{c_2}.$$

Ce résultat signifie que le même algorithme peut être employé, sans correction, pour additionner des nombres représentés de façon non signée et par complément à deux. Dans les deux cas, et même si l'on mélange les deux représentations au sein d'une même opération, la somme est calculée dans une arithmétique modulo 2^n, où n est le nombre de bits utilisés pour représenter les nombres.

Exemples : Nous illustrons l'opération d'addition de deux nombres représentés par complément à deux à l'aide des mêmes exemples que dans la section 2.3.2. Le nombre de bits utilisés pour représenter les nombres est toujours fixé à 8.

1. Calcul de 12 + 34 :

									n. sign.	compl. deux	diff.
	0	0	0	0	1	1	0	0	12	**12**	0
+	0	0	1	0	0	0	1	0	34	**34**	0
	0	0	1	0	1	1	1	0	46	**46**	0

2. Calcul de 120 + 34 (dépassement arithmétique) :

									n. sign.	compl. deux	diff.
	1	1									
	0	1	1	1	1	0	0	0	120	**120**	0
+	0	0	1	0	0	0	1	0	34	**34**	0
	1	0	0	1	1	0	1	0	154	**−102**	256

3. Calcul de 34 + (−12) :

									n. sign.	compl. deux	diff.
1	1	1									
	0	0	1	0	0	0	1	0	34	**34**	0
+	1	1	1	1	0	1	0	0	244	**−12**	256
	0	0	0	1	0	1	1	0	22	**22**	0

4. Calcul de 12 + (−34) :

									n. sign.	compl. deux	diff.
			1	1	1						
	0	0	0	0	1	1	0	0	12	**12**	0
+	1	1	0	1	1	1	1	0	222	**−34**	256
	1	1	1	0	1	0	1	0	234	**−22**	256

5. Calcul de (−34) + (−12) :

1	1	1	1	1	1				n. sign.	compl. deux	diff.
	1	1	0	1	1	1	1	0	222	**−34**	256
+	1	1	1	1	0	1	0	0	244	**−12**	256
	1	1	0	1	0	0	1	0	210	**−46**	256

6. Calcul de (−34) + (−120) (dépassement arithmétique) :

1			1	1					n. sign.	compl. deux	diff.
	1	1	0	1	1	1	1	0	222	**−34**	256
+	1	0	0	0	1	0	0	0	136	**−120**	256
	0	1	1	0	0	1	1	0	102	**102**	0

7. Calcul de (−120) + (−7) :

1	1	1	1	1					n. sign.	compl. deux	diff.
	1	0	0	0	1	0	0	0	136	**−120**	256
+	1	1	1	1	1	0	0	1	249	**−7**	256
	1	0	0	0	0	0	0	1	129	**−127**	256

□

Le produit de deux nombres représentés par complément à deux se calcule essentiellement de la même façon que pour les nombres non signés (cf. section 2.2.3). La seule précaution à prendre est de veiller à étendre les nombres manipulés sur l'ensemble des bits disponibles, en en répétant le bit de signe le nombre approprié de fois.

Exemples :

— Calcul du produit $(-12) \times 34$ sur 12 bits :

	1	1	1	1	1	1	1	1	0	1	0	0
×	0	0	0	0	0	0	1	0	0	0	1	0
	1	1	1	1	1							
	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	0	1	0	0	
	0	0	0	0	0	0	0	0	0	0		
	0	0	0	0	0	0	0	0	0			
	0	0	0	0	0	0	0	0				
	1	1	1	0	1	0	0					
	0	0	0	0	0	0						
	0	0	0	0	0							
	0	0	0	0								
	0	0	0									
	0	0										
+	0											
	1	1	1	0	0	1	1	0	1	0	0	0

— Calcul du produit $(-12) \times (-34)$ sur 12 bits :

		1	1	1	1	1	1	1	1	0	1	0	0
	×	1	1	1	1	1	1	0	1	1	1	1	0
			1		1								
1	1		1	1	1	1		1					
		0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	1	1	1	1	0	1	0	0	
		1	1	1	1	1	1	0	1	0	0		
		1	1	1	1	1	0	1	0	0			
		1	1	1	1	0	1	0	0				
		0	0	0	0	0	0	0					
		1	1	0	1	0	0						
		1	0	1	0	0							
		0	1	0	0								
		1	0	0									
		0	0										
	+	0											
		0	0	0	1	1	0	0	1	1	0	0	0

□

2.3.4 Récapitulatif

Dans les sections précédentes, nous avons étudié quatre représentations des nombres entiers : non signée, par valeur signée, par complément à un et par complément à deux. Lorsque le bit de poids fort (correspondant au bit de signe dans le cas de nombres signés) est égal à 0, ces quatre représentations coïncident. Lorsqu'il est égal à 1, elles sont en général différentes. Il est donc indispensable de connaître précisément le système de représentation utilisé lorsqu'on cherche à décoder un nombre ! A titre d'exemple, ce tableau fournit un récapitulatif des différents encodages des suites de 4 bits :

représentation	non signé	val. sign.	compl. un	compl. deux
0000	0	+0	+0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	−0	−7	−8
1001	9	−1	−6	−7
1010	10	−2	−5	−6
1011	11	−3	−4	−5
1100	12	−4	−3	−4
1101	13	−5	−2	−3
1110	14	−6	−1	−2
1111	15	−7	−0	−1

2.3.5 Les nombres entiers en programmation

Des trois procédés de représentation de nombres signés que nous avons étudiés, le complément à deux est celui qui présente le plus d'avantages. Pour cette raison, il est employé dans une grande majorité des applications.

Dans des langages de programmation comme C ou C++, l'encodage des types entiers n'est pas entièrement standardisé, mais dépend de l'environnement de programmation utilisé. Sur un PC moderne, les types `char`, `short`, `int` et `long` correspondent habituellement à des représentations par complément à deux sur respectivement 8, 16, 32 et 64 bits. Si ces types sont accompagnés du qualificateur `unsigned`, ils deviennent non signés.

Dans ces langages, la plupart des opérations impliquant les entiers s'effectuent dans une arithmétique modulo 2^n, où n est le nombre de bits utilisés. La distinction entre types signés et non signés présente alors peu d'importance, puisque le résultat des opérations est identique dans les deux cas. Cette distinction est cependant significative pour les opérations de *comparaison*.

Par exemple, pour les deux octets $w = 10101010$ et $w' = 01010101$, on a $[w]_{ns} > [w']_{ns}$, mais $[w]_{c_2} < [w']_{c_2}$.

D'autres langages de programmation tels que Java imposent une représentation spécifique des nombres entiers. Dans ce langage, les types `byte`, `short`, `int` et `long` correspondent à des représentations par complément à deux sur respectivement 8, 16, 32 et 64 bits, indépendamment des caractéristiques de l'ordinateur utilisé.

2.4 Les nombres réels

Beaucoup d'applications informatiques sont amenées à devoir manipuler des nombres réels, que ce soit pour représenter des quantités physiques comme des vitesses, des masses ou des durées, des sommes d'argent, des coordonnées ou encore des probabilités. Le problème consistant à représenter des nombres réels à l'aide d'une suite de bits se heurte cependant à une difficulté majeure : Encoder un réel arbitraire, même s'il est borné, nécessite une quantité infinie d'information. Il est bien sûr possible de représenter de façon finie certains réels, comme 0, 42, -1, $\sqrt{2}$ ou π, mais il ne s'agit que de cas très particuliers. Pour des raisons fondamentales [3], quel que soit le procédé de représentation employé, la probabilité qu'un nombre tiré au hasard uniformément dans $\mathbb{R}$ soit représentable est égale à 0. En d'autres termes, les nombres représentables forment un sous-ensemble négligeable des réels.

Heureusement, dans la plupart des cas, on peut se contenter de représenter les réels avec une précision limitée. Si l'on calcule, par exemple, la circonférence d'une roue de voiture à partir de son rayon en employant l'approximation $\pi \approx 3,14159265359$, l'erreur commise par rapport au même calcul effectué avec la valeur exacte de π reste largement inférieure à la taille d'un atome.

Lorsqu'on programme des opérations impliquant des nombres réels, il faut donc toujours garder à l'esprit que ceux-ci sont représentés de façon approximative. La plupart des opérations arithmétiques ont pour effet d'augmenter l'imprécision des nombres ; une longue chaîne de traitements peut ainsi produire un résultat très éloigné de la valeur exacte. La question de déterminer si le résultat d'un calcul est correct à un niveau de précision donné n'est pas simple. Elle est abordée dans des cours tels qu'*Analyse Numérique*.

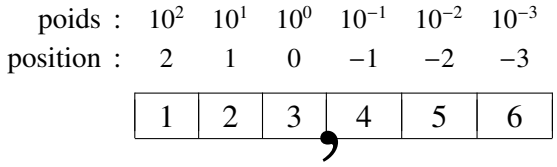
2.4.1 La représentation en virgule fixe

Dans la vie quotidienne, nous écrivons les nombres réels grâce à une variante de la notation positionnelle introduite à la section 2.2.1. Dans cette variante, un symbole supplémentaire appelé *séparateur*, prenant la forme d'une virgule ou d'un point, est introduit entre les parties *entière* (à

3. Les nombres représentables par un procédé donné, quel qu'il soit, sont nécessairement *dénombrables*, c'est-à-dire qu'il doit être possible de les énumérer un par un sans en manquer aucun. L'ensemble $\mathbb{R}$ n'est quant à lui pas dénombrable.

gauche) et *fractionnaire* (à droite) des représentations. La partie entière est exprimée de la même façon qu'un nombre entier. Les chiffres de la partie fractionnaire se voient quant à eux attribuer un poids correspondant à une puissance négative de la base.

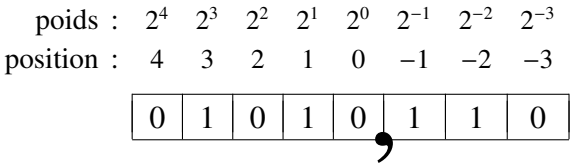
Par exemple, pour le nombre 123,456, on obtient



$$\begin{aligned} & 1\times 10^2 + 2\times 10^1 + 3\times 10^0 + 4\times 10^{-1} + 5\times 10^{-2} + 6\times 10^{-3} \\ &= 1\times 100 + 2\times 10 + 3\times 1 + 4\times \frac{1}{10} + 5\times \frac{1}{100} + 6\times \frac{1}{1000} \\ &= 123{,}456. \end{aligned}$$

Lorsqu'on fixe le nombre de chiffres de la représentation ainsi que la position du séparateur dans celle-ci, le procédé obtenu porte le nom de représentation en *virgule fixe*. On l'utilise lorsque la précision avec laquelle les nombres doivent être représentés est constante et connue. Par exemple, les applications bancaires utilisent la virgule fixe pour représenter les sommes d'argent avec une précision d'un centime.

Il est bien sûr possible de mettre en œuvre la virgule fixe dans d'autres bases que 10, en particulier en base 2. Par exemple, avec 3 bits après la virgule, l'octet 01010110 représente le nombre 10,75. En effet, on a



$$\begin{aligned} & 0\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2} + 0\times 2^{-3} \\ &= 8 + 2 + \frac{1}{2} + \frac{1}{4} \\ &= 10{,}75. \end{aligned}$$

Il est facile d'établir un lien entre la représentation des réels en virgule fixe et la représentation des entiers : Si l'on retire la virgule, on obtient un nombre entier 2^k fois plus grand que le nombre d'origine, où k est le nombre de chiffres après la virgule. Pour l'exemple précédent, on a ainsi

poids :	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
position :	7	6	5	4	3	2	1	0
	0	1	0	1	0	1	1	0

$$\begin{aligned} & 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 64 + 16 + 4 + 2 \\ &= 86. \end{aligned}$$

Le nombre obtenu 86 est 8 fois plus grand que 10,75, puisque le fait de retirer la virgule a eu pour effet de multiplier le poids de chaque bit par 2^3.

Nous venons d'établir que la représentation binaire en virgule fixe avec k bits après la virgule est équivalente à une représentation entière, à un facteur $\frac{1}{2^k}$ près. Toutes les représentations entières que nous avons étudiées dans les sections 2.2 et 2.3 peuvent potentiellement être employées : On obtient ainsi des représentations en virgule fixe non signée, par valeur signée, par complément à un et par complément à deux. En pratique, nous avons vu à la section 2.3.5 que les représentations non signée et par complément à deux sont celles utilisées par la majorité des applications qui manipulent des entiers. La situation est identique pour les nombres en virgule fixe, qui seront donc encodés soit de façon non signée, soit par complément à deux.

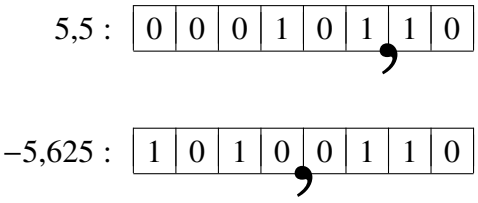
Exemple : Sur 8 bits et avec 2 chiffres après la virgule, le nombre signé $-24{,}25$ se représente 10011111. On a en effet

$$\begin{aligned} -24{,}25 &= \frac{1}{2^2}(-97) \\ &= \frac{1}{2^2}\left(-2^8 + 159\right), \text{ et} \\ [10011111]_{ns} &= 159. \end{aligned}$$

□

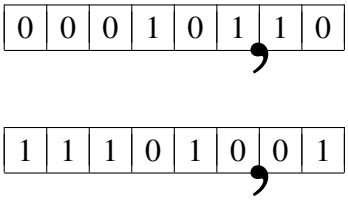
Les opérations arithmétiques sur des nombres représentés en virgule fixe se calculent de façon similaire au cas des entiers. Il y a cependant quelques précautions à observer. Lorsqu'on additionne deux nombres, ils faut veiller à ce qu'ils soient convenablement alignés : Le séparateur doit être à la même position dans les deux opérandes. Si ce n'est pas le cas, alors une de ces deux opérandes doit être décalée avant d'effectuer l'opération. Un tel décalage conduit à perdre de l'information (il s'agit d'une des sources de l'imprécision évoquée au début de la section 2.4), et sa direction doit donc être choisie de façon à n'effacer que des bits de poids faible.

Exemple : Calculons la somme sur 8 bits de 5,5 représenté avec 2 bits après la virgule, et de $-5{,}625$ représenté avec 4 bits après la virgule :

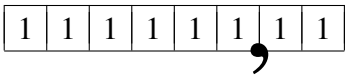


Additionner directement ces deux représentations n'aurait pas de sens, puisque le poids des bits situés à une position donnée dans chacune d'entre elles ne coïncide pas. Ces représentations doivent donc être décalées l'une par rapport à l'autre, de façon à placer le séparateur au même endroit.

Ce décalage ne peut pas conduire à supprimer des bits de poids fort. Si l'on ne dispose que de 8 bits pour effectuer l'opération, cela nous conduit à décaler la deuxième opérande de deux bits vers la droite. L'effet de cette opération est de perdre les deux bits de poids faible de cette opérande, et d'étendre de deux bits sa représentation vers la gauche, en en répétant le bit de signe. On obtient donc



L'addition de ces deux représentations fournit le résultat



qui représente $-0{,}25$. Par rapport au résultat correct, cette somme est affectée d'une erreur égale à 2^{-3}, résultant du décalage effectué sur la deuxième opérande. □

Pour calculer le produit d'un nombre en virgule fixe représenté sur n_1 bits avec k_1 chiffres après la virgule et d'un autre représenté sur n_2 bits avec k_2 chiffres après la virgule, $n_1 + n_2$ bits et $k_1 + k_2$ chiffres après la virgule sont généralement nécessaires si l'on souhaite un résultat exact. Si l'on ne dispose pas de ce nombre de bits pour effectuer l'opération, alors il faut procéder comme pour l'addition et décaler les opérandes de façon à les approximer, en veillant encore une fois à ce que ces décalages n'affectent que les bits de poids faible des représentations.

2.4.2 La représentation en virgule flottante

L'inconvénient de la représentation en virgule fixe est qu'elle impose de connaître le nombre de chiffres à prévoir avant et après le séparateur. Cette approche n'est pas adaptée à toutes les applications ; par exemple, en physique, raisonner sur la masse d'un objet peut nécessiter une quarantaine de chiffres décimaux après la virgule dans le cas de particules subatomiques, ou bien une quarantaine de chiffres avant la virgule pour des entités astronomiques. Dans un tel cas, il ne serait ni pratique ni efficace de manipuler des représentations à quatre-vingt chiffres.

La solution employée en physique et dans d'autres sciences consiste à écrire les nombres en *notation scientifique*, dans laquelle les chiffres significatifs d'une valeur sont placés dans une *mantisse*, et la grandeur de cette valeur est représentée par un *exposant* de la base. Par exemple, la masse d'un électron s'écrit

$$9{,}109 \times 10^{-31}\ \text{kg},$$

où 9,109 est la mantisse et -31 l'exposant. Ce procédé permet de représenter des nombres de grandeur très variable, en exprimant la mantisse en virgule fixe et en gardant un exposant entier. Le nombre de chiffres de la mantisse définit alors la précision avec laquelle les nombres sont représentés, et l'intervalle des valeurs possibles de l'exposant la grandeur de ceux-ci.

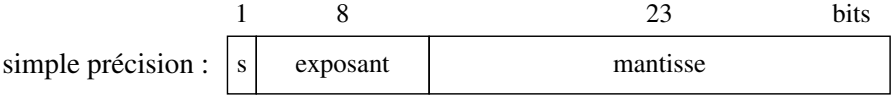
La notation scientifique, aussi appelée représentation en *virgule flottante (floating point)*, peut naturellement être utilisée dans n'importe quelle base. En binaire, un nombre r s'exprime sous la forme

$$r = m \times 2^e,$$

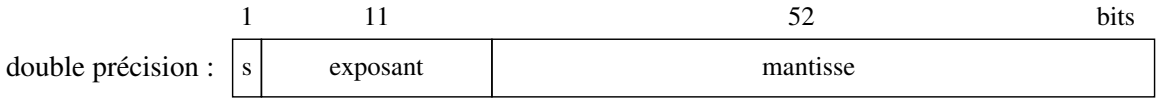
où m est la mantisse et e l'exposant. Pour représenter concrètement des réels, on pourrait donc directement exploiter les résultats des sections 2.3.3 et 2.4.1, en représentant la mantisse en virgule fixe à l'aide d'un nombre de bits avant et après la virgule déterminé, et en encodant l'exposant par complément à deux. La solution actuellement employée dans la plupart des applications est cependant légèrement différente, pour des raisons qui seront expliquées plus tard. Cette solution fait l'objet d'un standard appelé [4] *IEEE 754*. Ce standard est celui qui est imposé pour représenter les nombres réels par certains langages de programmation (comme Java), et qui est implémenté (plus ou moins fidèlement) par une majorité des processeurs modernes.

Le standard IEEE 754 définit en fait un ensemble de représentations différentes. Dans ce cours, nous allons étudier deux d'entre elles appelées *simple précision* et *double précision*. Ces représentations sont basées sur les mêmes principes, mais emploient des nombres de bits différents pour représenter la mantisse et l'exposant des nombres. Elle correspondent respectivement, pour l'architecture d'un PC moderne, aux types `float` et `double` de C, C++ et Java.

La représentation des nombres en simple et en double précision prend la forme suivante :



4. Ce nom provient de l'association d'ingénieurs qui en est à l'origine, l'*Institute of Electrical and Electronics Engineers*.



On a donc un total de 32 bits pour la simple précision et 64 pour la double. Le premier de ces bits (noté “s” sur les figures) représente le signe du nombre. Comme dans le cas des représentations entières, celui-ci est égal à 0 pour les nombres positifs et à 1 pour les nombres négatifs.

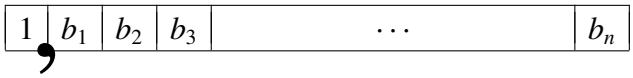
L’exposant est représenté par les 8 bits suivants en simple précision, et par 11 bits en double précision. La méthode employée consiste à encoder un exposant e comme la représentation entière non signée du nombre $e + 127$ pour la simple précision, et de $e + 1023$ pour la double précision. La valeur des exposants représentables se situe donc dans l’intervalle $[-127, \ldots, 128]$ pour la simple précision, et $[-1023, \ldots, 1024]$ pour la double précision.

La mantisse occupe les bits restants de la représentation, au nombre de 23 pour la simple précision et de 52 pour la double. Le procédé d’encodage utilisé dépend de la valeur de l’exposant. Le premier cas est celui d’un exposant qui n’est pas égal à une valeur extrême, c’est-à-dire -127 ou 128 pour la simple précision, et -1023 ou 1024 pour la double précision. Dans cette situation, la mantisse est dite *normalisée*.

Une mantisse normalisée est encodée de la façon suivante. Soit $b_1 b_2 \ldots b_n$ la représentation d’une telle mantisse, avec $n = 23$ pour la simple précision et $n = 52$ pour la double. Remarquons que, contrairement à l’encodage des entiers, les bits sont ici numérotés de gauche à droite. La valeur absolue de la mantisse représentée par cette suite de bits est égale à

$$|m| = 1 + \sum_{i=1}^{n} 2^{-i} b_i. \tag{2.6}$$

Intuitivement, il s’agit d’une représentation en virgule fixe un peu particulière : Elle comprend les n bits $b_1, b_2, \ldots, b_n$ après le séparateur, mais aussi un bit implicite systématiquement égal à 1 avant celui-ci. Cette situation est illustrée ci-dessous :



De l’expression (2.6), on déduit que la valeur absolue de m appartient à l’intervalle

$$[1,\ 2 - \frac{1}{2^n}].$$

On a donc

$$1 \leq |m| < 2$$

pour toute mantisse normalisée m.

Exemple : Calculons la représentation en simple précision du nombre $-7{,}5$:

1. Ce nombre est négatif, donc le bit de signe est égal à $\boxed{1}$.

2. Afin d'obtenir une mantisse normalisée, qui doit appartenir à l'intervalle $[1, 2)$, il faut choisir un exposant égal à 2. On a en effet

$$7{,}5 = 1{,}875 \times 2^2,$$

   avec $1 \leq 1{,}875 < 2$. L'exposant est alors encodé par la représentation non signée de $127 + 2 = 129$ sur 8 bits, c'est-à-dire $\boxed{10000001}$.

3. La valeur absolue de la mantisse se décompose en

$$1{,}875 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}.$$

   Sa représentation forme donc la suite de bits $\boxed{11100000000000000000000}$.

4. En assemblant les trois fragments obtenus, on obtient donc finalement

$$\boxed{11000000111100000000000000000000}.$$

□

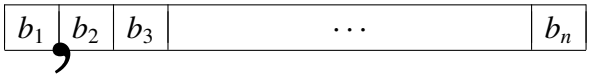
L'inconvénient des mantisses normalisées est qu'elles ne permettent pas de représenter des très petites valeurs. En simple précision, le plus petit nombre strictement positif possédant une mantisse normalisée est ainsi égal à 2^{-126}.

Pour pouvoir représenter des nombres plus petits, il est nécessaire de se débarrasser du bit systématiquement égal à 1 présent dans les mantisses normalisées. Ce sera le cas lorsque l'exposant possède la plus petite valeur possible, c'est à dire -127 pour la simple précision et -1023 pour la double. La mantisse est alors dite *dénormalisée*.

L'encodage $b_1 b_2 \ldots b_n$ d'une mantisse dénormalisée m est tel que

$$|m| = \sum_{i=1}^{n} 2^{-i+1}\, b_i. \tag{2.7}$$

En d'autres termes, le séparateur est maintenant placé entre les deux premiers bits de la représentation, et il n'y a plus de préfixe implicite :



Cette solution permet d'encoder des mantisses de petite valeur, en commençant leur représentation par un certain nombre de bits égaux à zéro, au prix d'une perte de chiffres significatifs. La valeur absolue des mantisses représentables de cette façon forme l'intervalle

$$[0,\ 2 - \frac{1}{2^{n-1}}].$$

On a donc

$$0 \leq |m| < 2$$

pour toute mantisse dénormalisée m.

Exemple : Calculons la représentation en simple précision du nombre 2^{-140} :

1. Ce nombre est positif, donc le bit de signe est égal à $\boxed{0}$.

2. Il n'est pas possible de choisir un exposant conduisant à une mantisse normalisée. En effet, pour $e \geq -126$, on a
$$\frac{2^{-140}}{2^e} \leq 2^{-14} < 1.$$
On fixe donc $e = -127$, ce qui conduit à représenter la mantisse de façon dénormalisée. Cet exposant est encodé par la représentation non signée de $-127 + 127 = 0$ sur 8 bits, c'est-à-dire $\boxed{00000000}$.

3. La mantisse m vaut alors
$$m = \frac{2^{-140}}{2^{-127}} = 2^{-13}$$
et est donc encodée par la suite de bits $\boxed{00000000000001000000000}$, où seul le quatorzième bit est non nul. On voit clairement dans cet exemple que la précision avec laquelle la mantisse est encodée est réduite, puisque seuls dix bits significatifs sont présents.

4. En assemblant les trois fragments obtenus, on obtient finalement
$$\boxed{00000000000000000000001000000000}.$$

□

Un avantage supplémentaire de l'encodage dénormalisé de la mantisse est qu'il permet de représenter le nombre zéro. Dans la norme IEEE 754, la représentation de ce nombre est composée

— d'un bit de signe quelconque, permettant donc de distinguer un *zéro positif* d'un *zéro négatif*. Il s'agit d'une propriété utile dans le cas de nombres réels, car elle permet de ne pas perdre le signe de quantités infinitésimales lorsqu'elles doivent être arrondies à zéro par manque de précision.

— d'un exposant égal à la plus petite valeur admise, c'est-à-dire -127 pour la simple précision et -1023 pour la double. Cet exposant est alors représenté par une suite de bits égaux à 0.

— d'une mantisse dénormalisée nulle, elle aussi représentée par une suite de bits égaux à zéro.

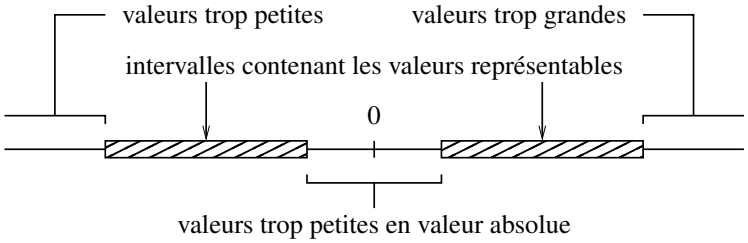
Nous sommes à présent en mesure d'expliquer pourquoi le standard IEEE 754 n'encode pas l'exposant des nombres à l'aide de la représentation par complément à deux. Le procédé utilisé présente l'avantage qu'une suite de bits égaux à 0 fournit une représentation valable du nombre zéro. Cette propriété facilite l'initialisation de structures de données contenant des champs réels, ceux-ci devenant égaux au zéro positif lorsque tous les bits de la structure sont mis à zéro.

Il reste à considérer le cas d'un exposant prenant la plus grande valeur possible, c'est-à-dire 128 pour la simple précision et 1024 pour la double. Cette situation correspond à un exposant encodé par une suite de bits tous égaux à 1. Elle est réservée pour la représentation de valeurs spéciales, indiquant que le résultat d'une opération ne correspond pas à un nombre réel :

— Si tous les bits correspondants à la mantisse sont nuls, alors la représentation correspond à un dépassement arithmétique vers le haut (si le bit de signe vaut 0) ou vers le bas (s'il est égal à 1). Par abus de langage, ces valeurs sont souvent appelées *infini positif* ($+\infty$) et *infini négatif* ($-\infty$). Elles sont produites lorsque le résultat d'une opération est trop grand en valeur absolue pour pouvoir être représenté.

— Si au moins un bit associé à la mantisse est égal à 1, alors la représentation correspond à une *valeur indéfinie*, notée *NaN (Not a Number)*. Cette valeur est produite lorsque le résultat d'une opération n'est pas défini, comme c'est le cas par exemple pour une division de 0 par 0 ou pour le calcul de la racine carrée réelle d'un nombre négatif.

Nous allons maintenant chercher à déterminer quels sont les nombres représentables en simple et en double précision. Ce problème est plus difficile que pour les nombres entiers, comme le montre cette illustration :



Sur la droite des réels, certains nombres sont trop grands en valeur absolue pour pouvoir être représentés ; ils sont remplacés par un infini positif ou négatif. De même, les nombres trop petits en valeur absolue sont arrondis vers un zéro positif ou négatif. Entre ces deux extrêmes se trouvent deux intervalles contenant les nombres positifs et négatifs représentables. Il faut bien sûr garder à l'esprit que dans ces intervalles, les nombres sont représentés avec une précision limitée. Les opérations effectuées sur les nombres réels visent en général à produire le nombre représentable le plus proche de la valeur exacte.

Le plus grand réel représentable est obtenu avec un exposant égal à 127 pour la simple précision et à 1023 pour la double, et une mantisse respectivement égale à $2 - 2^{-23}$ et $2 - 2^{-52}$. Ce

plus grand nombre est donc approximativement égal à $3{,}403 \times 10^{38}$ pour la simple précision, et à $1{,}798 \times 10^{308}$ pour la double précision.

Le plus petit nombre positif pouvant être distingué de zéro est représenté par un exposant égal à -127 pour la simple précision et à -1023 pour la double, avec une mantisse dénormalisée dont seul le dernier bit est non nul, c'est à dire égale respectivement à 2^{-22} et 2^{-51}. Ce nombre est donc égal à $2^{-149} \approx 1{,}401 \times 10^{-45}$ pour la simple précision, et à $2^{-1074} \approx 4{,}941 \times 10^{-324}$ pour la double précision.

Il reste à discuter des opérations arithmétiques sur les nombres réels. Ces opérations sont plus complexes à implémenter que sur les entiers. Les difficultés sont causées par la présence de deux encodages possibles des mantisses, par la nécessité d'arrondir correctement les résultats à la valeur représentable la plus proche, et par les mécanismes de détection et de traitement des valeurs spéciales. Dans ce cours, nous nous limitons à une description très sommaire de ces opérations.

Pour additionner deux nombres réels encodés dans la représentation IEEE 754, on procède schématiquement comme suit :

1. On aligne les exposants. Cette opération consiste à transformer l'exposant et la mantisse de l'opérande possédant l'exposant le plus petit, de façon à ce que les deux opérandes se retrouvent avec le même exposant.
   Formellement, si l'on cherche à additionner $m_1 \times 2^{e_1}$ et $m_2 \times 2^{e_2}$ avec $e_1 < e_2$, cette opération remplace dans la première opérande
   — l'exposant e_1 par e_2, ce qui revient à multiplier le nombre par $2^{e_2 - e_1}$.
   — la mantisse m_1 par $2^{e_1 - e_2}\, m_1$, ce qui rétablit la valeur du nombre. Cette opération revient à décaler de $e_2 - e_1$ positions vers la droite les bits encodant la mantisse.

   Cette étape peut conduire à perdre un certain nombre de bits dans l'encodage de la mantisse du premier nombre. Cette mantisse peut également devenir dénormalisée suite à cette opération.

2. On remplace chaque mantisse négative par sa représentation signée, en complémentant chaque bit de son encodage et en ajoutant 1 au résultat (cf. calcul de l'opposé d'un nombre dans la section 2.3.3).

3. On calcule la somme des mantisses en virgule fixe.

4. Si le résultat est négatif, on le complémente par la même méthode qu'à l'étape 2.

5. On normalise le résultat $m \times 2^e$, où m est la mantisse obtenue aux étapes précédentes et $e = e_2$, en ajustant si nécessaire e et m de façon à placer m dans l'intervalle prescrit, c'est-à-dire $[1, 2)$ dans le cas normalisé et $[0, 2)$ dans le cas dénormalisé.

   Cette opération peut conduire à détecter un dépassement, auquel cas son résultat sera remplacé par un infini de signe approprié.

Le produit de deux nombres $m_1 \times 2^{e_1}$ et $m_2 \times 2^{e_2}$ se calcule d'une façon similaire :

1. On détermine le signe du produit en fonction de celui des opérandes.

2. On calcule la somme $e = e_1 + e_2$ des exposants en arithmétique entière. Cette opération peut conduire à détecter un dépassement, ou à forcer un arrondi à zéro.

3. On calcule le produit $m = m_1 \times m_2$ en virgule fixe.

4. On normalise le résultat $m \times 2^e$, de la même façon qu'à la dernière étape de la procédure d'addition.

2.5 La représentation de textes

La très grande majorité des textes que nous lisons, que ce soit en ligne ou sur des supports imprimés, ont été produits par des applications informatiques. Ces applications représentent les textes en les décomposant en *caractères* qui peuvent être encodés par des combinaisons de bits.

Aux débuts de l'informatique, les procédés d'encodage employés par les différents constructeurs d'équipement étaient multiples et souvent mutuellement incompatibles. Il est néanmoins devenu rapidement indispensable de pouvoir transmettre des textes d'un système à un autre et les archiver en s'affranchissant le plus possible des contraintes liées au matériel utilisé. Plusieurs standards d'encodages de caractères ont ainsi vu le jour.

2.5.1 Le code ASCII

Le standard qui est à la base des systèmes d'encodage de texte actuellement utilisés est le code *ASCII (American Standard Code for Information Interchange)*. Celui-ci représente chaque caractère à l'aide d'une suite de 7 bits. Si on interprète ceux-ci comme formant la représentation binaire non signée d'un entier, chaque caractère correspond donc à un nombre situé dans l'intervalle [0, 127], c'est-à-dire [0, 0x7F] en hexadécimal.

Le jeu de caractères défini par le code ASCII est organisé de la façon suivante :

— Le premier groupe de 32 caractères (de 0x00 à 0x1F) contient des caractères de contrôle. Ceux-ci ne définissent pas des symboles destinés à être affichés ou imprimés, mais servent à contrôler certaines opérations de l'ordinateur ou de ses périphériques. Par exemple, le caractère codé 0x0A définit un saut de ligne, et le caractère 0x0D un retour du curseur à la marge gauche de la page ou de l'écran. Il est à noter que l'interprétation des caractères de contrôle n'est pas complètement standardisée et peut donc varier d'un système à un autre.

— Le deuxième groupe de 32 caractères (de 0x20 à 0x3F) contient des symboles mathématiques et de ponctuation (dont le symbole d'espacement, ou *blanc*, auquel on attribue le code 0x20) ainsi que les chiffres, codés de 0x30 à 0x39. Remarquons que la valeur d'un chiffre est égale aux 4 bits de poids faible de son encodage. En d'autres termes, pour tout $0 \leq k \leq 9$, le code 0x3k encode le chiffre k.

— Le troisième groupe de 32 caractères (de 0x40 à 0x5F) contient les lettres majuscules (de 0x41 à 0x5A) et quelques symboles spéciaux. Les lettres sont classées par ordre alphabétique, et possèdent des codes consécutifs. Ces deux propriétés permettent de comparer et de classer des caractères à l'aide d'opérations arithmétiques simples.

— Le dernier groupe de 32 caractères (de 0x60 à 0x7F) contient les lettres minuscules (de 0x61 à 0x7A), un caractère de contrôle (0x7F) et des symboles spéciaux. Notons que la différence de code entre une lettre minuscule et la lettre majuscule correspondante est toujours égale à 0x20. En d'autres termes, les deux encodages d'une même lettre partagent les mêmes 5 bits de poids faible.

Une table reprenant les 95 caractères imprimables définis par le code ASCII est donnée ci-dessous.

20		30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	’	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

2.5.2 Le standard ISO 8859-1

La plupart des ordinateurs modernes manipulent les données par groupes de 8 bits. Puisque le code ASCII requiert 7 bits par symbole, il est commode de représenter chaque caractère à l'aide d'un octet. Le bit supplémentaire contenu dans cet octet permet alors d'encoder 128 symboles de plus. Ceux-ci peuvent par exemple inclure les caractères accentués indispensables à plusieurs langues comme le français ou l'espagnol.

L'organisme de standardisation *ISO* a ainsi défini une quinzaine d'extensions du code ASCII regroupées sous le nom générique *ISO 8859*, couvrant notamment le français, les langues scandinaves, l'arabe, le grec, l'hébreu, le russe ... Ces standards sont tous compatibles avec le code ASCII dont ils empruntent les caractères de code inférieur à 0x80.

Le plus connu de ces standards, appelé *ISO 8859-1* ou *ISO latin1*, contient l'ensemble des caractères les plus utilisé par les langues européennes. Bien que tous les symboles n'y figurent pas [5], il s'agit d'un bon compromis lorsqu'il s'agit de représenter des textes rédigés en français à l'aide d'un seul octet par caractère.

2.5.3 Unicode

L'écriture de certaines langues nécessite plus de 256 symboles. Par exemple, les textes rédigée en chinois, en japonais ou en coréen sont composés de *logogrammes*, au nombre de plusieurs milliers ou même dizaines de milliers. Les applications informatiques ayant de plus en plus tendance à s'internationaliser, il est devenu indispensable de disposer d'un standard d'encodage compatible avec ces systèmes d'écriture.

Le standard *Unicode* a été créé dans le but d'unifier l'encodage de l'ensemble des symboles amenés à être traités par des applications informatiques. La version 12.1.0 de ce standard définit actuellement 137929 caractères couvrant la plupart des systèmes d'écriture modernes et historiques. Cet ensemble incorpore le jeu de symboles de plusieurs autres standards ; en particulier, les 256 premiers caractères définis par Unicode sont identiques à ceux du codage ISO 8859-1.

En Unicode, les caractères sont codés par un nombre entier dans l'intervalle [0, 0x10FFFF]. Tous les codes ne sont pas attribués ; un certain nombre d'entre eux sont réservés pour des extensions futures, ou pour un usage privé par certaines applications. Le standard définit également un certain nombre de caractères de contrôle. Le symbole possédant le code k est souvent noté U+k, où le nombre k est écrit en hexadécimal. Par exemple, le symbole "€" correspond à U+20AC.

Représenter en toute généralité un texte encodé en Unicode nécessite 21 bits par caractère. En pratique, comme la plupart des ordinateurs modernes sont conçus pour manipuler efficacement des groupes de 8, 16, 32 ou 64 bits, on est amené à représenter chaque caractère à l'aide de 3 ou 4 octets. Pour l'écriture de textes en français ou en anglais, ou encore de programmes informatiques, cette solution est donc 3 à 4 fois plus coûteuse en espace que le codage en ISO 8859-1.

Pour pallier cet inconvénient, plusieurs procédés de compression de textes codés en Unicode ont été développés. Le plus utilisé d'entre eux, *UTF-8*, fonctionne de la façon suivante. Pour représenter un caractère U+k, on détermine à quel intervalle k appartient. Il y a quatre possibilités :

— *Si* $k \in [0, 0x7F]$: Le caractère est représenté par l'octet

$$\boxed{0 b_6 b_5 \dots b_0},$$

5. Par exemple, le caractère "œ" est manquant.

où $b_6 b_5 \dots b_0$ est l'encodage binaire non signé de k.

— *Si* $k \in$ [0x80, 0x7FF] : Le caractère est représenté par les deux octets

$$\boxed{110 b_{10} b_9 \dots b_6} \boxed{10 b_5 b_4 \dots b_0},$$

où $b_{10} b_9 \dots b_0$ est l'encodage binaire non signé de k.

— *Si* $k \in$ [0x800, 0xFFFF] : Le caractère est représenté par les trois octets

$$\boxed{1110 b_{15} b_{14} b_{13} b_{12}} \boxed{10 b_{11} b_{10} \dots b_6} \boxed{10 b_5 b_4 \dots b_0},$$

où $b_{15} b_{14} \dots b_0$ est l'encodage binaire non signé de k.

— *Si* $k \in$ [0x10000, 0x10FFFF] : Le caractère est représenté par les quatre octets

$$\boxed{11110 b_{20} b_{19} b_{18}} \boxed{10 b_{17} b_{16} \dots b_{12}} \boxed{10 b_{11} b_{10} \dots b_6} \boxed{10 b_5 b_4 \dots b_0},$$

où $b_{20} b_{19} \dots b_0$ est l'encodage binaire non signé de k.

Exemple : Pour le caractère U+20AC ("€"), on a $k = 0x20AC \in [0x800, 0xFFFF]$. L'encodage binaire de 0x20AC est égal à

$$0010\,0000\,1010\,1100.$$

Ce symbole est donc représenté par les trois octets

$$\boxed{1110\,0010} \boxed{1000\,0010} \boxed{1010\,1100},$$

c'est-à-dire $\boxed{\text{E2}} \boxed{\text{82}} \boxed{\text{AC}}$ en hexadécimal. □

Cette méthode de compression fait l'hypothèse que les caractères les plus fréquents dans un texte sont ceux dont le code est petit. Dans le cas d'un texte composé uniquement de symboles appartenant à l'intervalle [U+0, U+7F], elle présente l'avantage d'en fournir une représentation identique à son encodage ISO 8859-1. La manipulation de chaînes de caractères Unicode comprimés en UTF-8 n'est cependant pas simple ; par exemple, le fait que les symboles sont de longueur variable complique le calcul de la longueur d'une chaîne. Il n'est également pas permis de tronquer une suite d'octets représentant une chaîne de caractères à n'importe quelle position, car cela pourrait produire des encodages incomplets. Le standard Unicode comprimé en UTF-8 est le procédé le plus répandu pour l'encodage du contenu du *World-Wide Web*.

2.6 Exercices résolus

1. Représenter le nombre -100 dans le format IEEE 754 en simple précision.

   Solution :

   — Ce nombre est négatif, donc le bit de signe est égal à $\boxed{1}$.

   — Pour obtenir une mantisse normalisée, il faut choisir un exposant égal à 6. Cet exposant est encodé par la représentation binaire non signée sur 8 bits de $6 + 127 = 133$, c'est-à-dire $\boxed{10000101}$.

   — La valeur absolue de la mantisse est égale à $1{,}5625 = 1 + 2^{-1} + 2^{-4}$. Seuls le premier et le quatrième bit de sa représentation sont donc égaux à un :

   $$\boxed{10010000000000000000000}$$

   — En résumé, on obtient donc

   $$\boxed{11000010110010000000000000000000}.$$

2. Représenter le nombre -54 dans les formats suivants :

   (a) Valeur signée sur 8 bits.

   (b) Complément à un sur 8 bits.

   (c) Complément à deux sur 8 bits.

   (d) Représentation en double précision de la norme IEEE 754.

   Solution :

   (a) Ce nombre est négatif, donc le bit de signe est égal à 1. La représentation binaire non signée de sa valeur absolue (54) sur 7 bits est égale à 0110110. La représentation demandée vaut donc $\boxed{10110110}$.

   (b) La représentation binaire non signée de 54 sur 8 bits vaut 00110110. En complémentant chaque bit, on obtient $\boxed{11001001}$.

   (c) Le nombre étant négatif, sa représentation par complément à deux est identique à la représentation par complément à un de $-54 + 1 = -53$.

   La représentation binaire non signée de 53 sur 8 bits vaut 00110101. En inversant chaque bit, on obtient $\boxed{11001010}$.

   (d) — Ce nombre est négatif, donc le bit de signe est égal à $\boxed{1}$.

   — Pour obtenir une mantisse normalisée, il faut choisir un exposant égal à 5. Cet exposant est encodé par la représentation binaire non signée sur 11 bits de $5 + 1023 = 1028$, c'est-à-dire $\boxed{10000000100}$.

— La mantisse est égale à $1{,}6875 = 1 + 2^{-1} + 2^{-3} + 2^{-4}$. Seuls le premier, le troisième et le quatrième bit de sa représentation sont donc égaux à 1 :

$$\boxed{10110000000000000000000000000000000000000000000000000000}$$

— La représentation demandée vaut donc

$$\boxed{1100000001001011000000000000000000000000000000000000000000000000}.$$

3. Quels sont les nombres représentés par une suite de 64 bits égaux à 1 dans les représentations

   (a) entière non signée ?

   (b) par valeur signée ?

   (c) par complément à un ?

   (d) par complément à deux ?

   (e) IEEE 754 ?

   Solution :

   (a) $\sum_{i=0}^{63} 2^i \, 1 = 2^{64} - 1.$

   (b) Le bit de signe étant égal à 1, le nombre est négatif (ou nul). Sa valeur absolue vaut $\sum_{i=0}^{62} 2^i \, 1 = 2^{63} - 1$. Le nombre demandé est donc $1 - 2^{63}$.

   (c) Le bit de signe étant égal à 1, le nombre est négatif (ou nul). Sa valeur absolue vaut $\sum_{i=0}^{62} 2^i \, (1 - 1) = 0$. Le nombre demandé est donc 0.

   (d) Le bit de signe étant égal à 1, le nombre est négatif. Appelons v ce nombre. Les 64 bits constituent le complément à un du nombre $v + 1$, qui vaut 0 (cf. sous-question précédente). Le nombre v recherché vaut donc -1.

   (e) Le nombre de bits étant égal à 64, il s'agit d'une représentation en double précision. Tous les bits de l'exposant sont égaux à un, et au moins un bit de l'encodage de la mantisse est non nul. Il s'agit donc de la représentation d'une valeur indéfinie (*NaN*).

4. Quels sont les nombres dont les représentations binaires sur n bits par valeur signée et par complément à deux sont identiques ?

   Solution :

   Tout d'abord, on sait que les nombres positifs ou nuls représentables par les deux procédés sont encodés de la même façon. Il s'agit des nombres appartenant à l'intervalle $[0, \ldots, 2^{n-1} - 1]$.

Recherchons maintenant s'il existe des nombres négatifs pour lesquels les deux représentations coïncident. Soient $b_{n-2}b_{n-3}\cdots b_1b_0$ les $n-1$ bits de poids faible de la représentation d'un tel nombre (nous ne considérons pas le bit de signe, que nous savons être égal à 1).

Si le nombre considéré est représenté par valeur signée, il est égal à

$$-\sum_{i=0}^{n-2} 2^i\, b_i.$$

S'il est représenté par complément à deux, il vaut

$$\begin{aligned} &-2^n + 2^{n-1} + \sum_{i=0}^{n-2} 2^i\, b_i \\ =\ &-2^{n-1} + \sum_{i=0}^{n-2} 2^i\, b_i. \end{aligned}$$

En égalant les deux expressions, on obtient

$$\sum_{i=0}^{n-2} 2^i\, b_i = 2^{n-2}.$$

Le seul nombre négatif représenté de façon identique par les deux procédés est donc

$$-\sum_{i=0}^{n-2} 2^i\, b_i = -2^{n-2},$$

à condition d'avoir $n \geq 2$.

5. Considérons un procédé analogue à la norme IEEE 754, mais attribuant seulement 5 bits à l'exposant et 10 bits à la mantisse. Quel est l'intervalle contenant tous les nombres représentables par ce procédé ?

   Solution :

   La plus grande valeur absolue représentable correspond à un exposant égal à 15 et à une mantisse (normalisée) égale à $2 - 2^{-10}$. Cette valeur vaut donc $2^{15}(2 - 2^{-10}) = 65504$. L'intervalle demandé est donc

   $$[-65504, \ldots, 65504].$$

6. Imaginer et décrire un algorithme général permettant de passer de la représentation d'un réel en double précision (format IEEE 754) à la représentation de ce réel en simple précision (format IEEE 754).

   Solution :

   Notons $b_0b_1\cdots b_{62}b_{63}$ la suite de bits fournie à l'algorithme de conversion.

(a) On isole le bit de signe b_0, qui sera également celui du résultat.

(b) On calcule la valeur de l'exposant e, en retranchant 1023 de la valeur du nombre non signé représenté par la suite de bits $b_1 b_2 \cdots b_{10}\ b_{11}$.

(c) — Si $e < -149$, alors la valeur absolue du nombre est trop petite pour être distinguée de zéro par une représentation en simple précision. On fait donc suivre le bit de signe b_0 de 31 bits égaux à 0.

— Si $-149 \leq e \leq -127$, alors le nombre peut être représenté en simple précision grâce à une mantisse dénormalisée. La représentation de l'exposant est dans ce cas composée de 8 bits égaux à zéro.

La mantisse du nombre fourni en entrée vaut

$$1 + \sum_{i=1}^{52} 2^{-i}\, b_{(11+i)}.$$

La représentation de la mantisse du résultat commence donc par $(-127 - e)$ bits égaux à zéro, suivis d'un bit égal à 1. Le reste de la représentation s'obtient en recopiant le nombre approprié de bits depuis le début de l'encodage de la mantisse originale $b_{12} b_{13} \cdots$. Les bits qui ne sont pas recopiés sont perdus.

— Si $-126 \leq e \leq 127$, alors le nombre peut être représenté à l'aide d'une mantisse normalisée. On représente donc l'exposant par les 8 bits de la représentation non signée de $e + 127$. La représentation de la mantisse est formée des 23 bits $b_{12} b_{13} \cdots b_{34}$. Les bits suivants sont perdus.

— Si $128 \leq e \leq 1023$, alors la valeur absolue du nombre est trop grande pour pouvoir être représentée. On génère donc un infini de signe correspondant à b_0.

— Si $e = 1024$, alors il s'agit d'une valeur spéciale. On génère donc une valeur spéciale identique en simple précision.

Note : Le procédé d'arrondi mis en œuvre consiste à ne pas tenir compte des bits ne pouvant pas être représentés. Cette méthode ne produit pas toujours la représentation la plus proche du nombre initial. Ce problème se manifeste lorsque le premier bit perdu de la mantisse est égal à 1. Il peut alors être corrigé en ajoutant à la mantisse du nombre une valeur égale au poids de son bit de poids faible. □

2.7 Exercices supplémentaires

1. Effectuer l'opération $26 - 37$ dans les représentations binaires par valeur signée, par complément à un et par complément à deux (à chaque fois sur 8 bits).

2. Calculer le produit -17×-23 à partir des représentations binaires par complément à deux de ces nombres.

3. Quel est le nombre représenté par la suite de bits

10000000000000000000000000000001

   (il y a 30 chiffres 0) dans les représentations

   (a) entière non signée ?
   (b) par valeur signée ?
   (c) par complément à un ?
   (d) par complément à deux ?
   (e) non signée en virgule fixe, avec autant de bits avant et après la virgule ?
   (f) IEEE 754 ?

4. Ajouter 1 au nombre représenté par la suite de bits

11000011110000111100001111000011

   dans les représentations

   (a) entière non signée,
   (b) par valeur signée,
   (c) par complément à un,
   (d) par complément à deux,
   (e) par complément à deux en virgule fixe, avec 8 bits après la virgule,
   (f) IEEE 754,

   et exprimer le résultat dans la même représentation.

5. Considérons les deux nombres représentés dans le format IEEE 754 par les suites de bits

0	10000010	10110100000000000000000

   et

1	10000000	10010000000000000000000

   .

   Calculer la somme de ces deux nombres dans le même format, en détaillant votre raisonnement.

6. Donner une représentation des nombres suivants dans le format IEEE 754 simple précision. (S'il existe plusieurs représentations, ou aucune, le signaler.)

   (a) $\frac{23}{8}$
   (b) 2^{-133}
   (c) -10^{-50}
   (d) 0

(e) *NaN*

7. Le nombre suivant est représenté en notation binaire non signée :

$$\boxed{10000000000000000000000000000001}$$

(Il y a 30 bits égaux à 0 entre les deux égaux à 1.)

(a) Encoder ce nombre dans le format IEEE 754 simple précision, le plus précisément possible.

(b) Après cette conversion, quelle est la valeur exacte du nombre réellement représenté ?

(c) Si l'on convertissait à nouveau ce nombre vers la notation binaire non signée, quelle serait la différence entre le résultat obtenu et le nombre original ?

8. Un format imaginaire semblable à la norme IEEE 754 attribue 7 bits à l'exposant et 16 bits à la mantisse. Quel serait le plus petit nombre strictement positif représentable dans ce format ?

9. Décrire un algorithme général permettant de passer de la représentation d'un réel en simple précision (format IEEE 754) à celle de sa moitié.

10. Décrire un algorithme permettant de décoder une chaîne de caractères comprimée en UTF-8, en veillant à signaler les caractères qui ne sont pas correctement encodés.

11. *(Examen de première session, 2018)*

(a) Quels sont les plus petits et les plus grands nombres représentables à l'aide des encodages

i. entier non signé sur 16 bits ?

ii. entier par complément à deux sur 16 bits ?

iii. en virgule fixe par complément à deux, avec 8 bits avant et 8 bits après la virgule ?

iv. par le procédé IEEE 754 en double précision ?

(b) Calculer le produit $-1 \times (-1)$ à l'aide de la représentation par complément à deux des entiers sur 4 bits.

(c) Quel est le nombre représenté par la suite de bits

$$\boxed{11000000000000000000000000000000}$$

(il y a 2 bits égaux à 1 suivis de 30 bits égaux à 0), par le procédé IEEE 754 ?

12. *(Examen de seconde session, 2018)*

(a) Quelle est la représentation sur n bits (avec $n \geq 2$) du nombre -1 par les procédés

i. par complément à un ?

ii. par complément à deux ?

iii. en virgule fixe par complément à deux avec 1 bit après la virgule ?

(b) Calculer la somme $-2 + (-1)$ à l'aide de la représentation par complément à un des entiers sur 4 bits.

13. *(Examen de première session, 2019)*

(a) Quel est le nombre représenté par la constante `0x80000001` dans les représentations
— par valeur signée ?
— par complément à un ?
— par complément à deux ?
— IEEE754 simple précision ?

(b) Quels sont les nombres dont les représentations sur n bits par complément à un et par complément à deux sont égales ?

(c) En utilisant la notation hexadécimale, donnez une représentation d'une valeur indéterminée (*Not A Number*) dans le système IEEE754 en double précision.

14. *(Examen de seconde session, 2019)*

(a) Représenter le nombre -16 sur 8 bits par valeur signée et par complément à deux.

(b) Représenter le nombre $-1{,}25$ en virgule fixe par complément à deux, avec 4 chiffres avant la virgule et 4 chiffres après la virgule.

(c) Représenter le nombre 2^{-128} dans le format IEEE754 en simple précision, en exprimant le résultat en hexadécimal.

(d) Exprimer la valeur du nombre entier dont la représentation par complément à deux forme la suite de bits $b_3 b_2 b_1 b_0$.

(e) Effectuer l'addition $(-2) + (-5)$ en représentant les nombres par complément à un sur 4 bits.

Chapitre 3

La structure d'un ordinateur

3.1 Introduction

Comme nous l'avons vu au chapitre 1, un ordinateur moderne est essentiellement un circuit électronique qui manipule de l'information encodée par des signaux binaires. Les capacités de traitement de l'ordinateur ne sont bien sûr pas limitées à des données booléennes ; grâce notamment aux mécanismes introduits au chapitre 2, les signaux binaires peuvent être employés pour représenter et traiter des données plus complexes, notamment des nombres et des textes.

Dans ce chapitre, nous allons étudier la structure et les principes de fonctionnement du circuit d'un ordinateur. La forme précise de ce circuit dépend largement de l'application, selon qu'il s'agisse d'un ordinateur à usage général ou bien d'un système informatique enfoui ou embarqué, mais ce circuit inclut toujours

— un ou plusieurs *processeurs (Central Processing Unit, CPU)*, dont le rôle est d'exécuter les programmes.

— de la *mémoire*, chargée de retenir les programmes et les informations traitées par ceux-ci.

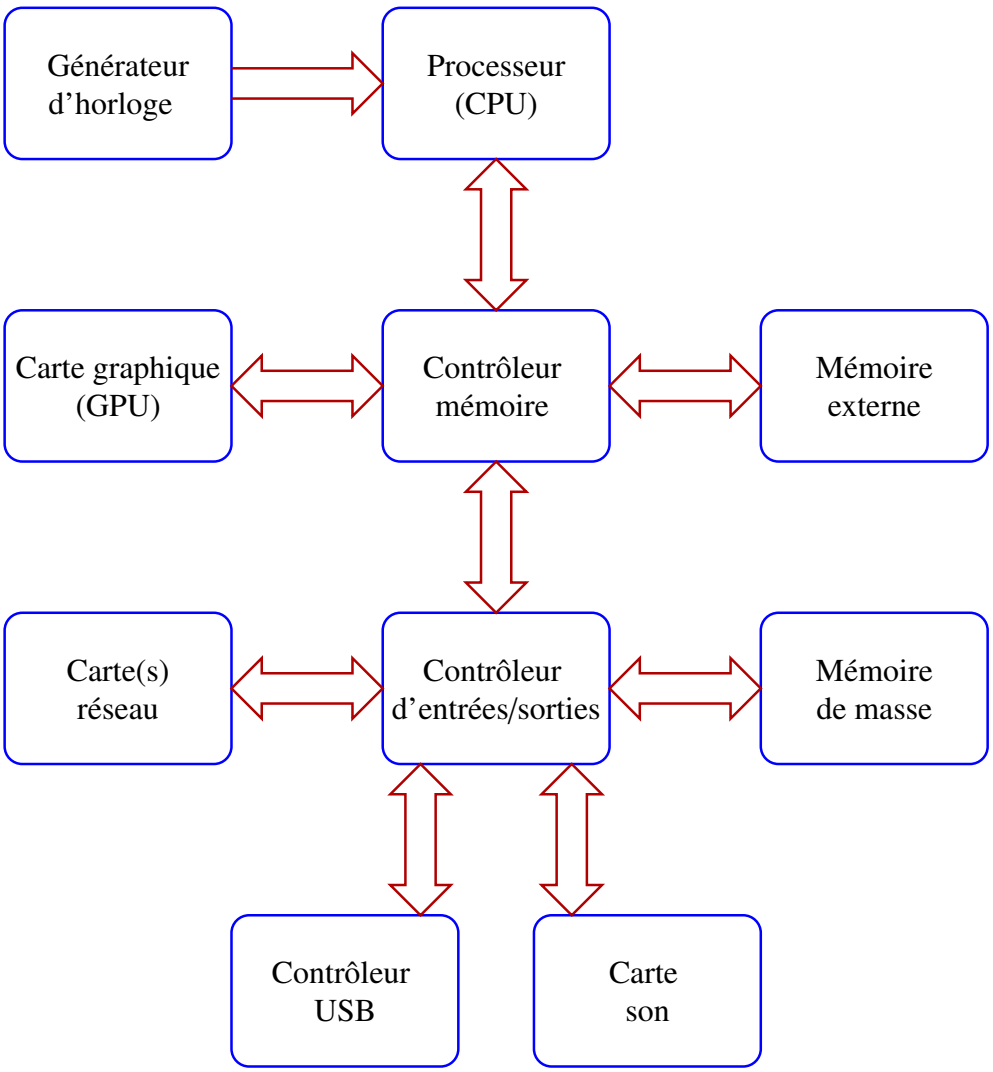
— un *générateur d'horloge*, qui produit un signal périodique indiquant au processeur à quel rythme il doit exécuter les instructions composant les programmes.

— un ensemble de *périphériques*, comme par exemple une *carte graphique (Graphics Processing Unit, GPU)* pour l'affichage sur un écran, une *carte réseau* pour échanger des informations avec d'autres ordinateurs, une carte son, des interfaces vers un clavier, une souris, ...

— des *contrôleurs* chargés de gérer les flux de données entre les différents composants.

— des *bus de communication*, qui constituent les canaux d'échange de données entre les composants.

Le diagramme suivant montre un exemple d'organisation possible de ces composants.



Les blocs qui apparaissent sur cette figure sont des unités fonctionnelles qui peuvent être implémentées par des composants électroniques distincts, ou bien être regroupées. Par exemple, beaucoup de processeurs modernes incluent un contrôleur mémoire, et même parfois une carte graphique. Les *microcontrôleurs*, qui sont des processeurs simples adaptés aux applications enfouies et embarquées, peuvent englober l'ensemble des blocs de ce schéma au sein d'un seul composant.

Nous allons à présent étudier plus en détail deux des composants de l'ordinateur : la mémoire (à la section 3.2), et le processeur (à la section 3.3).

3.2 La mémoire

En informatique, le terme *mémoire* est très général et couvre l'ensemble des composants capables de retenir de l'information. Ces composants diffèrent notamment par

— la nature des données qu'ils mémorisent,

— leurs modalités d'utilisation, par exemple, si les données mémorisées peuvent ou non être modifiées, et si elles peuvent l'être ou non de façon illimitée,

— leur capacité (correspondant à la quantité d'information qu'ils contiennent),

— la vitesse d'accès à leur contenu, caractérisée par leur *taux de transfert* (qui est la quantité d'information qu'ils peuvent lire ou écrire par seconde) et leur *latence* (correspondant au délai entre l'instant où une opération est commandée et celui où cette opération finit d'être réalisée).

Les trois sections suivantes décrivent les principaux types de mémoire que l'on rencontre dans un ordinateur.

3.2.1 La mémoire vive

La mémoire vive, ou *RAM (Random-Access Memory)* est une mémoire qui permet la consultation et la modification illimitées de n'importe quel élément de son contenu, avec un taux de transfert élevé et une latence faible. Les données mémorisées sont retenues tant que le composant reste sous tension, et sont généralement perdues lorsque l'ordinateur est éteint.

La mémoire vive se rencontre à différents endroits :

— La *mémoire externe* est formée par un ensemble de composants de mémoire vive extérieurs au processeur. Elle sert à retenir à la fois les programmes exécutés par le processeur et les données manipulées par ceux-ci.

  Pour la plupart des systèmes informatiques modernes, la capacité de cette mémoire va de quelques mégaoctets à plusieurs centaines de gigaoctets. La vitesse de lecture et d'écriture de la mémoire vive externe peut aller jusqu'à plusieurs dizaines de gigaoctets par seconde. La latence est typiquement de l'ordre d'une dizaine de nanosecondes.

— La *mémoire cache* est située à l'intérieur du processeur. Son but est d'accélérer les opérations impliquant la mémoire externe, en gérant une copie locale d'une partie de son contenu. Par exemple, lorsque le processeur doit lire une valeur depuis un emplacement de la mémoire externe, ses circuits commencent par déterminer si la mémoire cache contient déjà une copie de cette valeur. Le cas échéant, cette mémoire cache peut effectuer l'opération plus rapidement que la mémoire externe.

  Les processeurs actuels possèdent différents niveaux de mémoire cache. Le premier ni-

veau (L1) est de capacité très limitée (de quelques kilooctets à quelques dizaines de kilooctets), mais est très rapide. Les niveaux suivants (L2, L3, parfois L4) ne sont sollicités que pour les opérations qui mettent en défaut le niveau précédent. Ils sont de capacité supérieure (jusqu'à plusieurs dizaines de mégaoctets), mais plus lents. Enfin, lorsque la mémoire cache ne dispose pas de la donnée nécessaire à une opération, cette donnée est récupérée depuis la mémoire externe.

— Les *registres* sont des emplacements de mémoire situés au cœur du processeur, qui servent à retenir les valeurs manipulées par les instructions exécutées, et les résultats de ces manipulations. Il s'agit du type de mémoire vive le plus rapide. Le jeu de registres disponibles pour un modèle de processeur donné est défini par son *architecture*.

— Les *périphériques*, tels que la carte graphique, la carte réseau, le contrôleur USB, ..., sont des systèmes embarqués qui incluent également une certaine quantité de mémoire vive pour leurs opérations internes.

De nos jours, les composants de mémoire vive sont principalement construits à l'aide de deux technologies :

— La mémoire statique *(Static RAM, SRAM)* est, schématiquement, implémentée par un circuit électronique dans lequel des paires d'éléments se transmettent en boucle l'information mémorisée. Ses avantages sont d'être rapide et facile à utiliser. Il est cependant difficile et coûteux de construire des mémoires statiques de très grande taille. Les registres et la mémoire cache des processeurs sont constitués de mémoire de ce type.

— La mémoire dynamique *(Dynamic RAM, DRAM)* est organisée autour d'une matrice de condensateurs dont la charge représente l'information qu'elle contient. Son principal inconvénient est que la charge de ces condensateurs doit périodiquement être rafraîchie pour ne pas s'atténuer avec le temps, ce qui complique l'utilisation de ce type de mémoire. Son avantage est d'être plus simple et moins coûteuse à fabriquer que la mémoire statique, surtout pour des composants de grande capacité. La mémoire externe de la plupart des ordinateurs actuels est implémentée à l'aide de mémoire dynamique.

3.2.2 La mémoire morte

Par opposition à la mémoire vive, le terme *mémoire morte (Read-Only Memory, ROM)* désigne un type de mémoire dont le contenu ne peut pas être modifié durant son fonctionnement normal. Elle est utilisée pour mémoriser les données et les programmes qui ne sont pas amenés à changer lors de l'utilisation de l'ordinateur. Pour un ordinateur personnel, il s'agit principalement du logiciel responsable de démarrer l'ordinateur lorsque celui-ci est mis sous tension, ainsi que de paramètres dont la valeur doit être préservée. En revanche, dans des applications informatiques enfouies, le programme exécuté reste essentiellement identique durant toute la vie du système, et réside alors en mémoire morte.

Il existe différentes technologies de mémoire morte :

— Certains composants ne sont pas programmables, ce qui signifie que leur contenu est fixé au moment de leur fabrication. Ils sont principalement utilisés pour des applications embarquées simples déployées en un très grand nombre d'exemplaires.

— Les composants de mémoire morte *OTP (One-Time Programmable)* représentent l'information à l'aide d'un ensemble de fusibles qui peuvent être sélectivement détruits afin de programmer la valeur de chaque bit mémorisé. Cette programmation est irréversible. Après avoir été programmé, le contenu de la mémoire est préservé de façon permanente.

— Les mémoire mortes *EPROM (Erasable Programmable ROM)* et *EEPROM (Electrically Erasable PROM)* sont similaires à la technologie OTP, mais les fusibles qui représentent l'information mémorisée y sont remplacés par des composants électroniques spéciaux [1], qui sont capables de retenir une charge électrique pendant une très longue durée (plusieurs dizaines d'années). L'intérêt de ce procédé est qu'il permet de restaurer l'état d'origine des fusibles, c'est-à-dire d'effacer les données mémorisées, afin de pouvoir reprogrammer le composant. Cette opération d'effaçage s'effectue soit en exposant le composant de mémoire à une lumière ultraviolette intense (pour l'EPROM), soit par un procédé électrique (pour l'EEPROM). Dans les deux cas, il s'agit d'une opération lente, et qui ne peut être effectuée qu'un nombre limité de fois avant que le composant ne se détériore.

— La mémoire *Flash* est une forme améliorée de mémoire EEPROM, pour laquelle l'opération d'effaçage peut être appliquée à une partie seulement des données mémorisées. Elle est actuellement utilisée comme mémoire morte dans de nombreuses applications nécessitant une reprogrammation occasionnelle (par exemple, des systèmes enfouis dont le logiciel embarqué doit pouvoir être mis à jour), ainsi que comme mémoire de masse dans les ordinateurs et les dispositifs mobiles (cf. section suivante).

3.2.3 La mémoire de masse

La *mémoire de masse* sert à retenir l'ensemble des données qui doivent être préservées lorsque l'ordinateur est mis hors tension, et qui peuvent potentiellement être modifiées. Dans la plupart des ordinateurs actuels, cette mémoire est constituée

— de *disques durs*, dans lesquels l'information est stockée par l'état de la magnétisation de la surface d'un ou de plusieurs disques mis en rotation constante. Le principal avantage de cette solution est qu'elle permet de mémoriser de grandes quantités d'information (jusqu'à plusieurs téraoctets sur un seul support) à faible coût. Son inconvénient est de présenter une latence élevée : L'accès à une donnée arbitraire nécessite de déplacer mécaniquement une tête de lecture/écriture et d'attendre dans le pire des cas une rotation complète du disque, ce qui représente un délai qui peut s'élever à une dizaine de millise-

1. Il s'agit de *transistors à gâchette flottante*.

condes. Un autre défaut des disques durs est leur relative fragilité par rapport à des chocs mécaniques ou des variations de température.

— de *mémoire Flash*, qui possède une latence beaucoup plus faible que les disques durs et est insensible aux chocs mécaniques. Son principal inconvénient est son coût qui reste encore plus élevé que celui des disques durs, et le fait que le contenu de ce type de mémoire ne peut être modifié qu'un nombre limité de fois. Les composants de mémoire Flash destinés à être utilisés comme mémoire de masse incluent cependant un mécanisme d'égalisation capable de répartir les opérations de réécriture sur l'ensemble des emplacements de la mémoire, afin d'en augmenter la durée de vie.

3.2.4 L'adressage

Pour pouvoir accéder aux données stockées dans une mémoire, il est nécessaire d'en organiser le contenu. Pour la mémoire vive et la mémoire morte, on emploie un système d'*adressage* :

— Le contenu de la mémoire est organisé sous la forme d'un ensemble de *cellules*, de taille fixe.

— Chaque cellule possède une *adresse*, qui est un nombre qui l'identifie de façon unique.

— L'*espace d'adressage* est l'ensemble des adresses possibles. Certaines parties de cet espace peuvent être associées à des composants différents, ou ne correspondre à aucun composant.

Dans les systèmes informatiques actuels, la mémoire possède souvent des cellules de 8 bits. En d'autres termes, chaque cellule contient un octet. Ce n'est cependant pas toujours le cas ; par exemple, certains microcontrôleurs retiennent leurs programmes dans une mémoire dont la taille de cellule est supérieure, de façon à pouvoir placer chaque instruction dans une cellule.

Les adresses des cellules d'une mémoire prennent la forme de nombres entiers non signés, généralement représentés à l'aide d'un nombre de bits n fixé. L'espace d'adressage correspond alors à l'intervalle $[0, 2^n - 1]$. On a par exemple $n = 64$ pour la plupart des ordinateurs personnels actuels. Cela signifie que la taille de leur espace d'adressage s'élève à plus de seize millions de téraoctets. Bien sûr, la plus grande partie de cet espace n'est pas utilisée, la quantité de mémoire physiquement présente dans l'ordinateur étant très largement inférieure à cette taille.

La notion d'adresse est liée au concept de *pointeur* présent dans certains langages de programmation. Le diagramme suivant montre un exemple de contenu possible d'une mémoire. Dans cet exemple, la cellule d'adresse 0x100 contient la valeur 0x12. On dit alors que 0x100 est un pointeur vers 0x12, ou encore que 0x100 pointe vers 0x12.

0x103 :	0x78
0x102 :	0x56
0x101 :	0x34
0x100 :	0x12

Les données de taille supérieure à celle d'une cellule doivent être réparties sur plusieurs d'entre elles. Par exemple, pour représenter l'entier 0x12345678, quatre cellules de 8 bits sont nécessaires, qui accueilleront les octets 0x12, 0x34, 0x56 et 0x78 qui forment la représentation de ce nombre. Le plus simple consiste à placer ces octets dans des cellules consécutives de la mémoire. Il y a deux façons naturelles de le faire :

— Soit en énumérant les octets depuis le poids faible vers le poids fort, et en les plaçant à des adresses croissantes : L'octet de poids faible se retrouve ainsi à la plus petite adresse. Ce procédé porte le nom de représentation *petit-boutiste (little-endian)* [2].

— Soit en énumérant les octets depuis le poids fort vers le poids faible, et en les plaçant à des adresses croissantes : L'octet de poids faible se retrouve à la plus grande adresse. Ce procédé porte le nom de représentation *gros-boutiste (big-endian)*.

Exemple : Représentations du nombre 0x12345678 sur 4 cellules de 8 bits, à partir de l'adresse 0x100 :

0x103 :	0x12
0x102 :	0x34
0x101 :	0x56
0x100 :	0x78

Représentation petit-boutiste

0x103 :	0x78
0x102 :	0x56
0x101 :	0x34
0x100 :	0x12

Représentation gros-boutiste

□

Pour les applications informatiques, le choix d'une représentation petit-boutiste ou grand-boutiste n'a pas d'importance, tant qu'il est cohérent : Une valeur écrite en mémoire d'une des deux façons doit être lue de la même manière pour être correctement décodée. Les conventions employées par les principaux fabricants de processeurs diffèrent. Par exemple, les architectures x86 et x86-64 déployées dans les ordinateurs personnels modernes utilisent la représentation petit-boutiste. En revanche, l'échange de données entre deux ordinateurs connectés via un réseau

2. Ce terme est tiré des *Voyages de Gulliver* de Jonathan Swift, dans lesquels deux sectes de Lilliputiens s'affrontent autour de la question de déterminer s'il convient de manger les œufs à la coque en commençant par le petit ou par le grand bout.

suit généralement une convention gros-boutiste, de même que les processeurs installés dans certains ordinateurs *mainframe*. Certaines architectures de processeurs telles que ARM permettent au programmeur de choisir le mode de représentation qu'il ou elle préfère utiliser.

Le fait que la mémoire possède une taille de cellule de 8 bits ne signifie pas que les transferts de données vers et depuis cette mémoire s'effectuent octet par octet. Dans un souci de performance, les mémoires externes sont le plus souvent connectées à des bus capables de transmettre plusieurs octets, par exemple 8 ou 16, en une seule opération. Il faut parfois tenir compte de cette particularité lorsqu'on programme des opérations impliquant la mémoire. Certaines architectures (par exemple, MIPS) imposent en effet que les opérations de lecture ou d'écriture d'un bloc de données soient convenablement *alignées*. Cela signifie que lorsqu'une telle opération porte sur un bloc de k octets, avec k typiquement égal à 2, 4, 8 ou 16, ce bloc doit être placé à une adresse multiple de k. Dans le cas contraire, le gestionnaire de mémoire n'est pas capable d'effectuer l'opération. Pour d'autres architectures, comme x86 et x86-64, le transfert de données non alignées est possible, mais inefficace. Par exemple, lire 4 octets depuis l'adresse 0x100 (qui est un multiple de 4) prend significativement moins de temps qu'à partir de l'adresse 0x101.

Signalons enfin que les mémoires de masse utilisent également un mécanisme d'adressage, mais plus complexe que celui des mémoires vives et mortes : Leur contenu est découpé en *secteurs*, qui contiennent un nombre fixé d'octets (par exemple, 4K). Les secteurs sont identifiés par une adresse dans un espace d'adressage linéaire, ou par plusieurs paramètres qui permettent de les localiser sur le support physique de la mémoire.

3.3 Le processeur

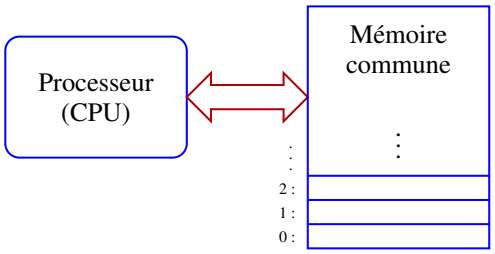
3.3.1 Introduction

Le processeur *(Central Processing Unit, CPU)* est le composant de l'ordinateur responsable de l'exécution des programmes. Pour pouvoir être traités par le processeur, ces programmes doivent être exprimés sous la forme de *code machine*, et placés en mémoire. Nous nous intéresserons plus tard à la forme prise par ce code machine ; à ce stade, on peut considérer qu'il s'agit simplement de valeurs situées en mémoire, que le processeur est capable de décoder.

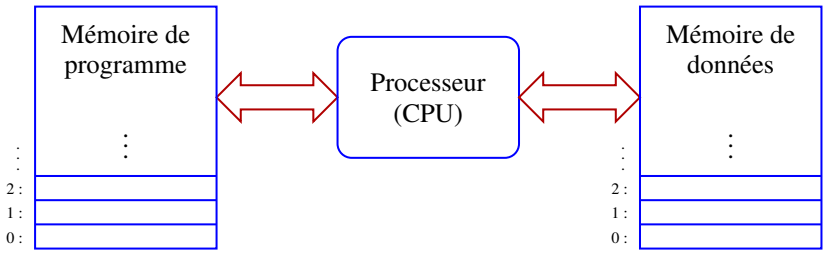
La mémoire contenant les programmes à exécuter (la *mémoire de programme*), peut partager le même espace d'adressage que celle qui contient les données à traiter et les résultats de ces traitements (la *mémoire de données*). On parle alors de modèle d'architecture *Von Neumann*. L'avantage de cette solution réside dans sa simplicité (le processeur possède une seule interface vers la mémoire externe), et le fait que le code machine qui constitue les programmes peut être manipulé de la même façon que les données. Cela permet, par exemple, d'utiliser le même mécanisme pour charger les programmes et les données depuis la mémoire de masse.

Une autre approche consiste à séparer les mémoires de programme et de données. Chacune d'entre elles possède alors son propre espace d'adressage, et une interface distincte vers le pro-

cesseur. Ce modèle d'architecture porte le nom d'*Harvard*. Son avantage est que l'accès à la mémoire de programme n'est pas ralenti par les transferts vers et depuis la mémoire de données, et vice-versa, puisque ces deux flux d'information empruntent des chemins distincts. Ce modèle permet également de doter les mémoires de programme et de données de caractéristiques différentes. Par exemple, la taille de cellule de la mémoire de programme peut être choisie de façon à encoder efficacement le code machine.



Architecture Von Neumann



Architecture Harvard

3.3.2 La structure d'un processeur

Les processeurs sont des circuits électroniques complexes [3] dont l'étude détaillée sort du cadre de ce cours. A un certain niveau d'abstraction, le circuit d'un processeur est organisé de la façon suivante :

— La *banque de registres* contient une petite quantité de mémoire vive utilisée comme espace de travail. Les registres accueillent notamment les résultats d'opérations qui doivent être traités par des instructions ultérieures.

L'ensemble des registres disponibles dans cette banque et leurs particularités sont fixés par l'architecture du processeur. Nous étudierons dans la suite de ce cours l'architecture

3. Certains sont composés d'environ dix milliards de transistors.

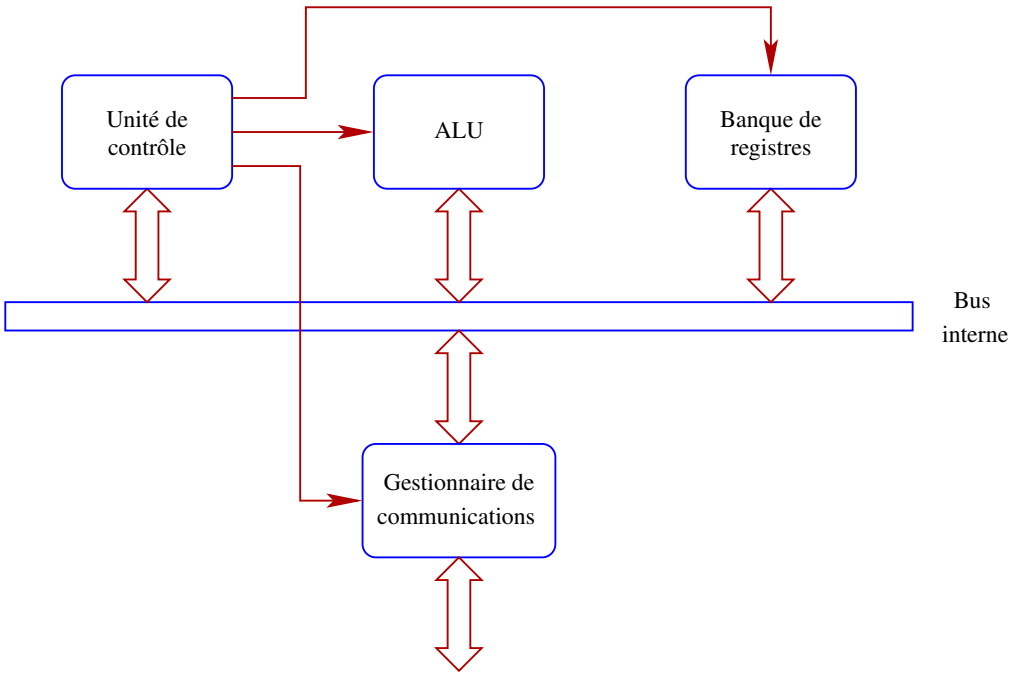
x86-64 utilisée dans les ordinateurs personnels actuels.

— L'*unité arithmétique et logique (Arithmetic Logic Unit, ALU)* est le composant du processeur chargé d'effectuer les opérations de traitement des données. L'ensemble des opérations supportées par l'ALU dépend de l'architecture du processeur. Pour les processeurs simples, ces opérations peuvent se limiter à des manipulations de bits (comme inverser tous les bits d'un octet ou les décaler d'un certain nombre de positions vers la droite ou la gauche) et des additions de nombres. L'ALU de processeurs plus complexes peut être capable de multiplier et diviser des nombres entiers. Elle peut également inclure une *unité de virgule flottante (Floating-Point Unit, FPU)*, capable de manipuler des nombres réels. Les opérations supportées peuvent aller jusqu'au calcul de racines carrées, de logarithmes ou de fonctions trigonométriques.

— Le *bus interne* fournit un canal de communication permettant aux composants du processeur de s'échanger des données. La taille de ce bus, c'est-à-dire la quantité d'information qu'il est capable de transmettre en une opération élémentaire, est une caractéristique importante de l'architecture du processeur.

— Le *gestionnaire de communications* établit un lien entre le bus interne et l'interface extérieure du processeur. Ce gestionnaire est notamment responsable de coordonner les échanges entre le processeur et la mémoire externe ainsi que les périphériques.

— L'*unité de contrôle* implémente le mécanisme d'exécution des programmes. Son rôle consiste à charger les instructions de code machine depuis la mémoire de programme, décoder ces instructions, et piloter les autres composants du processeur de façon à exécuter ces instructions. Le jeu d'instructions disponibles est défini par l'architecture du processeur.



Note : Sur ce diagramme, les flèches larges représentent les canaux d'échange de données, et les flèches simples les ordres envoyés de l'unité de contrôle vers les autres composants, afin d'en contrôler le fonctionnement. □

3.3.3 Le code machine

Lorsqu'un programmeur développe un programme, il le rédige dans un *langage de programmation*, tel que C, C++, Java ou Python. De tels programmes ne peuvent pas être directement exécutés par le processeur ; il est nécessaire de les traduire préalablement en *code machine*. Ce code machine contient des instructions que le processeur est capable d'exécuter, encodées d'une certaine façon.

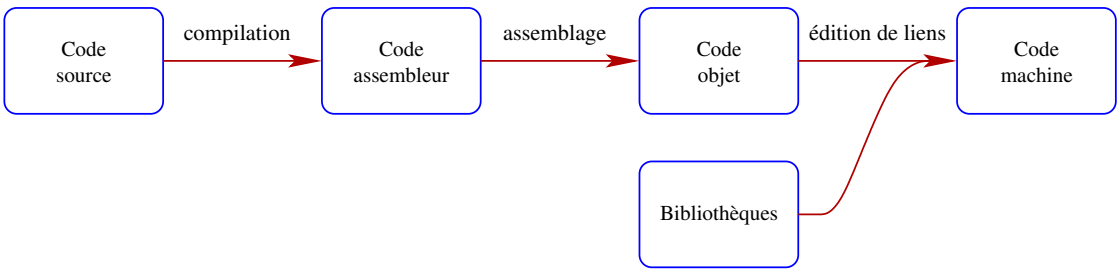
La traduction en code machine d'un code source rédigé dans un langage de programmation peut être réalisée une fois pour toutes pour un programme donné (*compilation*), ou bien au fur et à mesure de l'exécution de celui-ci (*interprétation*). La compilation d'un programme s'effectue habituellement en plusieurs étapes :

1. Le code source est traduit en *code assembleur*, dans lequel chaque instruction correspond à une opération réalisable par le processeur, mais est représentée dans un format lisible.

   Les mécanismes de traduction qui sont mis en œuvre peuvent être complexes [4], et inclure un certain nombre d'étapes intermédiaires (analyse lexicale et syntaxique du code source, optimisation du code généré, ...).

2. Le code assembleur est traduit en *code objet* par le *programme d'assemblage*. Le code objet est une représentation numérique des instructions figurant dans le code assembleur, proche du code machine. La différence entre le code objet et le code machine est que le premier peut contenir des références incomplètes vers du code extérieur, comme par exemple des fonctions utilisées par le programme et implémentées par une bibliothèque.

3. Le code objet est combiné avec du code provenant de *bibliothèques (libraries)* par un *éditeur de liens (linker)*, de façon à obtenir un code machine exécutable.



4. Ces mécanismes sont étudiés en détail dans le cours *Compilers*.

Notes : L'opération d'édition de liens peut être *dynamique*, ce qui signifie qu'elle est alors réalisée au moment où le programme est chargé en mémoire en vue d'être exécuté. L'édition de liens peut également servir à combiner plusieurs fragments de code objet en un seul. □.

3.3.4 L'exécution des instructions

Pour exécuter les instructions, l'unité de contrôle du processeur gère deux registres particuliers :

— Le *registre d'instruction (Instruction Register, IR)* contient à chaque instant le code [5] de l'instruction machine qui est en cours d'exécution.

— Le *compteur de programme (Program Counter, PC)*, appelé *pointeur d'instruction (Instruction Pointer, RIP)* dans l'architecture x86-64, contient l'adresse dans la mémoire de programme de la prochaine instruction à exécuter.

L'unité de contrôle effectue en permanence les opérations suivantes, en suivant un rythme dicté par le signal d'horloge que reçoit le processeur :

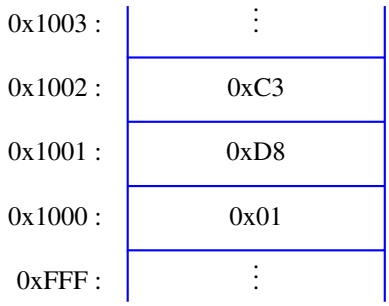
1. Lire depuis la mémoire de programme la valeur pointée par PC et la charger dans IR. Cette valeur représente l'opcode de l'instruction qui est sur le point d'être exécutée.

2. Décoder la valeur de IR afin de déterminer la nature de l'instruction à exécuter.

3. Exécuter cette instruction en commandant les différents composants du processeur (ALU, banque de registres, . . .). Pour les instructions accompagnées de paramètres, la lecture de ces derniers depuis la mémoire de programme et leur décodage font partie de cette opération. Le registre PC est également mis à jour, de façon à le faire pointer vers l'instruction qui suit l'instruction courante.

4. Recommencer à l'étape 1.

Illustration : Considérons un processeur basé sur l'architecture x86-64, dont la mémoire de programme contient les valeurs suivantes.



5. Le code d'une instruction est également appelé *opcode*.

Supposons que le registre RIP contient initialement 0x1000. La première opération de l'unité de contrôle sera de charger dans IR l'octet pointé par RIP. Ce registre IR deviendra donc égal à 0x01.

L'étape suivante consiste à décoder le contenu de IR de façon à identifier l'instruction représentée. Dans l'architecture x86-64, l'opcode 0x01 désigne une opération d'addition, et est toujours suivi d'un octet supplémentaire encodant les opérandes de cette opération.

Pour décoder ces opérandes, l'unité de contrôle charge l'octet situé à l'adresse 0x1001 (donnée par RIP + 1). Dans le cas présent, la valeur 0xD8 signifie que les nombres à additionner doivent être extraits de deux registres de 32 bits appelés EAX et EBX, et que le résultat de l'addition doit être écrit dans EAX.

La lecture du code machine de l'instruction d'addition étant complète, le registre RIP peut à présent être mis à jour. Etant donné que l'opération courante est encodée sur deux octets (un pour l'opcode et un pour les opérandes), RIP doit être incrémenté de deux unités ; ce registre contient alors 0x1002, qui est l'adresse de la prochaine instruction à exécuter.

L'unité de contrôle exécute ensuite l'opération d'addition. La procédure consiste à piloter la banque de registres de façon à lui faire lire les valeurs de EAX et de EBX, et les transférer à l'ALU par l'intermédiaire du bus interne. Au cycle d'horloge suivant, l'ALU reçoit l'ordre d'additionner ces deux valeurs et de placer le résultat sur le bus interne, et la banque de registres charge ce résultat dans EAX.

A ce stade, l'exécution de l'instruction d'addition est terminée, et RIP est égal à 0x1002. La même procédure reprend donc à partir de cette adresse, afin de charger, décoder et exécuter l'instruction située à cette adresse (dont l'opcode est 0xC3). □

Remarque : Certaines instructions peuvent modifier le compteur de programme au cours de leur exécution, de façon à le faire pointer ailleurs que vers l'instruction suivante. Ce mécanisme est utilisé pour implémenter des décisions (qui conduisent le programme à s'exécuter de façon différente selon le résultat d'un calcul) ou des boucles (pour lesquelles l'exécution doit recommencer au même endroit après chaque itération). □

Le procédé d'exécution des instructions que nous venons de décrire est une simplification de la réalité. Les processeurs modernes incluent en effet un certain nombre de mécanismes supplémentaires destinés à améliorer leurs performances :

— Plutôt que de charger, décoder et d'exécuter les instructions séquentiellement, certaines de ces opérations peuvent être réalisées simultanément par des unités fonctionnelles distinctes du processeur. Par exemple, une instruction peut déjà être chargée dans le registre d'instruction pendant que la précédente est toujours en train de s'exécuter. Deux instructions peuvent également être exécutées simultanément si elles font intervenir des registres et des circuits de calcul séparés. Ce mécanisme appelé *pipelining* permet dans certains cas d'exécuter plus d'une instruction par cycle de l'horloge.

— L'accès à la mémoire externe, que ce soit pour des données ou des programmes, peut être

géré par un *gestionnaire de mémoire (Memory Management Unit, MMU)*, implémentant des mécanismes avancés de protection (par exemple, pour empêcher un programme de manipuler des données auxquelles il ne devrait pas avoir accès). Le gestionnaire de mémoire peut aussi traduire les adresses définies par les programmes en adresses de la mémoire physique. Un tel gestionnaire de mémoire peut également recourir à de la *mémoire cache*, déjà évoquée à la section 3.2.1, afin d'éviter de ralentir le processeur par de trop fréquents accès à la mémoire externe.

— Les *processeurs multicœurs* sont composés de plusieurs processeurs individuels appelés *cœurs (cores)*, regroupés dans un seul circuit intégré, ce qui les rend capables d'exécuter simultanément plusieurs programmes. Les différents cœurs travaillent de façon indépendante, mais peuvent partager certains de leurs circuits, notamment une partie de leur mémoire cache.

— Tous les processeurs possèdent un mécanisme d'*interruption*, permettant de suspendre temporairement l'exécution du programme courant en vue d'effectuer une opération urgente (par exemple, réceptionner des données émises par un périphérique), et de reprendre ensuite cette exécution.

L'étude de ces mécanismes sort du cadre de ce cours. Certains d'entre eux figurent au programme des cours *Computation Structures* et *Embedded Systems*.

3.4 L'architecture x86-64

L'architecture x86-64 est présente dans une majorité des ordinateurs personnels actuels. Il s'agit d'une extension à 64 bits (ce qui signifie que le bus interne, la plupart des registres et les instructions sont compatibles avec des données de cette taille) de l'architecture x86 développée à la fin des années 1970.

Les processeurs x86-64 possèdent plusieurs modes de fonctionnement, afin notamment d'assurer une certaine compatibilité avec l'architecture x86. Ces modes conditionnent la façon dont le processeur accède à la mémoire externe, l'ensemble des registres disponibles, et le jeu d'instructions pouvant être exécutées. Dans ce cours, par simplicité, nous ne couvrirons qu'un seul de ces modes, le mode 64 bits, qui est celui le plus couramment utilisé dans les systèmes modernes.

En mode 64 bits, l'organisation de la mémoire du processeur suit le modèle Von Neumann et correspond à un espace d'adressage de 64 bits. En d'autres termes, les adresses valides forment l'intervalle [0, 0xFFFFFFFFFFFFFFFF]. Ces adresses ne correspondent pas nécessairement à celles de la mémoire physique ; en effet, le gestionnaire de mémoire peut mettre en place un mécanisme de traduction d'adresses, ce qui est utile notamment pour implémenter un système d'exploitation. Nous ne considérerons pas ce mécanisme dans ce cours [6].

6. Il est étudié dans le cours *Operating Systems*.

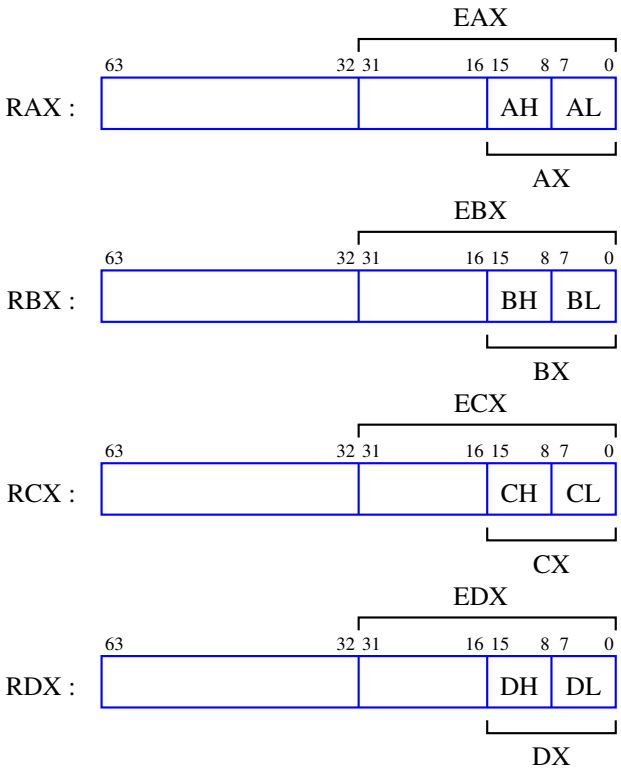
3.4.1 Les registres

L'architecture x86-64 définit 16 *registres généraux* de 64 bits appelés RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8, R9, R10, R11, R12, R13, R14 et R15.

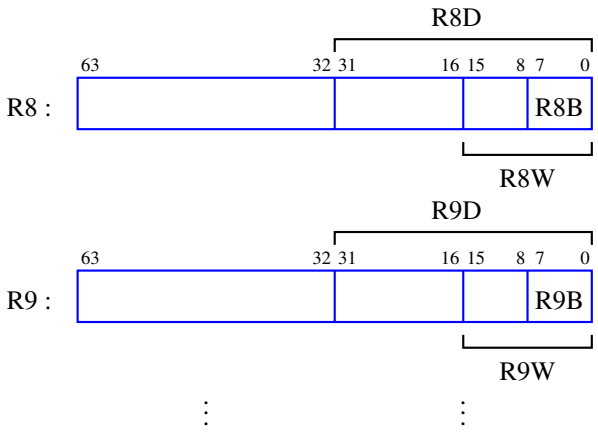
Ces registres peuvent être employés en tant qu'opérandes de la plupart des instructions. Ils sont donc utiles pour retenir temporairement des données produites par une instruction et utilisées ensuite dans le reste du programme.

Certains de ces registres possèdent des modes d'utilisation particuliers. Par exemple, RCX sert de compteur pour une instruction gérant les boucles (qui sera étudiée à la section 3.4.7), RSP sert de pointeur de pile (cf. section 3.4.6), et RAX et RDX sont utilisés pour recueillir le résultat d'opérations de multiplication (cf. section 3.4.4).

Lorsqu'on effectue des opérations impliquant des données encodées sur moins de 64 bits, il est possible d'utiliser des fragments de registres généraux. En remplaçant par "E" le "R" initial du nom des huit premiers registres généraux, on obtient des noms de registres qui correspondent à leurs 32 bits de poids faible. Par exemple, EAX désigne les 32 bits de poids faible de RAX. De façon similaire, les noms de registre AX, BX, CX, DX, BP, SI, DI et SP correspondent aux 16 bits de poids faible de (respectivement) EAX, EBX, ECX, EDX, EBP, ESI, EDI et ESP. Enfin, AH, BH, CH, DH et AL, BL, CL, DL représentent les 8 bits respectivement de poids fort et de poids faible de AX, BX, CX et DX.



Des mécanismes similaires permettent d'accéder aux 32, 16 ou 8 bits de poids faible des registres R8, R9, R10, R11, R12, R13, R14 et R15 : Il suffit d'ajouter respectivement le suffixe "D" (pour *double word*), "W" (pour *word*) ou "B" (pour *byte*) à leur nom. Par exemple, R8W dénote les 16 bits de poids faible du registre R8.



En addition aux registres généraux, l'architecture définit un registre RFLAGS de 64 bits contenant des *drapeaux (flags)*. Les drapeaux sont des bits d'information mis à jour par certaines instructions. Les drapeaux les plus utilisés sont les suivants :

— CF (*Carry Flag*, bit 0 de RFLAGS) : Indique si un report a été produit (CF = 1) ou non (CF = 0) à la position n par une opération arithmétique sur n bits.

— ZF (*Zero Flag*, bit 6 de RFLAGS) : Indique si le résultat d'une opération est nul (ZF = 1) ou non nul (ZF = 0).

— SF (*Sign Flag*, bit 7 de RFLAGS) : Contient le bit le plus significatif du résultat d'une opération (correspondant au bit de signe pour une donnée signée).

— OF (*Overflow Flag*, bit 11 de RFLAGS) : Indique si un dépassement s'est produit (OF = 1) ou non (OF = 0) lors d'une opération arithmétique sur des nombres signés.

Lorsque nous étudierons les instructions de l'architecture x86-64, nous préciserons pour chacune d'entre elles quels sont les drapeaux qu'elle met à jour.

Le registre RIP, discuté à la section 3.3.4, constitue le compteur de programme. Durant l'exécution d'une instruction, il contient l'adresse dans la mémoire de programme de l'instruction suivante.

Enfin, signalons que l'architecture x86-64 définit d'autres registres, liés à la manipulation de nombres en virgule flottante, au mécanisme d'exécution d'instructions particulières et à la configuration du processeur. Nous ne les considérerons pas dans cette première introduction à cette architecture.

3.4.2 Les modes d'adressage

En langage d'assemblage, l'écriture d'une instruction est composée

— d'une *mnémonique*, qui est le nom conventionnel donné à cette instruction (par exemple, `ADD` désigne une opération d'addition).

— de ses opérandes.

Les opérandes peuvent être spécifiées de différentes manières, afin d'indiquer par exemple si la donnée utilisée par une instruction est constante, extraite d'un registre, ou bien lue depuis un emplacement de la mémoire. Une opérande peut être une source, lorsqu'elle fournit une donnée d'entrée, une destination, lorsqu'elle indique où placer le résultat de l'opération, ou les deux.

L'architecture x86-64 définit les *modes d'adressage* suivants.

L'adressage registre

L'adressage *registre* signifie que l'opérande est lue depuis un registre (dans le cas d'une source), ou que le résultat de l'opération doit être placé dans un registre (s'il s'agit d'une destination). Sa syntaxe est égale au nom du registre concerné.

Par exemple, l'instruction `ADD` est accompagnée de deux opérandes, indiquant quelles sont les deux valeurs à additionner. La première opérande indique également la destination de l'opération. L'instruction

```
ADD RAX, RBX
```

qui utilise l'adressage registre pour les deux opérandes a donc pour effet d'additionner deux nombres de 64 bits extraits des registres RAX et RBX, et d'écrire le résultat dans RAX.

L'adressage immédiat

L'adressage *immédiat*, ou *littéral*, indique qu'une opérande est égale à une constante. Sa syntaxe correspond à la valeur de cette dernière.

Par exemple, l'instruction

```
ADD RDI, 0x10
```

ajoute 16 à la valeur du registre RDI.

Bien sûr, cela n'aurait pas de sens d'employer l'adressage immédiat pour la première opérande de l'instruction `ADD`, puisque celle-ci indique également la destination de l'opération. Comme nous le verrons plus tard, chaque instruction définit l'ensemble des modes d'adressage qu'elle supporte pour ses opérandes.

L'adressage direct

L'adressage *direct* définit une opérande qui correspond à un emplacement de la mémoire dont l'adresse est fixée. Cet emplacement peut représenter une source ou une destination. Une telle opérande s'écrit

<*taille*> `ptr [` <*adresse*> `]`,

où <*adresse*> est l'adresse de l'emplacement mémoire concerné, et où <*taille*> est un des mots-clés `qword`, `dword`, `word` ou `byte`, pour des données de respectivement 64, 32, 16 et 8 bits.

Par exemple, l'instruction

```
ADD dword ptr [0x1234], R8D
```

ajoute à l'entier de 32 bits situé à l'adresse 0x1234 de la mémoire la valeur du registre R8D, c'est-à-dire les 32 bits de poids faible du registre R8.

Notes :

— Pour représenter les entiers sur plusieurs octets de la mémoire, l'architecture x86-64 utilise la convention petit-boutiste.

— Comme discuté à la section 3.2.4, lorsqu'on accède à une donnée répartie sur plusieurs cellules de la mémoire, il faut veiller à ce qu'elle soit convenablement alignée. □

L'adressage indirect

L'adressage *indirect* indique qu'une opérande est située en mémoire, à une adresse donnée par le contenu d'un registre. Sa syntaxe est similaire à celle de l'adressage direct, en remplaçant la valeur numérique de l'adresse par le nom du registre qui la contient.

Par exemple, l'instruction

```
ADD AH, byte ptr [RBX]
```

commence par lire le contenu du registre RBX, et l'interprète comme une adresse. Ensuite, l'instruction lit l'octet situé à cette adresse (en d'autres termes, l'octet pointé par RBX), et en ajoute la valeur à celle du registre AH (c'est-à-dire la partie du registre RAX située entre les positions 8 et 15).

Note : Puisque le modèle mémoire x86-64 définit des adresses de 64 bits, l'adressage indirect fait nécessairement intervenir un registre de 64 bits. □

L'adressage indirect indexé

L'adressage *indirect indexé* est une variante de l'adressage indirect, dans laquelle le pointeur vers l'opérande est obtenu en ajoutant la valeur de deux registres généraux (la *base* et l'*index*) à une constante (le *déplacement*). Il est également possible de multiplier l'index par un facteur égal à 2, 4 ou 8. La syntaxe de cet adressage est la suivante :

<*taille*> `ptr [` <*base*> + <*facteur*> * <*index*> + <*déplacement*> `]`,

où

- <*taille*> est `qword`, `dword`, `word` ou `byte` selon la taille de la donnée concernée.
- <*base*> et <*index*> sont des registres de 64 bits.
- <*facteur*> vaut 1, 2, 4 ou 8.
- <*déplacement*> est une constante signée de 32 bits maximum.

Note : Certains éléments peuvent être omis, comme par exemple un facteur égal à 1 (et le symbole * qui l'accompagne) ou un déplacement égal à 0 (avec le symbole + qui le précède). Pour spécifier un déplacement négatif, on remplace le signe + par -. □

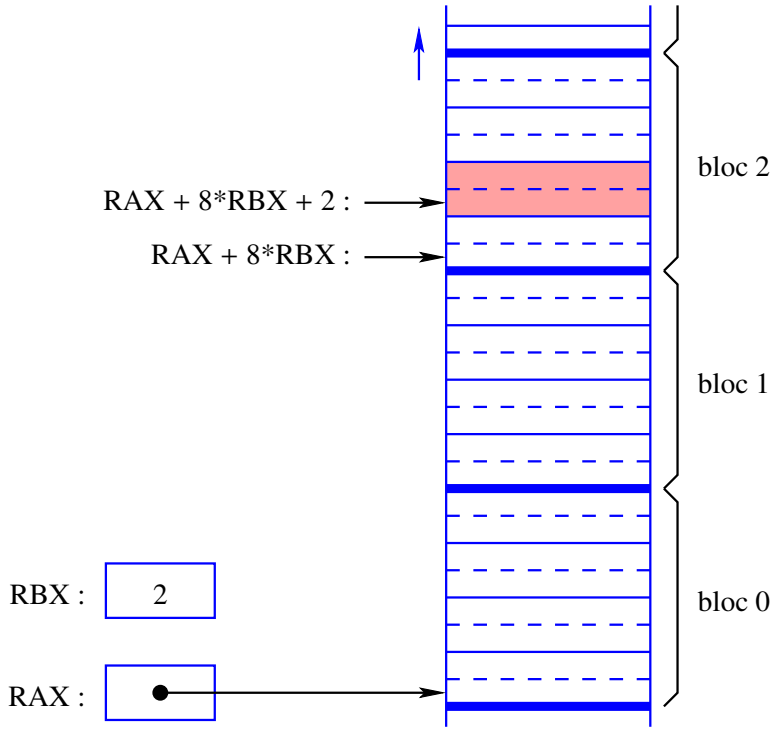
Par exemple, l'instruction

```
ADD DX, word ptr [RAX + 8*RBX + 2]
```

consulte d'abord les registres RAX et RBX, et calcule ensuite la valeur de l'expression RAX+ $8 \times$RBX$+2$. Le résultat est interprété comme une adresse à partir de laquelle on lit un nombre de

16 bits, qui est enfin ajouté à celui contenu dans le registre DX (c'est-à-dire les 16 bits de poids faible de RDX).

L'intérêt de l'adressage indirect indexé est qu'il permet d'accéder facilement à des structures de données linéaires. Considérons par exemple un tableau contenant un certain nombre de blocs de 8 octets, chacun de ces blocs étant composé de 4 champs entiers représentés sur 16 bits. L'adresse du tableau (c'est-à-dire, de son premier octet) est placée dans le registre RAX. Dans ce cas de figure, l'instruction de l'exemple précédent lit le deuxième champ du bloc dont le numéro est donné par RBX (en supposant que leur numérotation commence à zéro), et l'ajoute à DX.



Les sections suivantes sont consacrées à l'étude des instructions les plus importantes de l'architecture x86-64. Pour chacune d'entre elles, les modes d'adressage autorisés seront précisés à l'aide de la notation suivante :

— *imm* pour un adressage immédiat.

— *reg* pour un adressage registre.

— *mem* pour un adressage direct, indirect ou indirect indexé.

Remarque : Certaines contraintes sont imposées par l'architecture, restreignant dans quelques cas particuliers les registres qui peuvent être utilisés par les instructions. Dans cette première introduction à la programmation en assembleur, nous ignorerons ces contraintes. Le lecteur intéressé par ces détails peut consulter la documentation officielle de l'architecture x86-64. □

3.4.3 Les instructions de manipulation des données

L'instruction `MOV`

Cette instruction sert à déplacer des données. Elle possède deux opérandes, la première indiquant la destination du transfert et la seconde sa source. Ces deux opérandes doivent être de même taille.

Exemples :

— L'instruction

```
MOV EBX, dword ptr [0x100]
```

effectue une lecture de quatre octets en mémoire à partir de l'adresse 0x100, et les place dans le registre EBX.

— L'instruction

```
MOV byte ptr [RAX + RSI - 4], 0xFF
```

écrit l'octet 0xFF en mémoire, à l'adresse donnée par la somme des registres RAX et RSI moins 4. □

L'instruction `MOV` n'affecte pas les drapeaux. Elle supporte les modes d'adressage suivants :

Op.1	Op.2
reg	*imm*
mem	*imm*
reg	*reg*
reg	*mem*
mem	*reg*

En d'autres termes, il n'est pas permis d'utiliser un adressage qui représente un accès à la mémoire (c'est-à-dire, direct, indirect ou indirect indexé) à la fois pour la première et pour la deuxième opérande. Cette contrainte résulte d'une limitation du circuit de décodage et d'exécution des instuctions. Il n'est évidemment pas non plus permis d'utiliser un adressage immédiat comme destination.

L'instruction XCHG

Cette instruction échange la valeur de ses deux opérandes, c'est-à-dire que la valeur présente dans la première opérande avant son exécution est transférée dans la deuxième, et vice-versa. Les drapeaux ne sont pas affectés. Les modes d'adressage permis sont les suivants :

Op.1	Op.2
reg	*reg*
reg	*mem*
mem	*reg*

Le cas où les deux opérandes de cette instruction désignent le même registre est particulier ; dans ce cas, l'opération n'a aucun effet. L'instruction

```
XCHG EAX, EAX
```

est encodée par un seul octet de code machine [7], que l'on peut utiliser comme valeur de remplissage dans un programme lorsqu'on souhaite, par exemple, aligner convenablement l'adresse de l'instruction suivante. L'instruction fictive `NOP` *(No OPeration)*, sans opérande, est une abréviation de cette instruction.

3.4.4 Les instructions arithmétiques

L'instruction ADD

L'instruction `ADD` a déjà été introduite à la section 3.4.2. Cette instruction prend deux opérandes entières, les additionne, et place le résultat de cette opération dans la première. Par exemple, l'instruction

```
ADD R10, -1
```

décrémente la valeur du registre R10.

7. Cet octet est égal à 0x90.

Les modes d'adressage supportés sont identiques à ceux de l'instruction MOV. Les drapeaux CF, ZF, SF et OF sont mis à jour en fonction du résultat de l'opération.

Note : Comme nous l'avons vu au chapitre 2, l'opération d'addition s'effectue de la même façon pour des nombres non signés et pour des nombres signés représentés par complément à deux. L'instruction `ADD` peut donc être employée dans les deux cas. □

L'instruction `SUB`

Cette instruction est similaire à l'instruction `ADD`, mais au lieu d'ajouter ses deux opérandes, elle retire la seconde à la première. Tout comme `ADD`, la première opérande sert également de destination pour le résultat de l'opération. Par exemple, l'instruction

```
SUB R10, 1
```

a le même effet que celle de l'exemple précédent.

Cette instruction possède les mêmes modes d'adressage que `ADD`, et met à jour les mêmes drapeaux.

L'instruction `CMP`

Il s'agit d'une variante de l'instruction `SUB`, dont elle partage les modes d'adressage. Comme celle-ci, elle calcule la différence entre ses deux opérandes, mais contrairement à `SUB`, le résultat de cette opération n'est pas écrit à l'endroit spécifié par la première opérande. Le seul effet de l'instruction `CMP` est donc de modifier la valeur des drapeaux. L'objectif est que ceux-ci puissent ensuite être consultés pour orienter une décision ou effectuer une opération conditionnelle. Par exemple, l'instruction

```
CMP EAX, EBX
```

compare la valeur des deux registres de 32 bits EAX et EBX, en en calculant la différence Δ = EAX − EBX. Le résultat Δ de ce calcul n'est écrit nulle part. En revanche, les drapeaux sont mis à jour. Cela signifie que l'on aura ZF = 1 si et seulement si $\Delta = 0$, c'est-à-dire si EAX = EBX. Après avoir exécuté cette instruction, tester si EAX et EBX sont égaux revient donc à déterminer si le drapeau ZF est levé ou non.

De même, on aura SF = 1 si et seulement si Δ est strictement négatif, c'est-à-dire si EAX < EBX en supposant que ces registres contiennent des nombres signés. En examinant les drapeaux

ZF et SF, on peut donc déterminer si le contenu de EAX est inférieur, égal ou supérieur à celui de EBX, ce qui permet d'implémenter différents types de décisions conditionnelles impliquant ces registres.

Les instructions permettant d'orienter l'exécution du programme vers différentes routines en fonction de l'état des drapeaux seront étudiées à la section 3.4.7.

Les instruction `INC` et `DEC`

Ces instructions admettent une seule opérande, pour laquelle les modes d'adressage permis sont les suivants :

Op.1
reg
mem

Leur effet est d'incrémenter (`INC`) ou de décrémenter (`DEC`) leur opérande, qui sert donc à la fois de source et de destination. Par exemple, l'instruction

```
INC byte ptr [RBX]
```

ajoute 1 à l'octet pointé par RBX. Comme les autres opérations arithmétiques, les opérations d'incrément et de décrément s'effectuent dans un arithmétique modulo 2^n, où n est le nombre de bits de la valeur manipulée. Dans l'exemple précédent, si l'octet pointé par RBX vaut initialement 0xFF, cette instruction lui attribuera ainsi la valeur 0.

Le drapeau CF est préservé par cette instruction, ce qui permet d'utiliser celle-ci dans des situations où une instruction ADD équivalente le modifierait. Les drapeaux ZF, SF et OF sont mis à jour en accord avec le résultat de l'opération.

L'instruction `MUL`

Cette instruction calcule le produit de deux nombres non signés représentés sur n bits, avec $n \in \{8, 16, 32, 64\}$. Le résultat de cette opération est donc représenté, en toute généralité, sur $2n$ bits. A la différence de l'opération d'addition, le fait que les nombres traités sont non signés est ici important. En effet, comme cela a été discuté au chapitre 2, chaque opérande doit préalablement être étendue à une représentation sur $2n$ bits avant de commencer à calculer le produit. Cette extension s'effectue différemment pour les nombres non signés (auxquels on ajoute des bits nuls) et pour les nombres signés (dont on répète le bit de signe).

L'instruction `MUL` ne prend qu'une seule opérande, dont les modes d'adressage permis sont les suivants :

Op.1
reg
mem

L'opération effectuée dépend de la taille de cette opérande :

— 8 *bits :* L'opérande est multipliée par AL, et le résultat est placé dans AX.

— 16 *bits :* L'opérande est multipliée par AX, et le résultat est placé dans la paire de registres DX :AX. Cela signifie que les 16 bits de poids fort sont écrits dans DX, et les 16 bits de poids faible dans AX.

— 32 *bits :* L'opérande est multipliée par EAX, et le résultat est placé dans EDX :EAX.

— 64 *bits :* L'opérande est multipliée par RAX, et le résultat est placé dans RDX :RAX.

Par exemple, l'instruction

```
MUL dword ptr [0x1234]
```

indique une multiplication de deux nombres non signés de 32 bits chacun. Le premier est lu depuis le registre EAX, et le second depuis l'adresse 0x1234 de la mémoire. Les 32 bits de poids fort du résultat sont écrits dans le registre EDX, et les 32 bits de poids faible dans EAX. (Le contenu initial de EAX est donc perdu à l'issue de cette opération.)

L'instruction `MUL` modifie les drapeaux de la façon suivante. Pour une multiplication de deux nombres de n bits, les drapeaux CF et OF sont mis à zéro si les n bits de poids fort du résultat sont tous nuls (ce qui signifie que le produit est représentable sur n bits). Ces drapeaux sont mis à un sinon. Les autres drapeaux sont modifiés de façon imprévisible.

L'instruction `IMUL`

Cette instruction est similaire à l'instruction `MUL`, et possède les mêmes modalités d'utilisation, mais calcule le produit de deux nombres signés plutôt que non signés. Les drapeaux CF et OF sont, comme pour l'instruction `MUL`, mis à zéro si le résultat de la multiplication de deux nombres de n bits, avec $n \in \{8, 16, 32, 64\}$, est représentable sur n bits. Ces drapeaux sont donc mis à un lorsque les n bits de poids fort du résultat ne sont pas tous des copies du bit situé à la position $n - 1$ (qui correspond au bit de signe).

L'instruction `IMUL` possède également deux autres formes, dans lesquelles elle prend 2 ou 3 opérandes. Nous ne les étudierons pas dans ce cours.

3.4.5 Les instructions logiques

Les instructions logiques effectuent des opérations qui font intervenir les bits individuels d'information contenus dans les registres ou les emplacements de la mémoire.

Les instructions `AND`, `OR` et `XOR`

Ces instructions s'utilisent de la même façon que l'instruction `ADD`, et partagent avec elle les mêmes modes d'adressage. Elles appliquent une *opération booléenne* à chaque paire de bits situés à la même position dans leurs deux opérandes. Chaque bit du résultat est écrit à la même position, à l'endroit désigné par la première opérande. Les trois opérations booléennes définies par ces instructions sont les suivantes :

— `AND` : Le résultat est égal à 1 si et seulement si les deux bits sont égaux à 1 *(et logique)*.

— `OR` : Le résultat est égal à 1 si et seulement si au moins un des deux bits est égal à 1 *(ou inclusif)*.

— `XOR` : Le résultat est égal à 1 si et seulement si exactement un des deux bits est égal à 1 *(ou exclusif)*.

Ces instructions permettent de forcer à 0 (`AND`), de forcer à 1 (`OR`) ou d'inverser (`XOR`) les bits situés à des positions données d'une valeur.

Exemples :

— L'instruction

```
AND byte ptr [0x100], 0xFC
```

force à zéro les deux bits de poids faible de l'octet situé à l'adresse 0x100 de la mémoire. En effet, l'octet 0xFC donné par la deuxième opérande est formé par 2 bits nuls (aux positions 0 et 1) et 6 bits égaux à 1 (aux positions 2 à 7).

— L'instruction

```
OR AL, 0xF0
```

force à un les quatre bits de poids fort du registre AL.

— L'instruction

```
XOR RBX, 0xFF00
```

inverse les bits situés aux positions 8 à 15 du registre RBX. □

Ces instructions abaissent (c'est-à-dire, mettent à zéro) les drapeaux CF et OF. Elles mettent à jour les drapeaux ZF et SF en fonction du résultat de l'opération.

L'instruction `NOT`

Cette instruction admet une seule opérande, dont les modes d'adressage permis sont les suivants :

Op.1
reg
mem

Elle a pour effet d'inverser tous les bits de son opérande, en d'autres termes, d'en calculer le complément à un. Par exemple, si le registre DX contient initialement 0x5A, alors l'instruction

```
NOT DX
```

lui attribuera la valeur 0xFFA5. Notons que cette instruction est équivalente à

```
XOR DX, 0xFFFF
```

.

L'instruction `NOT` ne modifie pas l'état des drapeaux.

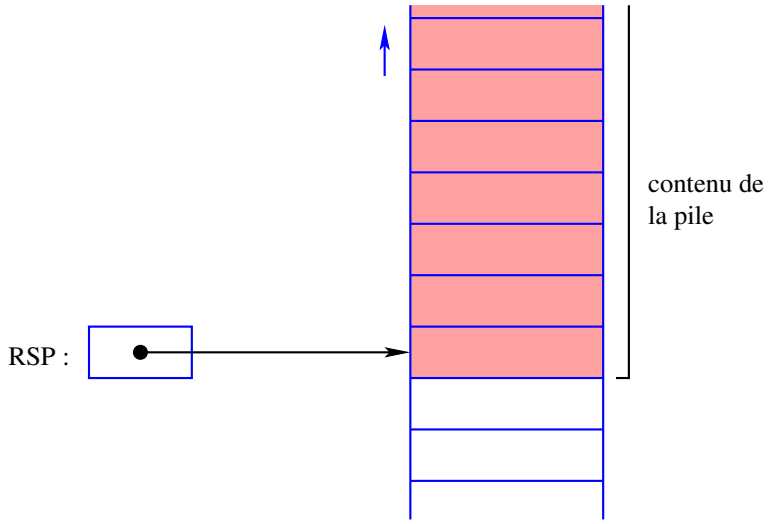
3.4.6 Les instructions de manipulation de la pile

Comme une très grande majorité de processeurs, l'architecture x86-64 implémente un mécanisme de gestion de *pile*. Une pile est une structure de données permettant de retenir un ensemble de valeurs, et d'y accéder selon une politique *LIFO (Last-In First-Out)*. Cette structure se manipule grâce à deux opérations de base : *Empiler* une valeur *(push)* ajoute cette valeur au sommet de la pile ; *dépiler (pop)* retire de la pile la valeur située à son sommet. La politique d'accès LIFO signifie que la donnée qui sera dépilée est, parmi celles qui sont présentes sur la pile, celle qui a été empilée en dernier lieu.

La pile gérée par le processeur sert à mémoriser des données temporaires, comme les paramètres et les variables locales des fonctions. Elle est également utilisée pour sauvegarder la valeur de certains registres lors de l'appel à une sous-routine. Par exemple, si un fragment de code utilise les registres RAX et RBX, et doit invoquer une fonction qui modifie potentiellement ces deux registres, il lui suffit de les empiler avant d'appeler cette fonction, et de les dépiler (dans l'ordre inverse) ensuite. L'intérêt du mécanisme de pile est que la fonction appelée peut

elle-même employer le même mécanisme pour invoquer des sous-fonctions. Les techniques de gestion de la pile mises en œuvre lors de l'appel d'une fonction seront étudiées en détail au chapitre suivant.

Dans l'architecture x86-64, le contenu de la pile est représenté par un ensemble de cellules consécutives de la mémoire. Ce contenu croît dans la direction des adresses décroissantes ; cela signifie que si l'on empile successivement un ensemble de valeurs, celles-ci seront écrites à des adresses de plus en plus petites. L'emplacement du sommet de la pile est représenté par le contenu du registre RSP ; celui-ci pointe à tout moment vers le dernier octet qui a été empilé.



L'instruction `PUSH`

L'instruction `PUSH` permet d'empiler une valeur de 64 bits. Cette valeur peut être constante, extraite d'un registre, ou bien lue en mémoire, comme le montrent les modes d'adressage permis :

Op.1
imm
reg
mem

Notes :

— Ces modes d'adressage ne peuvent faire intervenir qu'un registre de 64 bits. Les accès à la mémoire doivent spécifier la taille `qword`.

— Pour des raisons de compatibilité avec l'architecture x86, d'autres modalités d'utilisation de l'instruction `PUSH` existent, que nous n'étudierons pas dans ce cours. □

L'instruction `PUSH` effectue les opérations suivantes :

1. Décrémenter le registre RSP de 8 unités, ce qui revient à allouer 8 nouveaux octets au sommet de la pile.

2. Recopier l'opérande à l'endroit pointé par RSP.

Par exemple l'instruction

```
PUSH 0x42
```

empile la valeur constante 0x42, représentée sur 8 octets.

L'instruction `PUSH` n'affecte pas les drapeaux.

L'instruction POP

Cette instruction réalise l'opération symétrique de l'instruction `PUSH`. Elle dépile une valeur de 64 bits, et la place à l'endroit spécifié par son opérande. Celle-ci peut prendre la forme d'un registre de 64 bits, ou d'un accès à la mémoire de taille `qword` :

Op.1
reg
mem

L'opération effectuée par `POP` consiste donc à

1. Lire 8 octets depuis l'emplacement de la mémoire pointé par RSP, et les transférer à l'endroit spécifié par l'opérande.

2. Incrémenter le registre RSP de 8 unités.

Tout comme `PUSH`, cette instruction préserve l'état des drapeaux.

Exemple : La séquence d'instructions suivante permute le contenu des registres R8 et R9 (en supposant que le registre RSP pointe vers une zone de la mémoire pouvant accueillir la pile) :

```
PUSH R8
PUSH R9
POP R8
POP R9
```

□

3.4.7 Les instructions de contrôle

Normalement, l'exécution d'un programme est séquentielle : Les instructions successives du programme sont exécutées les unes après les autres, dans l'ordre où elles sont situées en mémoire.

Les instructions de contrôle servent à modifier ce mode de fonctionnement, en permettant de continuer l'exécution du programme à un autre endroit.

L'instruction `JMP`

L'instruction `JMP` *(jump)* définit un *saut inconditionnel*. Cette instruction admet une opérande, dont la valeur indique l'adresse en mémoire de la prochaine instruction à exécuter. En d'autres termes, l'instruction `JMP` recopie son opérande dans le compteur de programme. Les modes d'adressage permis sont les suivants :

Op.1
imm
reg
mem

Les drapeaux ne sont pas affectés.

Par exemple, l'instruction

```
JMP 0x1000
```

continue l'exécution du programme à partir de l'instruction située à l'adresse 0x1000. En pratique, les programmeurs n'écrivent pas explicitement une telle adresse dans leur code, mais la remplacent par une *étiquette* dont la valeur sera automatiquement calculée par le programme d'assemblage. Par exemple, le fragment de code suivant implémente une boucle infinie :

```
boucle:  NOP
         NOP
         NOP
         JMP boucle
```

Dans cet exemple, l'opérande de l'instruction `JMP` utilise le mode d'adressage immédiat. L'étiquette `boucle` est un symbole qui représente l'adresse où sera placée la première instruction de ce programme. La valeur de ce symbole sera calculée par le programme d'assemblage. L'intérêt de cette façon de procéder est que si le programme est ultérieurement modifié, par exemple en ajoutant des instructions avant la boucle, alors la valeur attribuée à l'étiquette sera automatiquement recalculée.

L'instruction `JMP` peut employer un adressage indirect, ou indirect indexé. Par exemple, l'instruction

```
JMP qword ptr [8*RBX + 0x1000]
```

extrait un nombre de 64 bits depuis un tableau situé à l'adresse 0x1000, à la position donnée par le contenu de RBX. Elle effectue alors un saut vers l'instruction pointée par ce nombre. Ce mécanisme permet notamment d'implémenter une décision multiple, qui nécessite de continuer l'exécution du programme dans un certain nombre de branches en fonction de la valeur d'une expression (de façon analogue à l'instruction `switch` du langage C).

Les instructions de saut conditionnel

Ces instructions sont similaires à `JMP` et en possèdent les mêmes modalités d'utilisation, mais elles effectuent un *saut conditionnel*, c'est-à-dire que ce saut n'a lieu que si une condition bien précise est satisfaite. Dans le cas contraire, l'exécution du programme continue à l'instruction suivante.

Pour un premier groupe d'instructions de saut conditionnel, la condition à satisfaire porte sur l'état d'un drapeau :

Instruction	Condition
`JC`	CF = 1
`JNC`	CF = 0
`JZ`	ZF = 1
`JNZ`	ZF = 0
`JS`	SF = 1
`JNS`	SF = 0
`JO`	OF = 1
`JNO`	OF = 0

Par exemple, le fragment de code suivant calcule la somme de deux nombres contenus dans R8 et R9, et effectue ensuite un saut vers l'instruction repérée par l'étiquette `report` si cette addition a produit un report à la position 64 :

```
ADD R8, R9
JC report
```

Pour le deuxième groupe d'instructions, la condition est exprimée en termes des opérandes *op1* et *op2* d'une instruction `CMP` exécutée avant l'instruction de saut :

Instruction	Condition	
`JE`	$op1 = op2$	
`JNE`	$op1 \neq op2$	
`JG`	$op1 > op2$	(valeurs signées)
`JGE`	$op1 \geq op2$	(valeurs signées)
`JL`	$op1 < op2$	(valeurs signées)
`JLE`	$op1 \leq op2$	(valeurs signées)
`JA`	$op1 > op2$	(valeurs non signées)
`JAE`	$op1 \geq op2$	(valeurs non signées)
`JB`	$op1 < op2$	(valeurs non signées)
`JBE`	$op1 \leq op2$	(valeurs non signées)

Par exemple, les instructions

```
CMP EAX, 0xFFFF
JA dépassement
```

effectuent en saut vers l'étiquette `dépassement` seulement si EAX contient une valeur (interprétée comme un nombre non signé) supérieure à 0xFFFF.

Remarque : Certaines instructions de saut conditionnel sont équivalentes, et représentent en fait la même instruction de code machine. Par exemple, `JE` et `JZ` sont deux instructions identiques. □

Tout comme pour l'instruction `JMP`, les drapeaux ne sont pas affectés par les instructions de saut conditionnel.

L'instruction LOOP

Comme son nom l'indique, l'instruction `LOOP` offre un mécanisme permettant d'implémenter une boucle. Cette instruction prend une seule opérande, de la même forme que celle des instructions de saut. Les modes d'adressage permis sont identiques.

L'instruction `LOOP` effectue les opérations suivantes :

1. Décrémenter RCX d'une unité.
2. Tester la nouvelle valeur de RCX. Si celle-ci est différente de zéro, effectuer un saut à l'adresse spécifiée par l'opérande.

Par exemple, le code suivant définit une boucle qui effectuera 256 itérations :

```
        MOV RCX, 0x100
boucle: NOP
        NOP
        NOP
        LOOP boucle
```

Notes :

— L'instruction `LOOP` utilise obligatoirement le registre RCX comme compteur de boucle.

— La valeur initiale de RCX ne correspond pas toujours au nombre d'itérations à effectuer. En effet, si cette valeur est nulle, alors 2^{64} opérations de décrémentation seront nécessaires avant de pouvoir sortir de la boucle (ce qui, pour un ordinateur actuel, représente un temps d'exécution particulièrement prohibitif).

L'instruction LOOP ne modifie pas l'état des drapeaux.

Les instructions CALL et RET

Ces instructions permettent de définir des *sous-routines*. Une sous-routine est une portion de code vers laquelle on souhaite pouvoir effectuer un saut à partir d'autres parties du programme. Lorsque la sous-routine se termine, le programme doit reprendre son exécution à l'instruction qui suit l'instruction de saut.

L'instruction `CALL` prend une opérande similaire à celle de `JMP`, avec les mêmes modes d'adressage autorisés. Elle effectue les opérations suivantes :

1. Empiler la valeur courante de RIP, c'est-à-dire l'adresse de l'instruction suivante du programme. Il s'agit de l'endroit où le programme devra reprendre son exécution lorsque la sous-routine sera terminée.

2. Effectuer un saut à l'adresse donnée par l'opérande.

L'instruction `RET` indique la fin de l'exécution de la sous-routine. Elle ne prend pas d'opérande, et effectue l'opération réciproque de `CALL` :

1. Dépiler un entier de 64 bits et l'interpréter comme une adresse.

2. Effectuer un saut vers cette adresse.

Ces instructions n'affectent pas les drapeaux.

Par exemple, le fragment de programme suivant définit une fonction `minswap` qui s'assure que les contenus de ECX et EDX (interprétés comme des nombres signés) satisfont ECX < EDX, et les permute si ce n'est pas le cas. Cette fonction est ensuite invoquée plusieurs fois :

```
minswap:  CMP ECX, EDX
          JL sortie
          XCHG ECX, EDX
sortie:   RET
          ...
          MOV ECX, dword ptr [0x100]
          MOV EDX, dword ptr [0x104]
          CALL minswap
          ...
          MOV ECX, dword ptr [0x108]
          MOV EDX, dword ptr [0x10C]
          CALL minswap
          ...
```

3.5 Exercices

3.5.1 Encodage de données en mémoire

1. Un processeur d'architecture petit-boutiste accède à une mémoire dont le contenu est le suivant :

0x1237 :	0xC2
0x1236 :	0x20
0x1235 :	0x00
0x1234 :	0x00

   (a) Quelle est la valeur de l'entier sur 32 bits situé à l'adresse 0x1234 ?

   (b) Quelle est la valeur du nombre réel situé à cette adresse, en supposant qu'il soit encodé dans le format IEEE 754 en simple précision ?

3.5.2 Fonctionnement d'un processeur

Pour cette série d'exercices, nous considérons un processeur fictif capable d'effectuer les opérations suivantes :

— Placer des valeurs constantes dans des registres.

— Lire et écrire des octets en mémoire, à des adresses constantes ou pointées par un registre. La destination des données lues et la source des données écrites sont toujours des registres.

— Effectuer des additions, des soustractions, et des comparaisons de valeurs de 8 bits contenues dans des registres.

— Réaliser un saut vers une autre partie du programme, soit de façon inconditionnelle, soit en fonction du résultat d'une comparaison.

Les exercices consistent à développer, pour chacun des problèmes suivants, un algorithme permettant de les résoudre, en n'employant que ces opérations. Les résultats peuvent être écrits en pseudocode. Le nom des registres et le nombre de registres disponibles peuvent être librement choisis. Les données d'entrée de chaque problème sont fournies dans des registres, ou placées en mémoire.

1. Écrire une valeur constante donnée dans tous les emplacements d'un tableau d'octets, donné par son adresse et sa taille.

2. Recopier un tableau d'octets, donné par son adresse et sa taille, vers une autre adresse donnée de la mémoire.

   Attention, il est possible que les zones de mémoire associées au tableau initial et à sa copie se recouvrent.

3. Retourner un tableau d'octets, donné par son adresse et sa taille, c'est-à-dire en permuter les éléments de façon à ce que le premier prenne la place du dernier, et vice-versa, le second la place de l'avant-dernier, et vice-versa, et ainsi de suite.

4. Calculer la somme de deux nombres représentés de façon petit-boutiste, à l'aide de n octets chacun. Les données d'entrée sont la valeur de n, et deux pointeurs vers la représentation des nombres.

5. Compter le nombre d'octets nuls dans un tableau d'octets d'adresse et de taille données.

6. En langage C, une chaîne de caractères est représentée par un tableau d'octets suivis par un octet nul. A partir d'un pointeur vers une telle chaîne de caractères, calculer sa longueur.

7. Convertir, dans une chaîne de caractères ASCII terminée par un zéro située à une adresse donnée, toutes les lettres minuscules en majuscules. Les autres caractères ne doivent pas être modifiés.

8. Dans un tableau d'octet d'adresse et de taille données, calculer la longueur de la plus longue suite d'octets consécutifs identiques.

9. Déterminer si deux chaînes de caractères terminées par un zéro, d'adresses données, sont égales.

3.5.3 Modes d'adressage

1. Les instructions suivantes effectuent des transferts de 16 bits de données. On demande de décomposer chacune d'entre elles en deux instructions `MOV` réalisant la même opération, mais avec des opérandes de 8 bits.

   (a) `MOV AX, 0x1234`

   (b) `MOV AX, word ptr [0x1234]`

   (c) `MOV BX, CX`

   (d) `MOV word ptr [RBX], CX`

   (e) `MOV word ptr [RBP + RSI - 6], 0x32`

2. Après l'exécution de la séquence d'instructions suivante, quel sera le contenu du registre AX ?

```
MOV RBX, 0
MOV RSI, 0
MOV AX, 0x100
MOV BX, AX
MOV word ptr [RBX + 1], AX
MOV word ptr [0x100], AX
MOV SI, word ptr [RBX + 1]
MOV AL, byte ptr [RSI]
MOV AH, byte ptr [RBX + RSI - 0x101]
```

3. Pourquoi les instructions suivantes sont-elles incorrectes ?

   (a) `MOV RAX, CX`

   (b) `MOV byte ptr [EAX], CL`

   (c) `MOV 0x1234, AX`

   (d) `MOV EDX, qword ptr [RBP + RBX]`

   (e) `MOV RDX, RBP + RSI`

   (f) `MOV RDX, qword ptr[RBP - RSI - 9]`

   (g) `MOV RDX, qword ptr[2 * RBP + RBX + RCX]`

3.5.4 Programmation x86-64

Pour chacun des exercices de la section 3.5.2, traduire votre solution en instructions x86-64. Préciser dans chaque cas les registres utilisés pour les données d'entrée et de sortie.

3.5.5 Exercices supplémentaires

1. *(Examen de première session, 2018)* Le plus simplement possible, décrivez l'effet des instructions x86-64 suivantes :

   (a) `AND word ptr[R8], 0xff`

   (b) `CMP EAX, EAX`

   (c) `PUSH qword ptr[RAX]`

   (d) `RET`

2. *(Examen de seconde session, 2018)* Dans un programme assembleur x86-64, on souhaite écrire la valeur v dans la k-ème case d'un tableau d'entiers (de 32 bits) pointé par le registre RAX. Le premier élément du tableau correspond à $k = 1$, le second à $k = 2$, et ainsi de suite. Les valeurs de v et de k sont respectivement disponibles dans les registres R8D et RSI. On demande d'écrire une instruction x86-64 réalisant cette opération d'écriture.

3. *(Examen de seconde session, 2018)* Le plus simplement possible, décrivez l'effet des instructions x86-64 suivantes :

   (a) `IMUL AX`

   (b) `SUB R8D, -1`

   (c) `POP qword ptr[0x100]`

   (d) `CALL qword ptr[8 * RAX]`

4. *(Examen de première session, 2019)* Décrivez, le plus simplement possible, l'effet des fragments de code assembleur x86-64 suivants. (On suppose que la pile est correctement configurée avant leur exécution.)

   — Fragment 1 :

   ```
   PUSH RBX
   PUSH RAX
   MOV  AL, byte ptr[RSP+0]
   MOV  BL, byte ptr[RSP+7]
   MOV  byte ptr [RSP+0], BL
   MOV  byte ptr [RSP+7], AL
   POP  RAX
   POP  RBX
   ```

   — Fragment 2 :

   ```
   PUSH RDX
   CMP  AX, 0
   JGE  p
n: MOV RDX, -1
   MOV DX, AX
   JMP f
p: MOV RDX, 0
   MOV DX, AX
f: MOV RAX, RDX
   POP RDX
   ```

5. *(Examen de seconde session, 2019)* Écrire des fragments de code assembleur x86-64 réalisant les opérations suivantes. (On supposera chaque fois que la pile est initialement correctement configurée.)

   (a) Permuter les deux valeurs de 64 bits situées au sommet de la pile.

   (b) Déterminer si le contenu du registre AL correspond au premier octet d'un caractère représenté sur 16 bits selon l'encodage UTF-8. À l'issue de cette opération, AH vaudra 1 dans le cas positif, et 0 sinon.

Chapitre 4

La programmation en assembleur

Ce chapitre aborde la programmation en langage d'assemblage. Ce langage n'est pas universel. En effet, comme cela a été discuté au chapitre précédent, le modèle mémoire d'un processeur, ses registres, son jeu d'instructions et les modes d'adressage qu'il supporte diffèrent généralement d'une architecture à une autre. Même pour une architecture donnée, il peut exister plusieurs syntaxes de langage d'assemblage, spécifiques aux outils utilisés. Enfin, les conventions employées pour mettre en œuvre le mécanisme d'appel de fonctions dépendent du système d'exploitation de l'ordinateur.

Dans ce cours, nous étudions l'architecture x86-64, qui est celle de la majorité des ordinateurs personnels actuels. Nous opterons pour le langage d'assemblage défini par l'outil *GNU Assembler*, dans sa variante syntaxique *Intel*. Les conventions d'appel de fonctions seront celles des système d'exploitation de type *UNIX* (dont font partie *Linux* et *macOS*).

Les exemples de programmes en assembleur proposés dans cette partie du cours peuvent être compilés et exécutés dans l'environnement suivant :

— un processeur d'architecture x86-64,

— le système d'exploitation Linux, dans sa version 64 bits.

— le compilateur GCC [1] *(the GNU Compiler Collection)*, incluant des outils de bas niveau tels que le *GNU Assembler*.

1. Ce compilateur peut être configuré de différentes façons. Dans certains environnements de développement, les exemples du cours nécessitent d'activer l'option `-no-pie` afin de pouvoir être compilés.

4.1 Le langage d'assemblage

Comme nous l'avons vu au chapitre 3, un programme écrit en langage d'assemblage est essentiellement composé d'instructions exécutables par le processeur, exprimées dans une syntaxe lisible. Dans un tel programme, on trouve également

— des définitions de *données* à placer en mémoire.

— des *directives* utilisées pour spécifier certaines options de compilation, ou implémenter des mécanismes particuliers.

4.1.1 Un premier programme

Un exemple de programme très simple est donné ci-dessous :

```
                .intel_syntax noprefix
                .text
                .global deep_thought
                .type deep_thought, @function
deep_thought:   MOV   EAX, 42
                RET
```

Examinons les éléments de ce programme :

— La directive `.intel_syntax noprefix` indique que ce programme est rédigé dans la variante syntaxique [2] *Intel* du langage d'assemblage x86-64. Pour cette variante, l'option `noprefix` permet d'écrire les noms de registres et les constantes numériques directement, sans devoir les précéder d'un symbole spécial.

— La directive `.text` signale le début d'un *segment de code*, ce qui signifie que la suite du programme contient des instructions exécutables qui seront placées dans la mémoire de programme.

— Les directives

```
.global deep_thought
.type deep_thought, @function
```

indiquent que l'étiquette `deep_thought` (qui sera définie dans la suite du programme comme étant son *point d'entrée*, c'est-à-dire l'adresse de sa première instruction) est un

2. Il en existe une autre appelée *AT&T*.

symbole *global*, ce qui signifie qu'il sera possible d'y faire référence depuis une autre partie du programme, et que ce symbole représente une fonction.

— L'instruction

```
deep_thought:   MOV    EAX, 42
```

est la première instruction exécutable du programme. L'étiquette `deep_thought` est définie comme étant égale à l'adresse de cette instruction dans la mémoire de programme.

Cette instruction a pour effet d'écrire la valeur constante 42 dans le registre de 32 bits EAX. Nous verrons à la section 4.2 que ce registre contient la valeur de retour d'une fonction lorsque son type correspond à un entier représenté sur 32 bits. Le programme implémente donc une fonction appelée `deep_thought`, qui retourne simplement [3] le nombre 42.

— Enfin, l'instruction `RET` termine l'exécution de la fonction.

4.1.2 Les mécanismes de compilation et d'exécution

Pour pouvoir exécuter le programme simple introduit à la section précédente, il faut le compiler, et écrire un programme de test qui invoque la fonction `deep_thought`. Voici un exemple de tel programme, rédigé en langage C :

```
#include <stdio.h>

extern int deep_thought(void);

int main()
{
  printf("%d\n", deep_thought());
}
```

Ce programme déclare une fonction `deep_thought` sans paramètres, retournant un entier. Cette fonction est externe, ce qui signifie que son implémentation se trouve dans une autre partie du programme. Après avoir invoqué la fonction `deep_thought`, l'entier que celle-ci retourne est affiché sur la console.

3. La pertinence d'un tel programme est expliquée dans *The Hitchhiker's Guide to the Galaxy*, par Douglas Adams.

Pour obtenir un programme exécutable, on peut placer le code assembleur de la fonction `deep_thought` (donné à la section 4.1.1) dans un fichier `dt.s`, et le programme C de test (que nous venons d'examiner) dans un fichier `test.c`. La commande

```
gcc -Wall -O3 -o test test.c dt.s
```

lance alors la compilation de ces deux fichiers source, et réalise l'édition de liens afin d'obtenir un code machine exécutable.

Notes : Cette commande utilise les options suivantes du compilateur :

— `-Wall` active l'ensemble des avertissements pouvant être produits par le compilateur.

— `-O3` sélectionne le mode d'optimisation le plus performant du compilateur C.

— `-o test` signale que le résultat de la compilation doit être placé dans un fichier exécutable appelé `test`. □

Sans surprise, l'exécution du programme compilé (en invoquant par exemple `./test`) affiche la valeur attendue (42) sur la console.

La chaîne de traitement mise en œuvre pour compiler et exécuter le programme est la suivante :

1. Le compilateur C traduit le code source `test.c` en code assembleur.
2. Le programme d'assemblage traduit en code objet le code assembleur contenu dans `dt.s`, ainsi que celui produit par l'étape précédente.
3. L'éditeur de lien combine trois fragments de code objet, de façon à obtenir un programme exécutable complet :
   — celui produit par la première étape, implémentant la fonction `main`,
   — celui produit par l'assemblage de `dt.s`, implémentant la fonction `deep_thought`.
   — l'implémentation de la fonction `printf` et d'autres fonctions auxiliaires nécessaires au programme, extraites de la *bibliothèque standard* C.

Notes :

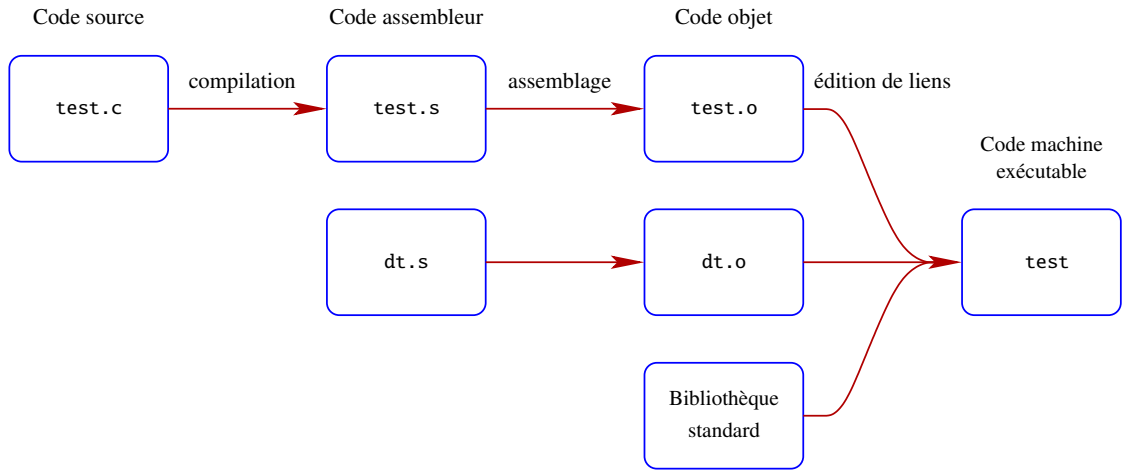
— Les fichiers `test.s`, `test.o` et `dt.o` figurant dans le diagramme suivant, contenant respectivement la traduction en assembleur de `test.c`, et la conversion en code objet de ce code assembleur ainsi que de celui de `dt.s`, ne sont par défaut pas sauvegardés par le compilateur GCC. Au besoin, ils peuvent être produits par les commandes suivantes :

— `gcc -S -masm=intel test.c`

— `gcc -c test.c`

— `gcc -c dt.s`

— Dans cet exemple, une partie de l'opération d'édition de liens est réalisée au moment où le fichier exécutable `test` est chargé en vue de son exécution. □



4.1.3 Les étiquettes

Comme on l'a vu dans l'exemple précédent, un programme assembleur peut définir des *étiquettes*, notamment pour représenter symboliquement l'adresse de certaines instructions. La définition d'une étiquette peut prendre deux formes :

— Une instruction de la forme

`<étiquette>: <instruction> <opérandes>`

indique qu'*<étiquette>* prend pour valeur l'adresse de l'instruction suivante dans la mémoire de programme. Un saut vers *<étiquette>* continuera donc l'exécution du programme à cette instruction.

Remarques :

— C'est le programme d'assemblage qui attribue une valeur numérique à l'étiquette au moment de la compilation du programme.

— Il est permis de faire référence à une telle étiquette avant sa définition, de façon à pouvoir effectuer des sauts vers la suite du programme.

— Nous verrons à la section suivante que l'utilisation d'étiquettes définies de cette façon est sujette à quelques restrictions. □

— La directive

`.equ` <*étiquette*>, <*valeur*>

définit l'étiquette <*étiquette*> et lui attribue la valeur <*valeur*>. Avec ce mécanisme, l'étiquette ne peut être utilisée que dans la suite du programme.

Le programme suivant donne un exemple d'utilisation des deux formes de définition d'étiquettes :

```
                .intel_syntax noprefix
                .text
                .global deep_thought2
                .type deep_thought2, @function
                .equ   answer, 42
end:            RET
deep_thought2:  MOV    EAX, answer
                JMP    end
```

4.1.4 Le segment de données

Dans un programme assembleur, en addition aux instructions à exécuter, il est également possible de spécifier l'organisation de la mémoire de données, ainsi que son contenu initial. Les directives suivantes peuvent être employées :

— La directive `.data` indique le début d'un segment de données, c'est-à-dire de la zone de la mémoire dans laquelle les données manipulées par le programme seront placées.

— Les directives `.byte`, `.word`, `.int` et `.quad` permettent de définir une ou plusieurs valeurs entières, séparées par des virgules, encodées respectivement sur 1, 2, 4 et 8 octets. Ces valeurs sont placées les unes après les autres, à partir de l'endroit courant de la mémoire de données.

— La directive `.fill` doit être suivie par trois nombres <*répétition*>, <*taille*> et <*valeur*>, séparés par des virgules, avec <*taille*> $\in \{1, 2, 4\}$. Elle a pour effet de remplir la mémoire avec <*répétition*> copies de la constante <*valeur*> encodée sur <*taille*> octets, à partir de l'adresse courante.

— La directive `.ascii`, suivie par une chaîne de caractère écrite entre guillemets, encode cette chaîne en ASCII et la place en mémoire à partir de l'adresse courante.

— La directive `.asciz` est similaire à la précédente, mais ajoute un octet nul de terminaison à la fin de la chaîne, selon la convention de représentation des chaînes de caractères du langage C.

Remarque : Comme cela a été discuté au chapitre 3, il ne faut pas oublier d'aligner les données mémorisées sur plus d'un octet. Cela peut se faire grâce à la directive `.balign`. Celle-ci doit être suivie d'un nombre <*taille*>. Elle a pour effet de déplacer l'adresse courante au prochain emplacement qui est un multiple entier de <*taille*>. Par exemple, le fragment de code

```
                .balign 4
taille_tableau: .int   0x1000
```

s'assure que l'entier de 32 bits pointé par l'étiquette `taille_tableau` est convenablement aligné. Il n'est pas nécessaire d'utiliser ce mécanisme au début du segment de données ; la directive `.data` aligne implicitement l'adresse courante à un multiple de 8 octets. □

Comme le montre l'exemple précédent, il est permis de définir des étiquettes faisant référence au contenu de la mémoire de données. Le plus souvent, ces étiquettes sont utilisées avec un adressage direct, ou comme déplacement d'un adressage indirect indexé.

Exemple :

Le programme suivant implémente un mécanisme de ticket pour une file d'attente. Afin que chaque appel à la fonction `ticket` retourne une valeur différente de celles générées par les appels précédents, cette fonction alloue un emplacement de mémoire (repéré par l'étiquette `nb_tickets`) servant à retenir le nombre de tickets déjà émis. Lorsque la fonction `ticket` est appelée, elle incrémente cette variable et en retourne la nouvelle valeur.

```
                .intel_syntax noprefix
                .data
nb_tickets:     .int   0
                .text
                .global ticket
                .type ticket, @function
ticket :        INC    dword ptr[nb_tickets]
                MOV    EAX, dword ptr[nb_tickets]
                RET
```

Cette fonction peut par exemple être testée grâce à ce programme C, qui crée et affiche un nombre illimité de tickets :

```
#include <stdio.h>

extern int ticket(void);

int main()
{
  for (;;)
    printf("%d\n", ticket());
}
```

□

Le mécanisme d'étiquettes présente cependant une complication. Pour une raison technique, les étiquettes qui représentent une adresse ne peuvent pas toujours être directement utilisées dans un adressage immédiat. Par exemple, si `x` est une étiquette pointant vers un emplacement du segment de données, alors l'instruction

```
MOV RAX, x
```

visant à recopier la valeur de `x` dans le registre RAX est invalide.

Note : Certains programmes d'assemblage tels que *GNU Assembler* remplacent automatiquement une telle construction par un adressage direct, c'est-à-dire

```
MOV RAX, qword ptr[x]
```

dans le cas de l'exemple précédent. Il s'agit d'un mécanisme particulièrement contre-intuitif, car la signification de l'instruction qui sera réellement produite diffère fortement de celle écrite par le programmeur. □

La raison pour laquelle certains adressages immédiats ne sont pas directement compatibles avec les étiquettes d'adresse est que l'emplacement exact d'une donnée ou d'une instruction en mémoire n'est pas toujours connu au moment de l'assemblage. C'est en effet l'éditeur de liens qui est chargé de déterminer la place en mémoire de tous les composants d'un programme lorsque toutes les parties de celui-ci sont combinées. Dans l'architecture x86-64, on génère fréquemment du code machine qui est *relocalisable (relocatable)*, ce qui signifie que son exécution ne dépend pas de l'endroit particulier où il est placé en mémoire. Un tel code relocalisable peut être obtenu en encodant les adresses de façon *relative* à la valeur courante du pointeur de programme. Par exemple, dans le fragment de code

```
boucle:         NOP
                NOP
                JMP    boucle
```

l'opérande de l'instruction `JMP` sera encodée de façon à indiquer un saut vers l'adresse RIP – 4, plutôt que vers l'endroit absolu où le début de la boucle sera placé en mémoire. Le déplacement relatif -4 correspond ici au nombre d'octets de code machine qui constituent le corps de la boucle, l'instruction `NOP` étant codée sur un octet et l'instruction `JMP` sur deux. Un mécanisme similaire est employé pour encoder un adressage direct vers des emplacements du segment de données.

L'utilisation de code relocalisable simplifie l'opération d'édition de liens, mais complique le calcul d'un pointeur absolu vers une donnée particulière. Pour obtenir un tel pointeur, il faut préfixer l'étiquette correspondante des mots-clés `offset flat:` dans le programme. Par exemple, au lieu d'écrire

```
MOV RAX, x   # Invalide!
```

où `x` est une étiquette pointant dans le segment de données ou vers une instruction, on écrira

```
MOV RAX, offset flat:x
```

Exemple : Le programme suivant retourne un pointeur vers un tableau qui contient les carrés de tous les nombres entiers non signés encodables sur 16 bits :

```
                .intel_syntax noprefix
                .data
tableau:        .fill  0x10000, 4, 0
                .text
                .global squares
                .type squares, @function
squares:        MOV    RDI, 0
boucle:         MOV    AX, DI
                MUL    AX
                MOV    word ptr[4*RDI + tableau], AX
                MOV    word ptr[4*RDI + (tableau + 2)], DX
                INC    DI
                JNZ    boucle
                MOV    RAX, offset flat:tableau
                RET
```

Un programme de test permettant d'essayer cette fonction est donné ci-dessous :

```
#include <stdio.h>

extern unsigned *squares(void);

int main()
{
  unsigned i, *s;

  s = squares();

  for (i = 0; i < 0x10000; i++)
    printf("%u: %u\n", i, s[i]);
}
```

L'implémentation de la fonction `squares` alloue un tableau dans le segment de données, contenant 2^{16} entiers encodés sur 4 octets initialisés à 0. La fonction utilise le registre RDI comme index pour accéder à ce tableau : Ce registre est initialisé à 0, et ses 16 bits de poids faible (représentés par le registre fictif DI) sont incrémentés à chaque itération, jusqu'à ce qu'ils redeviennent égaux à 0. A chaque itération, le produit de DI par lui-même est calculé à l'aide de l'instruction `MUL`. Les 16 bits de poids faible et de poids fort du résultat sont ensuite placés aux endroits appropriés du tableau. A l'issue de l'exécution de la boucle, la fonction retourne un pointeur vers le tableau. Comme nous le verrons au chapitre suivant, RAX est en effet le registre qu'une fonction utilise pour retourner un pointeur.

Enfin, remarquons que dans cette fonction, l'étiquette `boucle` est utilisée sans précaution particulière comme opérande de l'instruction `JNZ`, avec un adressage immédiat. Techniquement, cette opérande sera encodée dans le code machine de façon relative à la valeur courante du compteur de programme. L'étiquette `tableau` est également employée comme déplacement pour deux opérandes utilisant l'adressage indirect indexé, dont l'encodage sera également relatif. En revanche, l'avant-dernière instruction du programme nécessite de connaître l'adresse absolue du tableau en mémoire, ce qui oblige à utiliser les mots clés `offset flat`. □

4.2 L'appel d'une fonction

Dans les langages de programmation impératifs tels que C, une *fonction* est un ensemble d'instructions dont l'exécution peut être invoquée (par un *appel de fonction*), à partir de n'importe quel endroit du programme. Une fonction peut définir des *paramètres*, qui sont des variables dont la valeur est attribuée lors de chaque appel de cette fonction, et peut également renvoyer une *valeur de retour* vers l'appelant à la fin de son exécution. Un fonction peut aussi allouer des

variables locales, qui correspondent à des emplacements de mémoire temporairement alloués au cours de son exécution.

4.2.1 La convention d'appel

L'implémentation de fonctions en code machine exploite bien entendu le mécanisme de gestion de pile introduit au chapitre 3 : Lors de l'appel d'une fonction (par l'instruction `CALL`), l'adresse de retour (c'est-à-dire, celle de l'instruction qui suit cet appel) est automatiquement empilée, afin d'être récupérée (par l'instruction `RET`) lorsque cette fonction termine son exécution. Ce mécanisme n'est cependant pas suffisant ; il est aussi nécessaire de définir un protocole régissant le transfert des arguments [4] et de la valeur de retour de la fonction, ainsi que le procédé employé par celle-ci pour allouer des variables temporaires.

Ce protocole, appelé *convention d'appel (calling convention)* diffère selon le système d'exploitation utilisé. Dans ce cours, nous allons étudier celui qui est mis en œuvre dans les systèmes d'exploitation de type *UNIX* (incluant Linux et macOS), pour l'architecture x86-64. Cette étude sera limitée au cas de paramètres et de valeurs de retour de type entier ou pointeur. Les principes de cette convention d'appel sont les suivants :

— Les six premiers arguments éventuels de l'appel sont placés dans les registres respectifs RDI, RSI, RDX, RCX, R8 et R9. S'ils sont encodés sur 8, 16 ou 32 bits, ils occuperont les parties de poids faible correspondantes de ces registres.

— Si l'appel possède des arguments supplémentaires, alors ceux-ci sont placés sur la pile, en tant que valeurs représentées sur 64 bits. Ces arguments sont empilés dans l'ordre inverse où ils apparaissent dans l'appel de la fonction. Ils sont empilés avant que l'instruction `CALL` n'empile l'adresse de retour.

— La valeur de retour d'une fonction est placée dans le registre RAX, ou dans une partie de poids faible de ce registre si cette valeur est encodée sur 8, 16 ou 32 bits.

— Si une fonction modifie les registres RBX, RBP, R12, R13, R14 ou R15, alors elle doit se charger d'en sauvegarder la valeur, et de la restaurer avant de terminer son exécution.

— La fonction doit maintenir le pointeur de pile RSP à une adresse multiple de 16.

4.2.2 La structure de pile

Les données relatives à un appel de fonction qui sont placées sur la pile sont organisées selon une *structure de pile (stack frame)*. Chaque appel de fonction place une telle structure sur la pile, et chaque retour de fonction retire de la pile la structure correspondant à l'appel qui se termine.

4. On appelle *argument* la valeur d'un paramètre d'une fonction.

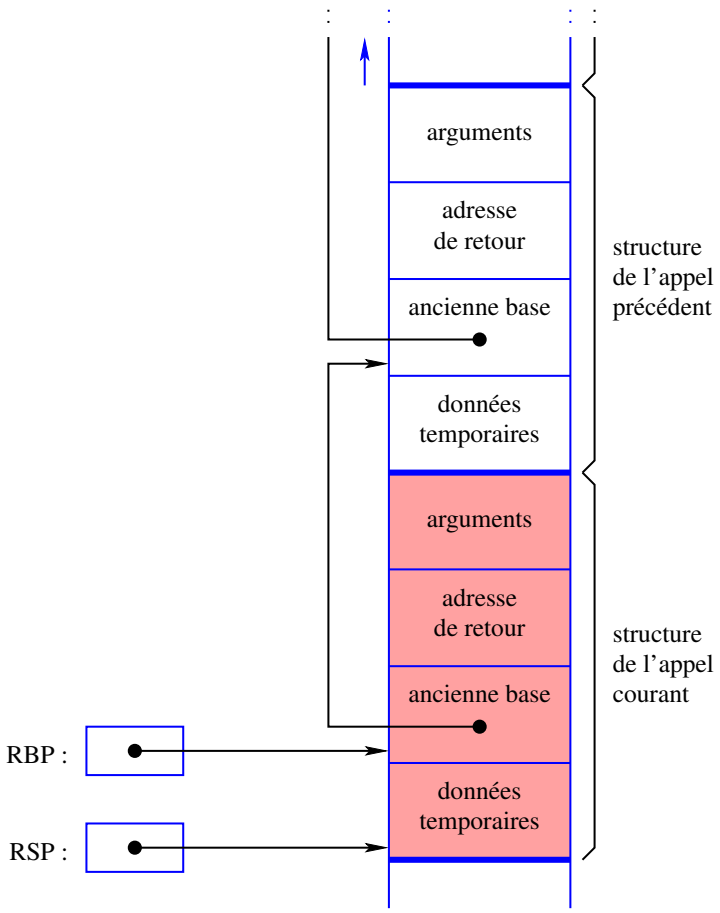
Le contenu d'une structure de pile est indexé à partir de sa *base*, qui est représentée par le contenu du registre [5] RBP. Cette structure contient les éléments suivants, par ordre décroissant d'adresse (c'est-à-dire, dans l'ordre où ils sont empilés) :

— Les arguments de la fonction appelée, à partir du septième. Il sont représentés sur 64 bits et empilés dans l'ordre inverse de leur position.

— L'adresse de retour de la fonction.

— Un pointeur vers la base de la structure de pile précédente. Ce pointeur est situé à la base de la structure courante, et a pour but de permettre la libération de cette structure à la fin de l'appel de fonction en cours.

— Les données temporaires allouées par la fonction, telles que ses variables locales.



5. Il s'agit de l'extension à 64 bits du registre BP provenant de l'architecture x86, dont le nom signifie *Base Pointer*.

Note : L'intérêt de faire pointer le registre RBP vers la base de la structure est que cela permet d'accéder facilement aux éléments de celle-ci. Par exemple, les arguments de la fonction peuvent être consultés à l'aide des adressages indirects indexés `qword ptr [RBP + 16]`, `qword ptr [RBP + 24]`, ... Les données temporaires allouées par la fonction correspondent quant à elles à un déplacement négatif. □

4.3 Exemples

4.3.1 Calcul récursif d'une factorielle

Cette fonction C permet de calculer la factorielle d'un nombre, sur 64 bits :

```
unsigned long factorielle(unsigned n)
{
  if (n <= 1)
    return 1;
  else
    return n * factorielle(n - 1);
}
```

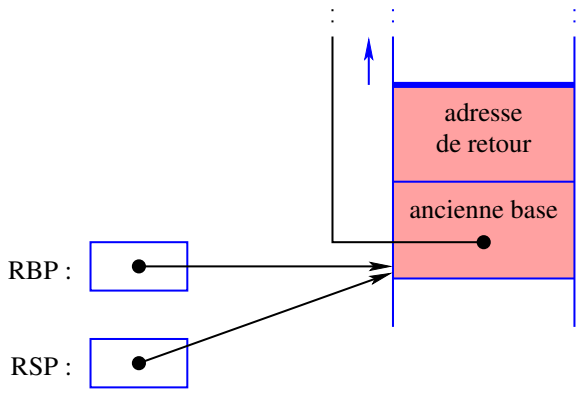
Cette fonction peut se traduire en assembleur de la façon suivante :

```
                .intel_syntax noprefix
                .text
                .global factorielle
                .type factorielle, @function
factorielle:    PUSH   RBP
                MOV    RBP, RSP
                CMP    EDI, 1
                JBE    retour_un
                PUSH   RDI
                DEC    RDI
                SUB    RSP, 8
                CALL   factorielle
                ADD    RSP, 8
                POP    RDI
                MUL    RDI
                JMP    retour
retour_un:      MOV    RAX, 1
retour:         POP    RBP
                RET
```

Les deux premières instructions

```
                PUSH   RBP
                MOV    RBP, RSP
```

de cette fonction créent la structure de pile, en empilant la base de la structure précédente et en situant la base de la structure courante.

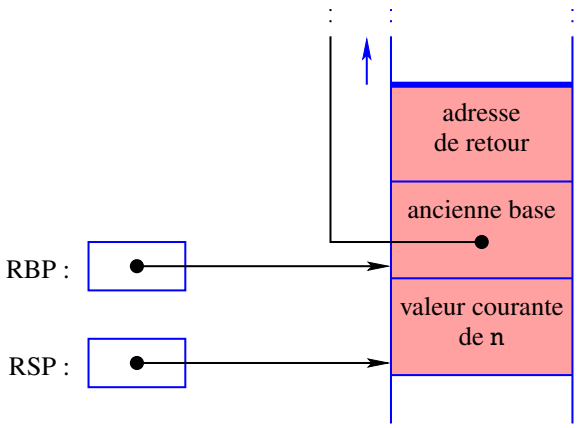


Ensuite, les instructions

```
                CMP    EDI, 1
                JBE    retour_un
                ...
retour_un:      MOV    RAX, 1
                POP    RBP
                RET
```

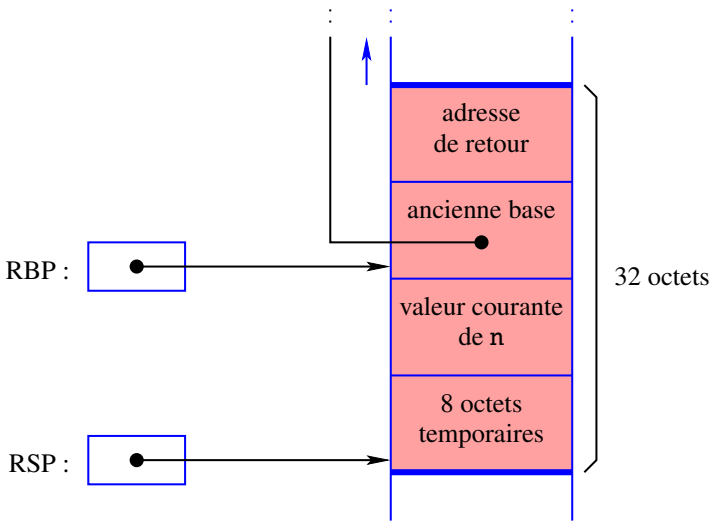
traitent le cas de base de la récursion : Si l'argument `n` de la fonction (fourni dans les 32 bits de poids faible de RDI, c'est-à-dire EDI) est inférieur ou égal à 1, la fonction retourne 1. Remarquons que l'instruction `POP RBP` a pour effet de supprimer la structure de pile associée à l'appel courant.

Si `n` est en revanche supérieur à 1, il faut effectuer un appel à la fonction `factorielle`, en lui passant l'argument `n` − 1. La valeur de `n` doit cependant être préservée pour l'appel courant, car elle sera utilisée dans la suite du calcul. Grâce à l'instruction `PUSH RDI`, cette valeur est placée dans un emplacement temporaire alloué dans la structure de pile courante :



Pour l'appel de `factorielle`, il faut placer dans RDI la valeur `n` − 1, ce qui est l'objet de l'instruction `DEC RDI`.

A ce stade, il ne serait pas correct d'appeler directement `factorielle`, car cela violerait une des contraintes de la convention d'appel. En effet, cette convention impose que le registre de pile RSP possède une valeur multiple de 16 au moment de l'appel d'une fonction. Cette propriété n'est pas respectée, car la structure de pile qui a été construite (correspondant à la zone colorée de la figure précédente) occupe 24 octets en mémoire. L'instruction `SUB RSP, 8` vise à corriger ce problème, en allouant 8 octets supplémentaires sur la pile. La configuration obtenue est alors la suivante :



Les instructions suivantes

```
CALL   factorielle
ADD    RSP, 8
POP    RDI
```

appellent la fonction `factorielle`, et récupèrent dans RDI la valeur courante de `n` qui avait été sauvegardée dans l'espace temporaire (cet espace est par la même occasion supprimé). La valeur retournée par `factorielle(n - 1)` est quant à elle disponible dans RAX.

L'instruction `MUL RDI` calcule alors le produit de ces deux registres. Les 64 bits de poids faible du résultat, qui forment la valeur à retourner, sont alors disponibles dans RAX. L'exécution de la fonction peut alors se terminer.

Le programme C suivant permet de tester cette implémentation de la fonction `factorielle`.

```
#include <stdio.h>

extern unsigned long factorielle(unsigned);

int main()
{
  unsigned i;

  for (i = 0; i < 20; i++)
    printf("%u: %lu\n", i, factorielle(i));
}
```

4.3.2 Hello, world

Tous les programmes en assembleur que nous avons écrits jusqu'à présent nécessitaient un programme C annexe pour lancer leur exécution. Pour obtenir un programme autonome, c'est-à-dire capable de s'exécuter directement, il suffit qu'il implémente la fonction `main`, qui constituera son point d'entrée. Par exemple, le programme suivant affiche une chaîne de caractères sur la console, en utilisant la fonction `printf` de la bibliothèque standard C :

```
                .intel_syntax noprefix
                .data
msg:            .asciz "Hello, world!\n"
                .text
                .global main
                .type main, @function
main:           PUSH   RBP
                MOV    RBP, RSP
                MOV    RDI, offset flat:msg
                CALL   printf
                MOV    EAX, 0
                POP    RBP
                RET
```

Notes :

— Dans ce programme, l'étiquette `msg` pointe vers un emplacement du segment de données, et sa valeur est utilisée comme argument de la fonction `printf`. Il est donc nécessaire d'obtenir une représentation absolue du pointeur correspondant, ce qui oblige à écrire le préfixe `offset flat`.

— La fonction `main` est supposée retourner un entier, représentant le code de sortie du programme. Dans le programme, l'instruction `EAX, 0` retourne un code égal à 0, signalant que l'exécution s'est déroulée sans erreur.

— En supposant que son code source soit placé dans un fichier appelé `hw.s`, ce programme peut être compilé grâce à la commande

```
gcc -o hw hw.s
```

Le code machine est alors placé dans le fichier exécutable `hw`. □

4.3.3 Conversion en minuscules

Le programme suivant affiche sur des lignes séparées les arguments qui lui sont fournis en ligne de commande, après y avoir converti les lettres ASCII majuscules en minuscules :

```
                .intel_syntax noprefix
                .text
                .global main
                .type main, @function
main:           PUSH   RBP
                MOV    RBP, RSP
boucle:         DEC    RDI
                JZ     fin
                ADD    RSI, 8
                PUSH   RDI
                PUSH   RSI
                MOV    RDI, qword ptr[RSI]
                CALL   conversion
                MOV    RAX, qword ptr[RBP - 16]
                MOV    RDI, qword ptr[RAX]
                CALL   puts
                POP    RSI
                POP    RDI
                JMP    boucle
fin:            MOV    EAX, 0
                POP    RBP
                RET
conversion:     PUSH   RBP
                MOV    RBP, RSP
boucle2:        MOV    AL, byte ptr[RDI]
                CMP    AL, 0
                JE     fin
                INC    RDI
                CMP    AL, 'A'
                JB     boucle2
                CMP    AL, 'Z'
                JA     boucle2
                ADD    AL, 0x20
                MOV    byte ptr[RDI - 1], AL
                JMP    boucle2
```

Ce programme fonctionne de la façon suivante. Lorsqu'elle est appelée, la fonction `main` reçoit deux valeurs :

— le nombre `argc` d'arguments fournis au programme lors de son exécution, incluant le nom du programme qui constitue le premier de ces arguments.

— un tableau `argv` contenant `argc` pointeurs vers des chaînes de caractères représentant les arguments du programme.

Suivant la convention d'appel, la fonction `main` reçoit la valeur d'`argc` dans le registre RDI, et celle d'`argv` dans le registre RSI. Elle rentre alors dans une boucle visant à traiter un argument à chaque itération. Cette boucle commence par décrémenter RDI, de façon à ce que ce registre contienne le nombre d'arguments restant à traiter. Si ce nombre est nul, la fonction se termine.

Ensuite, RSI est incrémenté de 8 unités, afin de le faire pointer vers l'entrée du tableau `argv` qui pointe vers la chaîne à convertir et à afficher au cours de l'itération courante. Ces deux opérations seront respectivement effectuées par la fonction `conversion`, définie dans la suite du programme, et par la fonction `puts` de la bibliothèque standard C. Ces deux fonctions pouvant potentiellement modifier les registres RDI et RSI, ceux-ci sont sauvegardés sur la pile avant d'invoquer ces fonctions.

La fonction `conversion` admet un seul paramètre, correspondant à l'adresse de la chaîne de caractères à convertir. Cette adresse est lue depuis l'entrée du tableau pointée par RSI, et recopiée dans le registre RDI qui contient l'argument de la fonction à appeler.

Remarquons qu'à ce stade, la structure de pile correspondant à l'appel de `main` comprend 32 octets : 8 pour l'adresse de retour, 8 pour le pointeur vers l'ancienne base, et 16 pour la sauvegarde des registres RDI et RSI. La contrainte d'obtenir un registre de pile égal à un multiple de 16 est donc bien préservée.

Après l'appel à `conversion`, il reste à afficher la chaîne de caractères convertie. Cette opération peut s'effectuer par un appel à la fonction `puts` de la bibliothèque standard C, qui prend comme argument un pointeur vers la chaîne à afficher. Pour obtenir ce pointeur, le programme récupère d'abord dans RAX la valeur sauvegardée du registre RSI grâce à l'adressage indirect indexé `qword ptr[RBP - 16]`. (En effet, il s'agit de la deuxième valeur de 8 bits empilée à la suite de l'ancienne base.) Ensuite, il lit la valeur pointée par RAX et la transfère dans le registre RDI, qui contiendra l'argument de l'appel à `puts`. Après cet appel, il reste simplement à récupérer les valeurs sauvegardées de RSI et de RDI, et à passer à l'itération suivante.

L'implémentation de la fonction `conversion` n'appelle pas de commentaire particulier. Cette fonction examine l'ensemble des caractères qui composent la chaîne reçue en argument, jusqu'à l'octet nul qui en indique la fin. Pour chaque caractère traité, la fonction détermine s'il s'agit d'une lettre majuscule, et la convertit le cas échéant en minuscules.

Note : Les constantes `'A'` et `'Z'` qui apparaissent dans les instructions de comparaison seront remplacées par le code ASCII de ces symboles par le programme d'assemblage. □

4.4 Exercices

1. Pour chacun des exercices de la section 3.5.2, traduire votre solution en un programme assembleur x86-64 complet, accompagné si nécessaire d'un programme C permettant de le tester.

2. Implémenter en assembleur x86-64 une fonction `produit_scalaire(v1, v2, n)` capable de calculer le produit scalaire de deux vecteurs `v1` et `v2`. Chacun de ces deux paramètres contient un pointeur vers un tableau d'entiers signés représentés sur 8 bits. Le paramètre `n` fournit le nombre de valeurs contenues dans chacun de ces tableaux. Le résultat de l'opération doit prendre la forme d'un entier signé représenté sur 64 bits.

3. La suite de Fibonacci F_0, F_1, F_2, ... est définie ainsi :

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_2 &= F_0 + F_1 \\ F_3 &= F_1 + F_2 \\ F_4 &= F_2 + F_3 \\ &\vdots \quad\quad \vdots \\ F_{i+2} &= F_i + F_{i+1} \\ &\vdots \quad\quad \vdots \end{aligned}$$

   Les premiers termes de cette suite sont donc 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

   Implémenter en assembleur x86-64 une fonction `fibonacci(n)` calculant F_n à partir de `n`. Par exemple, `fibonacci(10)` doit retourner 55.

   Le résultat sera représenté sur 32 bits. En cas de dépassement arithmétique, la fonction doit retourner ce résultat modulo 2^{32}.

4. Un palindrome est un mot qui se lit de façon identique de gauche à droite et de droite à gauche. Par exemple, le mot "RADAR" est un palindrome.

   Implémenter en assembleur x86-64 une fonction `est_palindrome(s)` qui retourne 1 si la chaîne de caractères pointée par `s` est un palindrome, et 0 sinon. La chaîne `s` est encodée en Iso latin1, et son dernier caractère est suivi par un octet nul.

5. (a) Implémenter en assembleur x86-64 une fonction `division(a, b)` capable de calculer, en effectuant des soustractions, la division entière du nombre `a` par le nombre `b`. Les nombres `a` et `b` sont non signés et représentés sur 32 bits. On peut supposer que `b` est non nul.

(b) En exploitant la routine obtenue au point précédent, implémenter en assembleur x86-64 une fonction `factoriser(x)` capable de factoriser un nombre entier `x` non signé représenté sur 32 bits. Cette fonction doit invoquer la fonction `printf` de la bibliothèque standard C, de façon à afficher chaque facteur de `x` sur une ligne séparée de la console.

6. *(Examen de première session, 2018)*
On souhaite programmer une fonction `prefixe_commun(t1, t2, n)` chargée de déterminer la longueur du préfixe commun entre les deux tableaux d'octets pointés par `t1` et `t2`, tous deux de taille `n` $\in [0, 2^{32} - 1]$. En d'autres termes, cette fonction doit déterminer la plus grande valeur k telle que `t1`$[0] =$ `t2`$[0]$, `t1`$[1] =$ `t2`$[1], \ldots,$ `t1`$[k-1] =$ `t2`$[k-1]$, et retourner cette valeur. Dans le cas où `t1`$[0] \neq$ `t2`$[0]$, la fonction doit retourner 0.

   (a) Écrire, en pseudocode ou en langage C (au choix), un algorithme permettant de résoudre ce problème.

   (b) Traduire cet algorithme en assembleur x86-64, en veillant à respecter la convention d'appel de fonctions des systèmes *Unix*.

7. *(Examen de seconde session, 2018)*
On souhaite programmer une fonction `facteur_deux(n)` chargée de calculer le plus grand entier k tel que 2^k divise n, où n est un nombre de 64 bits supposé strictement positif. Par exemple, `facteur_deux(96)` doit retourner 5, car $96 = 2^5\,3$. Lorsque `n` est impair, `facteur_deux(n)` doit retourner 0.

   (a) Écrire, en pseudocode ou en langage C (au choix), un algorithme permettant de résoudre ce problème.

   Suggestion : En exploitant les instructions logiques, compter le nombre de bits nuls situés à la fin de la représentation binaire de n.

   (b) Traduire cet algorithme en assembleur x86-64, en veillant à respecter la convention d'appel de fonctions des systèmes *Unix*.

8. *(Examen de première session, 2019)* Dans le cadre du développement d'une application de calcul statistique, on souhaite programmer une fonction `histogramme` capable de calculer l'histogramme d'un tableau d'octets donné. Cette fonction prend pour entrée

   — l'adresse d'un tableau d'octets `ts`,
   — la taille `n` de ce tableau, exprimée sur 64 bits,
   — l'adresse d'un tableau `th` contenant 256 entiers de 64 bits, non initialisés.

   À l'issue de l'exécution de cette fonction, chaque case de `th` doit contenir le nombre d'octets de `ts` égaux à son indice. Par exemple, la cinquième case `th[4]` de `th` contiendra le nombre d'octets de `ts` égaux à 4. La fonction ne retourne rien.

   — Écrire, en pseudocode ou en langage C (au choix), un algorithme permettant de résoudre ce problème.
   — Traduire cet algorithme en assembleur x86-64, en veillant à respecter la convention d'appel de fonctions des systèmes *Unix*.

9. *(Examen de seconde session, 2019)*
   On souhaite programmer une fonction `count_ok(str)` comptant le nombre d'occurrences de la chaîne de caractères "ok" dans une chaîne `str`. Cette dernière contient des caractères encodés en ASCII, et est terminée par un zéro.

   (a) Écrire, en pseudocode ou en langage C (au choix), un algorithme permettant de résoudre ce problème.

   (b) Traduire cet algorithme en assembleur x86-64, en veillant à respecter la convention d'appel de fonctions des systèmes *Unix*.