Architecture des Ordinateurs

Chapitre 2 : Architecture générale des ordinateurs

Jacques Supcik, Daniel Gachet, Luca Haab

2021-10-20

Table des matières

2 Architecture générale des ordinateurs

Si les premiers outils de calculs remontent à des milliers d'années, l'histoire des ordinateurs débute vers le milieu du 20e siècle. En effet, l'architecture générale des ordinateurs modernes remonte à ces années-là. Naturellement, depuis la mise sur le marché, en 1971, du premier processeur sur un seul et unique circuit intégré, le processeur Intel 4004 (figure 2.1), l'évolution du matériel et l'intégration toujours plus dense des transistors offrent aujourd'hui des processeurs avec une puissance de calcul inimaginable au siècle passé.

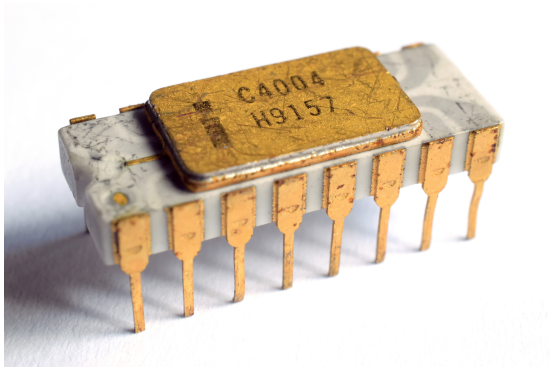


FIGURE 2.1 – Intel C4004 [1]

2.1 Ordinateur

Lorsque l'on parle d'un ordinateur, on peut se poser la question des problèmes que l'on souhaite résoudre avec cette machine. Une réponse vient assez rapidement à l'esprit, le traitement d'information. Ce traitement a pour objectif la manipulation de données acquises avec des capteurs par un logiciel afin de les restituer sous une autre forme à l'aide d'actionneurs ou d'actuateurs (figure 2.2). Par essence un ordinateur n'a de sens, qui l'on dispose d'une source d'information et d'un consommateur de cette information et de pouvoir adapter le traitement de cette information aux besoins du consommateur.

1. Source : https ://upload.wikimedia.org/wikipedia/commons/5/55/Intel_C4004.jpg

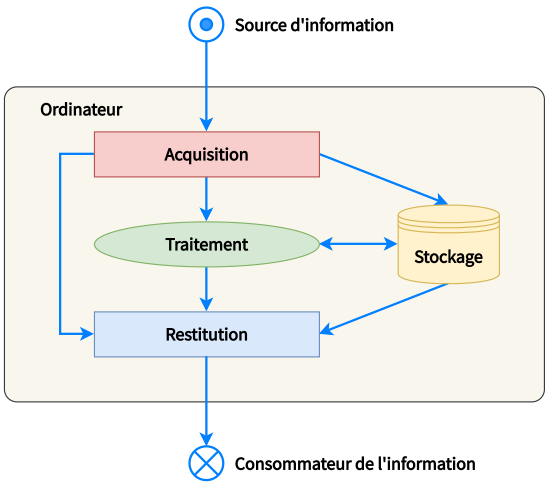


FIGURE 2.2 – Architecture générale d'un ordinateur

Tous les ordinateurs disposent d'une unité de stockage. Elle sert à la sauvegarde des instructions du programme ainsi que des données utiles au traitement de l'information.

2.2 Microprocesseur et microcontrôleur

Sur les systèmes à microprocesseurs (µP) ou microcontrôleurs (µC) (figure 2.3), l'unité centrale de traitement (CPU - *Central Processing Unit*) se charge du traitement de l'information. Pour acquérir les données à traiter et pour restituer l'information, résultat des traitements effectués, le CPU interagit avec des contrôleurs de périphériques d'entrées/sorties (I/O - *Input/Output*). Des interfaces physiques permettent d'adapter le niveau des signaux internes du µP/µC (TTL - *Transistor - Transistor Level*) aux signaux physiques des systèmes externes.

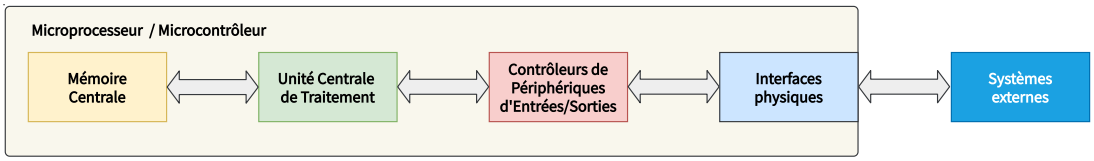


FIGURE 2.3 – Architecture générale des systèmes à µP/µC

La mémoire, souvent appelée *mémoire centrale*, permet à l'unité centrale de traitement de stocker les données et informations ainsi que le programme de traitement. Les µP/µC modernes utilisent généralement de la mémoire Flash pour un stockage persistant, idéal pour le programme lequel ne doit pas être perdu si le système est mis hors tension, tandis qu'ils mettent en oeuvre de la mémoire RAM pour les données et les informations volatiles.

Contrairement au microprocesseur (µP) qui ne dispose que de l'unité centrale de traitement, le microcontrôleur (µC) est un µP qui intègre sur la même puce la mémoire et les périphériques d'entrées/sorties.

2.3 Unité centrale de traitement

Pour traiter l'information, l'unité centrale de traitement (CPU) met en oeuvre 2 composants principaux, l'unité de contrôle (CU - *Control Unit*) et l'unité arithmétique et logique (ALU - *Arithmetic and Logic Unit*) avec sa banque de registres (mémoire interne du CPU) (figure 2.4). Le bus microprocesseur (*µP-Bus*), constitué d'un bus de contrôle, d'un bus d'adresses et d'un bus de données, permet au CPU de dialoguer avec les contrôleurs de périphériques d'entrées/sorties et la mémoire centrale.

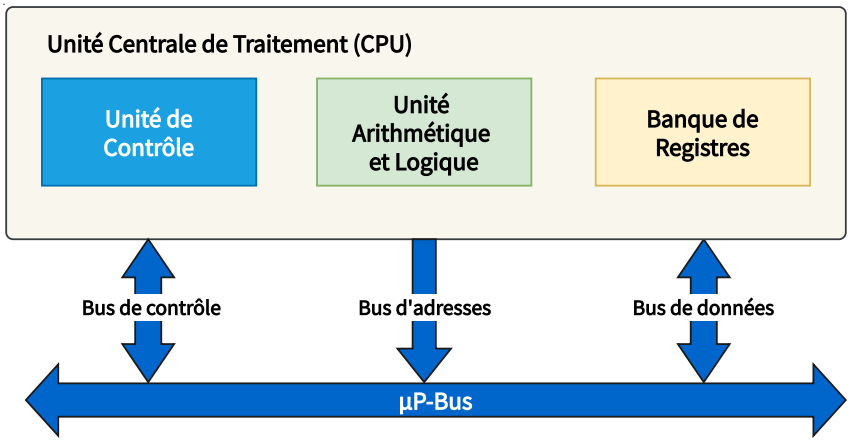


FIGURE 2.4 – Architecture générale de l'unité centrale de traitement

Le bus de données permet de transférer les données entre l'unité centrale de traitement et la mémoire centrale ou les contrôleurs de périphériques. Ces transferts peuvent s'effectuer dans les deux sens (du CPU vers la mémoire ou les I/O, de la mémoire ou des I/O vers le CPU). Le bus d'adresses permet d'identifier les données à transférer dans la mémoire ou les contrôleurs de périphériques. Le bus de contrôle permet de gérer les transferts entre le CPU et la mémoire et les contrôleurs de périphériques (sens du transfert, taille des données, etc.).

L'unité de contrôle se charge d'exécuter les instructions du programme de traitement (figure 2.5). L'unité arithmétique et logique (ALU), lui sert de "toolbox" fournissant toutes les opérations arithmiques et logiques nécessaires au traitement des données. Les registres (mémoire interne du CPU) servent au stockage intermédiaire des données nécessaires au traitement de l'information. Ils sont indispensables à l'ALU afin qu'elle puisse exécuter les opérations requises par le programme.

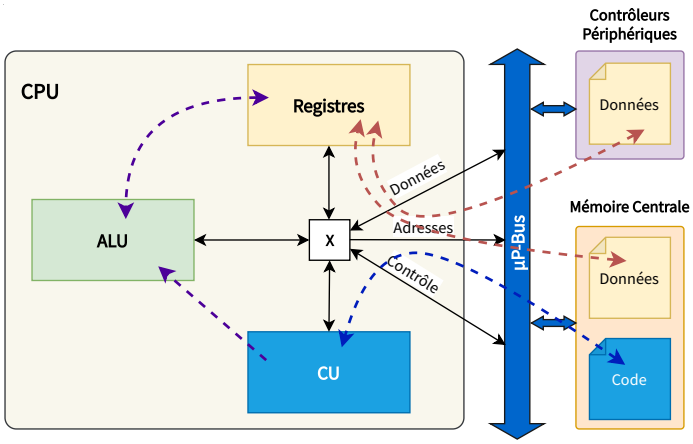


FIGURE 2.5 – Unité centrale de traitement et la mémoire

2.4 Principe de traitement de l'information

Le traitement de l'information s'effectue en 3 cycles principaux (figure 2.6), soit la lecture (*fetch*), le décodage (*decode*) et l'exécution (*execute*).

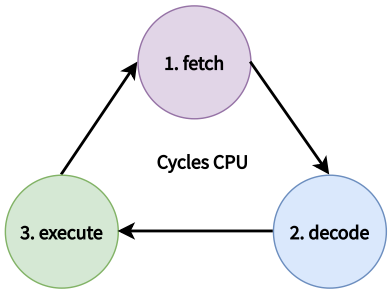


FIGURE 2.6 – Cycles du CPU pour le traitement de l'information

Durant le premier cycle (*fetch*), le CPU lit l'instruction de la mémoire centrale pour la placer dans l'unité de contrôle (figure 2.7).

Une fois l'instruction lue, le CPU effectue dans un deuxième cycle (*decode*) le décodage de l'instruction. Lors du troisième cycle (*execute*), le CPU exécute l'instruction en effectuant l'opération requise par l'instruction, par exemple un transfert (écriture ou lecture) d'une donnée entre un registre du CPU et la mémoire centrale ou un contrôleur de périphérique (figure 2.8) ou une opération arithmétique ou logique sur une donnée stockée dans l'un de ses registres.

Il paraît assez évident que chacun de ces cycles nécessite plusieurs cycles d'horloge CPU pour effectuer toutes les opérations nécessaires à l'accomplissement de la tâche lié au cycle. Sur les processeurs de première génération, le CPU effectue séquentiellement chacun de ces 3 cycles principaux (figure

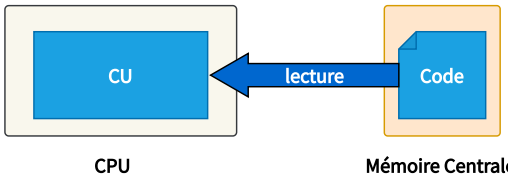


FIGURE 2.7 – Fetch - Chargement de l'instruction

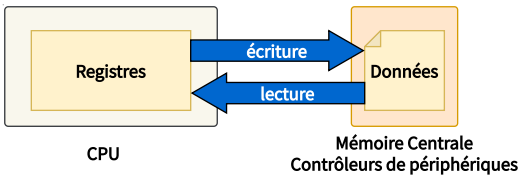


FIGURE 2.8 – Execute - Exécution de l'instruction (transfert de données)

2.9). Il est dès lors très simple de calculer de temps nécessaire à l'exécution du programme. Pour cela il suffit de sommer le nombre de cycles d'horloge CPU de chaque instruction du programme.

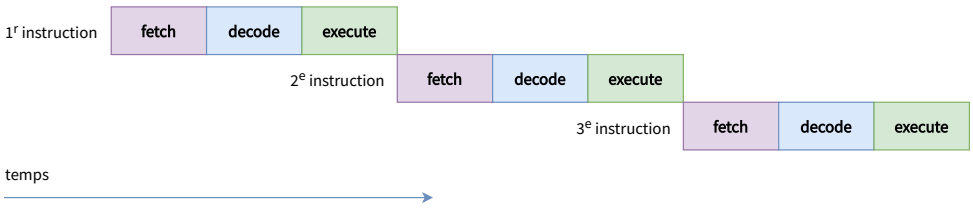


FIGURE 2.9 – Exécution séquentielle

Sur des processeurs de dernière génération, le déroulement de ces 3 cycles principaux s'effectue en parallèle (figure 2.10). Ce parallélisme permet de réduire drastiquement le temps d'exécution d'un programme. Il est connu sous le nom de *pipeline* (*instruction pipelining*).

2.5 Taxonomie de Flynn

Avec l'évolution des ordinateurs, plusieurs types d'architectures ont été imaginés. En 1966, Michael J. Flynn (docteur en génie électrique) propose une classification des architectures des ordinateurs en 4 catégories. Cette classification est connue sous le nom de *Taxonomie de Flynn*. Elle se base sur le flux des données et du flux des instructions entre les composants externes de l'ordinateur (mémoire centrale et contrôleurs de périphériques d'entrées/sorties) et les composants internes de l'unité centrale de traitement (d'un côté l'unité de contrôle et de l'autre l'unité arithmétique et logique et la banque de registres).

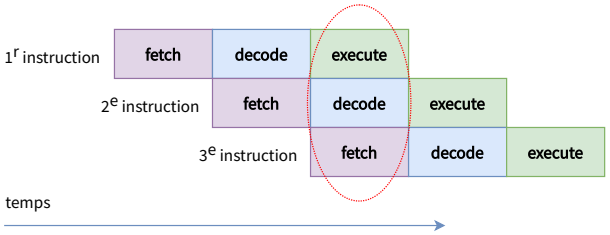


FIGURE 2.10 – Exécution parallèle

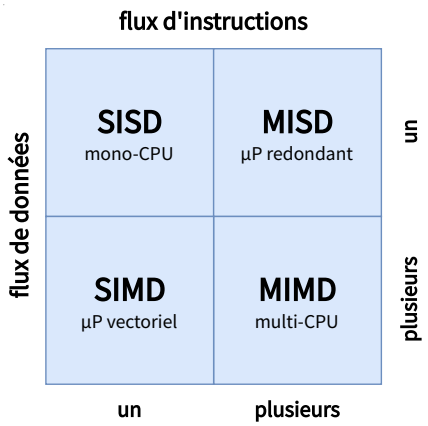


FIGURE 2.11 – Taxonomie de Flynn

La classification de Flynn distingue 4 architectures ou type de classes (figure 2.11) :

- Architecture SISD : Single Instruction, Single Data (instruction unique, donnée unique)
- Architecture SIMD : Single Instruction, Multiple Data (instruction unique, données multiples)
- Architecture MISD : Multiple Instruction, Single Data (instructions multiples, donnée unique)
- Architecture MIMD : Multiple Instruction, Multiple Data (instructions multiples, données multiples)

2.5.1 Architecture SISD

Dans l'architecture SISD, il n'existe aucun parallélisme, ni pour le flux d'instructions ni pour le flux de données. Les µP de ce type sont des machines mono-CPU. Ils exécutent séquentiellement une seule instruction à la fois traitant une seule donnée.

La vitesse de traitement de l'information dans l'architecture SISD est fortement dépendante de la vitesse à laquelle le CPU peut transférer l'information stockée dans la mémoire externe (mémoire centrale ou périphérique) dans la mémoire interne (banque de registres pour les données ou unité de contrôle pour les instructions).

L'architecture SISD est à la base de la plupart des µP modernes. Il existe 2 architectures fondamentales, soit :

- Architecture Harvard
- Architecture von Neumann

Architecture Harvard

Dans l'architecture Harvard, élaborée en 1944 à l'Université Harvard, le programme et les données sont stockés dans deux mémoires distinctes et physiquement séparées (figure 2.12).

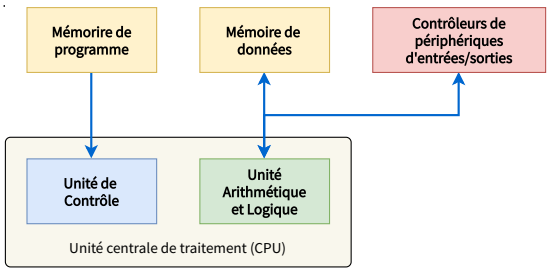


FIGURE 2.12 – Architecture Harvard

Chacune de ces mémoires dispose de son propre bus d'accès. Cette architecture autorise le transfert simultané des données et des instructions, permettant ainsi d'augmenter les performances du µP et

de réduire le temps de traitement de l'information. Le coût de ce gain en performance est au détriment d'une complexité accrue de sa structure.

Architecture von Neumann

L'architecture von Neumann, conçue en 1945 et baptisée du nom de son concepteur, stocke le programme et les données dans la mémoire centrale (figure 2.13).

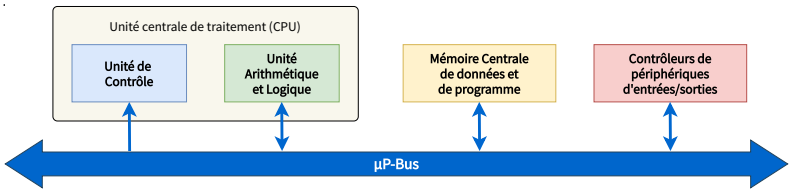


FIGURE 2.13 – Architecture von Neumann

Dans cette architecture, un bus système commun (µP-Bus) connecte l'unité de contrôle, l'unité arithmétique et logique avec sa banque de registres à la mémoire centrale et aux contrôleurs de périphériques d'entrées/sorties. La mémoire quant à elle contient aussi bien les instructions du programme que les données. On utilisera plutôt une mémoire non volatile pour stocker le code et une mémoire volatile pour les données.

Cette architecture permet de diminuer les nombres de broches nécessaires pour interfacer la mémoire centrale et les contrôleurs de périphériques d'entrées/sorties avec le CPU. Cependant, ceci est au détriment de la performance. En effet, l'accès au code et l'accès aux données doivent s'exécuter séquentiellement contrairement à l'architecture Harvard.

Architecture moderne des µP SISD

Les processeurs modernes utilisent aussi bien l'architecture Harvard que von Neumann (figure 2.14).

L'architecture Harvard permet d'interfacer les deux mémoires caches de premier niveau, le i-cache (les instructions) et le d-cache (les données). Tandis que l'architecture von Neumann sert à interfacer les mémoires et les contrôleurs de périphériques internes au µP ainsi que la mémoire centrale et les contrôleurs de périphériques externes à l'unité centrale.

Cette architecture se retrouve sur les µP ARM mono-coeur.

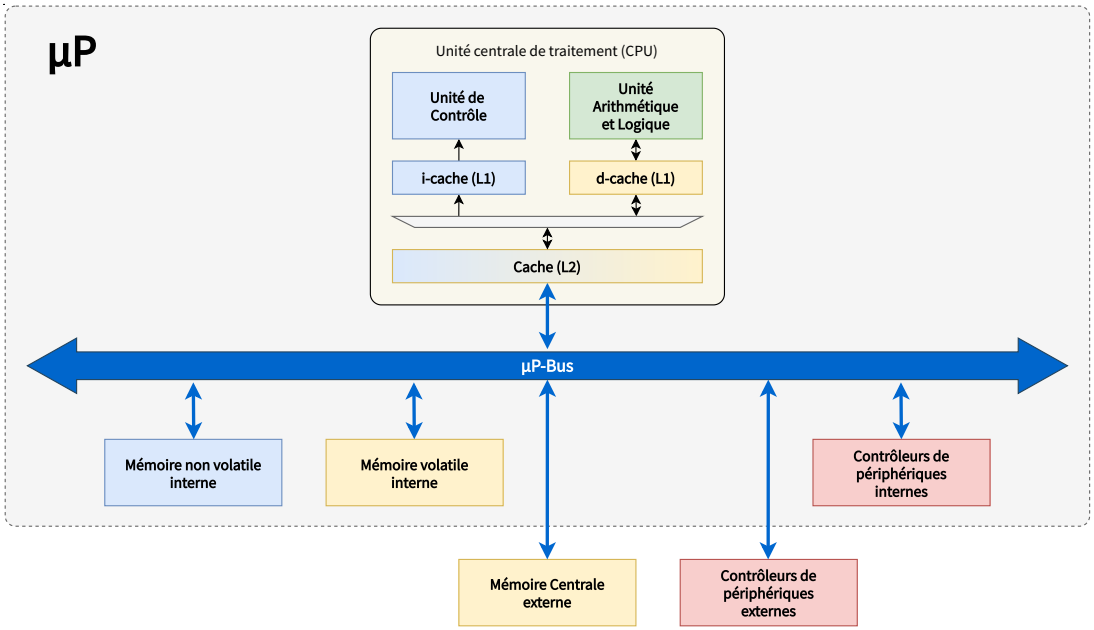


FIGURE 2.14 – Architecture moderne des µP SISD

2.5.2 Architecture SIMD

Dans l'architecture SIMD, l'unité centrale de traitement dispose de plusieurs unités arithmétiques et logiques (figure 2.15), permettant d'exécuter la même instruction sur plusieurs ALU. Cette architecture est particulièrement bien adaptée au calcul vectoriel et matriciel.

Le calcul vectoriel ou matriciel nécessite généralement d'effectuer la même opération en parallèle sur des données différentes (figure 2.16), par exemple des additions, des soustractions, des multiplications.

Les processeurs graphiques (GPU - *Graphics Processing Unit*) utilisent cette architecture. On la retrouve également sur le coprocesseur mathématique NEON des µP ARM.

2.5.3 Architecture MISD

Dans l'architecture MISD, l'unité centrale de traitement dispose de plusieurs unités de contrôle et unités arithmétiques et logiques (figure 2.17), permettant d'exécuter en parallèle des instructions sur la même donnée.

Étant donné que chaque ALU effectue la même opération sur la même donnée, cette architecture est capable de détecter des erreurs d'exécution et permet ainsi de construire des ordinateurs tolérants aux pannes. Cependant, il n'existe que très peu de processeurs de ce type. Le processeur IBM AP-101S,

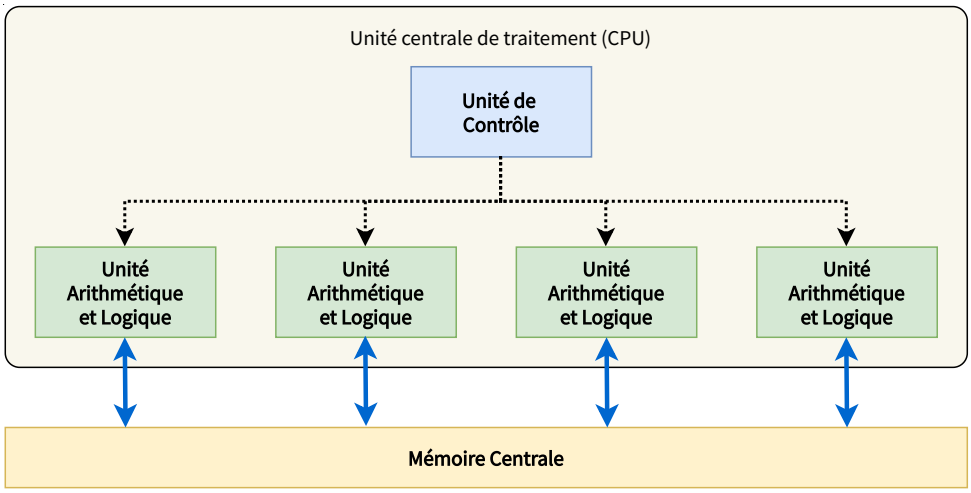


FIGURE 2.15 – Architecture SIMD

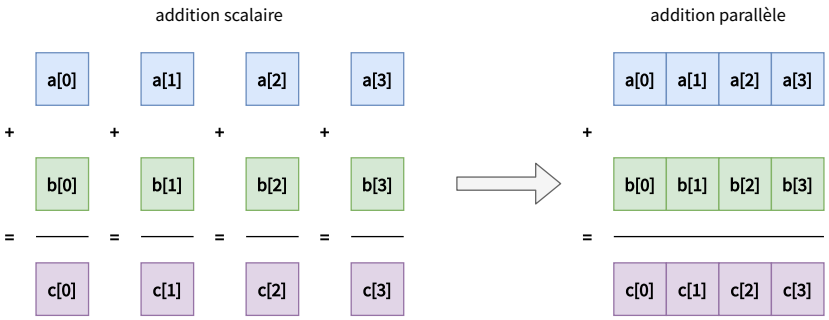


FIGURE 2.16 – Exemple d'addition matricielle

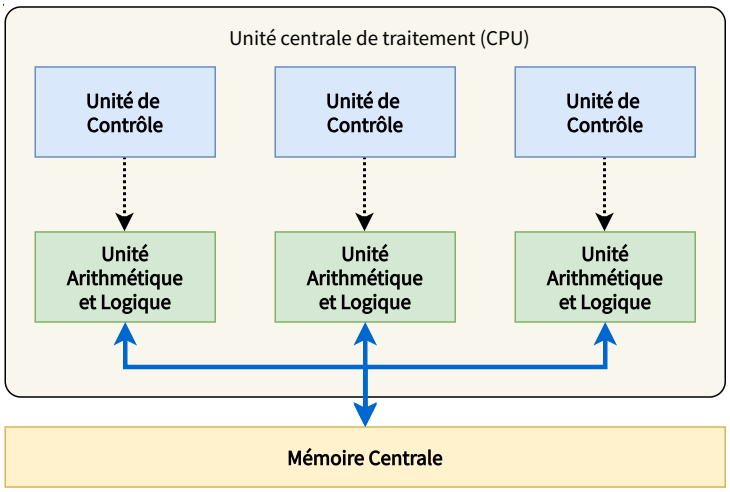


FIGURE 2.17 – Architecture MISD

utilisé par la navette spatiale “Space Shuttle” comme ordinateur de contrôle, en est l’un des rares exemples.

2.5.4 Architecture MIMD

Dans l’architecture MIMD, le processeur est une machine composée de plusieurs unités centrales de traitement (*CPU*), lesquelles sont nommées *coeur* (*core*). Ces machines sont capables de traiter en parallèle plusieurs flux de données et d’instructions distincts de façon complètement asynchrone (figure 2.18).

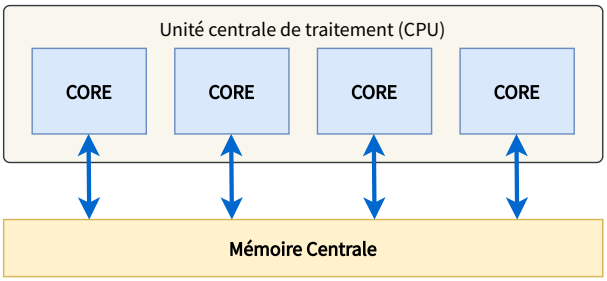


FIGURE 2.18 – Architecture MIMD

L’architecture MIMD distingue deux modèles, le MIMD à mémoire partagée (*Shared Memory*) et le MIMD à mémoire distribuée (*Distributed Memory*). Dans le modèle à mémoire partagée, les coeurs du processeur se partagent une même mémoire centrale (figure 2.19). La communication entre les coeurs du processeur s’effectue au travers de la mémoire.

Dans le modèle à mémoire distribuée, chaque coeur dispose de sa propre mémoire centrale. La communication entre les coeurs s’effectue au travers d’un réseau d’interconnexion (figure 2.20). Différentes topologies permettent de connecter les unités de traitement, dont la topologie en arbre, la topologie maillée ou la topologie hypercube.

Le modèle à mémoire partagée est plus simple à programmer que le modèle à mémoire distribuée. Tel GNU/Linux, de nombreux systèmes d’exploitation modernes supportent ce modèle. Par contre, le modèle à mémoire distribuée permet de réaliser des systèmes plus résistants aux pannes et des systèmes plus évolutifs.

2.6 Architectures des jeux d’instructions

De nombreux facteurs influencent les performances d’un ordinateur, c’est-à-dire le temps de calcul que le programme utilise pour traiter les données. L’énergie nécessaire à ces traitements est égale-

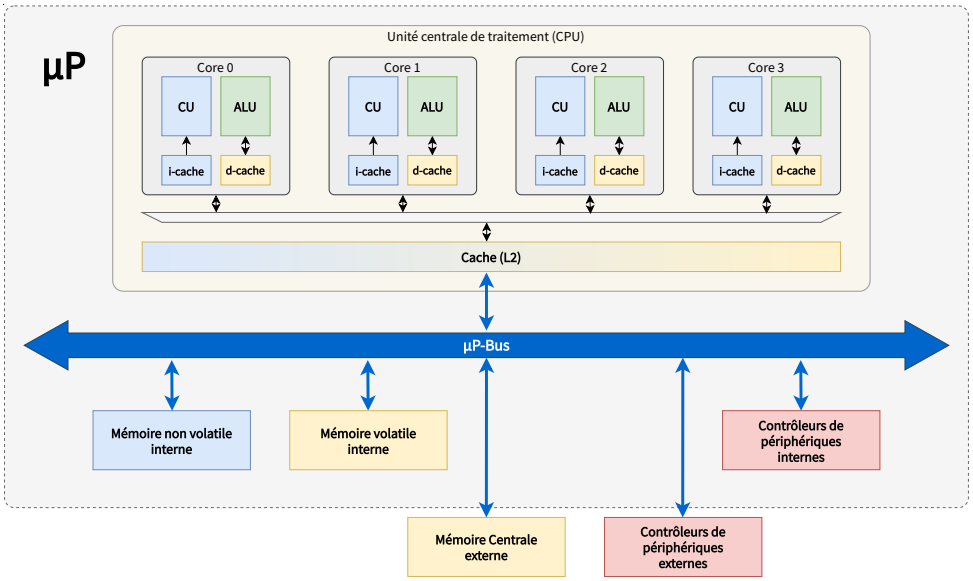


FIGURE 2.19 – Architecture MIMD à mémoire partagée

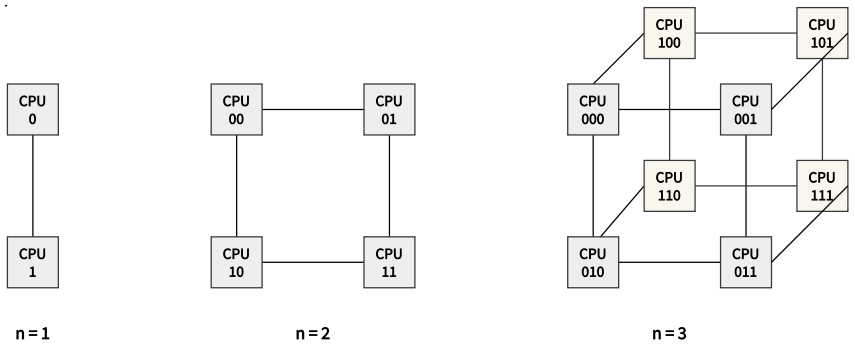


FIGURE 2.20 – Architecture MIMD à mémoire distribuée (hypercube)

ment un facteur déterminant pour la mise en oeuvre d'un système embarqué ou d'un objet de l'Internet.

2.6.1 Performances et temps d'exécution

D'un point de vue logiciel, trois facteurs principaux sont à considérer pour évaluer les performances d'un processeur (*Iron Law*) :

- le nombre d'instructions contenues dans le programme
- le nombre de cycles d'horloge nécessaires pour exécuter une instruction
- le temps d'un cycle d'horloge

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{ClockCycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{ClockCycles}}$$

Le nombre d'instructions dans un programme est naturellement largement dû à la complexité du problème à solutionner et de l'application qui en découle. Le choix des structures de données et d'algorithmes appropriés est primordial à la réalisation d'un code performant. Si le temps d'un cycle d'horloge (CP - *Clock Period*) ne dépend que de la fréquence de l'horloge système, le nombre de cycles d'horloge pour exécuter une instruction élémentaire (CPI - *Clock Per Instruction*) dépend quant à lui principalement de l'architecture du jeu d'instructions (ISA - *Instruction Set Architecture*) du processeur et de son implémentation.

Le temps nécessaire à l'exécution d'un programme ou d'un algorithme se calcule en multipliant ces trois facteurs :

$$\text{Time} = \text{Instructions} \times \text{CPI} \times \text{CP}$$

Avec l'évolution des processeurs, différentes architectures du jeu d'instructions (ISA - *Instruction Set Architecture*) coexistent sur les différentes familles de processeurs. Actuellement, on distingue généralement deux architectures principales :

- CISC : Complex Instruction Set Computer
- RISC : Reduced Instruction Set Computer

Hormis les processeurs de la famille x86 d'Intel, y compris les versions 64 bits, qui utilisent un jeu d'instructions étendu (CISC), la plupart des autres processeurs modernes utilisent un jeu d'instructions réduit (RISC) et en particulier les processeurs ARM.

2.6.2 Architecture CISC

Les processeurs de l'architecture CISC disposent d'un très grand nombre d'instructions, certaines très complexes, mélangeant opérations arithmétiques ou logiques avec modes d'adressages complexes. La taille des instructions est de longueur variable. Le nombre de cycles nécessaire à l'exécution des instructions est variable et dépend de ces dernières.

L'exemple ci-dessous montre une simple incrémentation d'une variable en C.

```
1    int a = 15;
2    void inc() { a++; }
```

Après compilation de ce code pour un processeur x86 à 64 bits, on obtient le code machine suivant :

```
1    0000000100003f90 _inc:
2    100003f90: 55                   pushq %rbp
3    100003f91: 48 89 e5             movq %rsp, %rbp
4    100003f94: ff 05 6a 00 00 00    incl 106(%rip)
5    100003f9a: 5d                   popq %rbp
6    100003f9b: c3                   retq
```

Deux caractéristiques principales des machines CISC sont facilement visibles, premièrement la taille variable des instructions et deuxièmement l'instruction `incl` permettant d'incrémenter directement de la variable `a` dans la mémoire sans devoir préalablement charger son contenu dans un registre interne du µP pour effectuer cette incrémentation.

2.6.3 Architecture RISC

Les processeurs de l'architecture RISC disposent d'un jeu d'instructions simple, facile à décoder et de taille fixe. Cette taille fixe facilite grandement l'architecture de la mémoire cache des instructions et la mise en oeuvre d'un pipeline. Grâce au pipeline, chaque instruction est exécutée en "un" cycle d'horloge (CPI = 1.0). Cette architecture permet également de réduire la perte d'énergie par dissipation thermique.

Pour des raisons de simplicité, les processeurs RISC ne disposent pas d'instructions permettant d'exécuter des opérations arithmétiques et logiques directement sur des données en mémoire. Par conséquent, le µP doit préalablement charger la donnée dans un de ses registres internes pour être traitée et finalement restockée dans la mémoire. On parle d'architecture *Load-Store*.

Après compilation du code C de l'exemple ci-dessus incrémentant une variable, on obtient le code machine suivant pour un processeur ARMv7, µP à 32 bits :

```
1    void inc() { a++; }
2    80001120: e59f200c   ldr r2, [pc, #12]
```

```
3      80001124: e5923000   ldr r3, [r2]
4      80001128: e2833001   add r3, r3, #1, 0
5      8000112c: e5823000   str r3, [r2]
6      80001130: e12fff1e   bx lr
7      80001134: 80006a00   .word 0x80006a00
```

Avec l'exemple ci-dessus, il est aisé de constater que la taille des instructions est bel et bien fixe.

Toutes les instructions étant de taille fixe, le µP ne peut obtenir l'adresse de la variable `a` que de manière indirecte. Pour cela, il charge dans un de ses registres internes un mot de 32 bits stocké dans le code en utilisant l'adresse courante du compteur ordinaire (PC - *Program Counter*) et une offset (un décalage). Ce mot de 32 bits est le résultat du compilateur et de l'éditeur de liens qui stockent l'adresse de la variable dans le code, ci-dessus après la dernière instruction de la routine. Le compilateur calcule l'offset correspondante au décalage entre l'adresse de l'instruction de chargement de l'adresse de la variable `a` à sa position dans le code. Dans l'exemple ci-dessus, le décalage est de 20 octets (0x80001134 - 0x80001120), mais l'offset n'est que de 12 octets, car le *PC* sur les µP ARMv7 est déjà incrémenté de 8 octets lors de l'exécution de l'instruction. On remarque également que pour incrémenter la variable `a`, le µP doit effectuer trois opérations, une première de lecture de la valeur de `a` (`ldr r3, [r2]`), puis l'incrément de la variable dans le registre du µP (`add r3, r3, #1, 0`) et finalement le stockage du résultat dans la mémoire (`str r3, [r2]`).

2.7 Information et données

En informatique, l'information est un ensemble d'unités de données stockées dans la mémoire centrale ou dans les registres des contrôleurs de périphériques d'entrées/sorties. Selon le type d'information à traiter par le µP, les données peuvent prendre différentes formes (texte, valeurs numériques, valeurs booléennes, etc. ou des instructions d'un programme). Cependant, toutes les données sont représentées par une suite de *bits*, où le *bit* (symbole *b*) désigne un chiffre binaire `0` ou `1`.

2.7.1 Représentation mémoire

Vu du µP, la mémoire est un tableau d'éléments à une dimension (figure 2.21). Chacun de ces éléments représente une unité de données. La taille de ce tableau, appelé espace adressable ou espace mémoire, dépend du nombre de bits dont dispose le µP pour l'accéder, lequel correspond à la taille de l'*adresse*.

La plus petite unité de données adressable et accessible est le ***byte*** (symbole *B*). Le terme ***byte*** vient de l'anglais *bite*, signifiant bouchée, mais écrit avec un *y* afin d'éviter toute confusion avec le *bit*. Le nombre de bits contenus dans un ***byte*** n'est pas défini, cependant des *bytes* à 8 bits, un *octet*

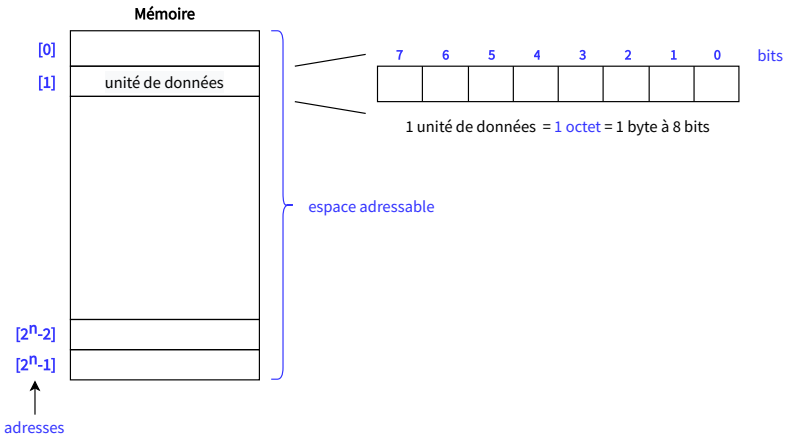


FIGURE 2.21 – Espace adressable

(symbole *o*), sont la norme aujourd'hui. Par convention, la numérotation des bits d'un octet se fait généralement de droite à gauche (figure 2.21) et le bit de poids faible est le bit 0 tandis que le bit de poids fort est le bit 7.

2.7.2 Capacité mémoire

La capacité d'une mémoire, le chip, se donne par les fabricants en bits. Par contre, la capacité mémoire d'un ordinateur ou d'un système à microprocesseur se donne en bytes.

Pour éviter toutes confusions entre le système décimal (puissances de 10) et le système binaire (puissances de 2), l'organisation internationale de normalisation (ISO) ainsi que la commission électronique internationale (IEC) ont proposé une nouvelle nomenclature pour indiquer la capacité mémoire d'un ordinateur, soit :

Décimal			Binaire			
Valeur	Symbole	Terme	Valeur	Symbole	Terme	Anciennement
1000^1	kB	kilobyte	1024^1	KiB	kibibyte	KB - kilobyte
1000^2	MB	megabyte	1024^2	MiB	mebibyte	MB - megabyte
1000^3	GB	megabyte	1024^3	GiB	gibibyte	GB - gigabyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte	TB - terabyte
1000^5	PB	petabyte	1024^5	PiB	pebibyte	-
1000^6	EB	exabyte	1024^6	EiB	exbibyte	-

Décimal			Binaire			
Valeur	Symbole	Terme	Valeur	Symbole	Terme	Anciennement
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte	-
1000^8	YB	yottabyte	1024^8	YiB	yobibyte	-

Si l'on souhaite indiquer la capacité en bits au lieu de bytes, il suffit de remplacer le *B* du symbole par un *b*.

2.7.3 Accès mémoire

La figure ci-dessous (figure 2.22) présente un ordinateur élémentaire, avec juste une unité centrale de traitement, une mémoire volatile pour les données, un mémoire non volatile pour le programme ainsi que des contrôleurs de périphériques d'entrées/sorties. Trois bus offrent au CPU une interface de communication pour le transfert de données entre ses composants internes (registres et unité de contrôle) et les composants externes (mémoire centrale et contrôleurs de périphériques).

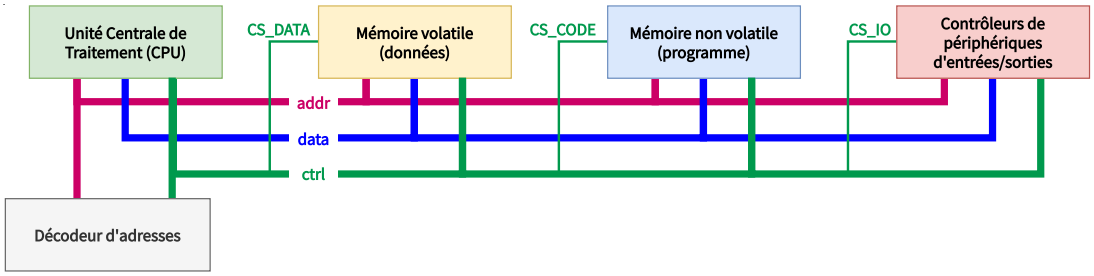


FIGURE 2.22 – Bus du µP

Les bus servent à regrouper les signaux d'une même fonctionnalité, soit :

- Bus d'adresses (*addr*)
- Bus de données (*data*)
- Bus de contrôle (*ctrl*)

Le bus d'adresses (*addr*) permet d'identifier et d'accéder de façon univoque chaque unité de données ou chaque octet dans l'espace mémoire. La taille du bus d'adresses dépend fortement de l'architecture du processeur. Pour des processeurs avec une architecture à 32 bits, il est généralement également de 32 bits, offrant ainsi un espace adressable de 4GiB. Par contre pour les processeurs avec une architecture de 64 bits, il est généralement plus étroit, par exemple 44 bits (16TiB) pour les µP Cortex ARMv8-A et de 48 bits (256TiB) pour les processeurs x86-64.

Le bus de données (*data*) permet le transfert de l'information entre le processeur et ses composants placés en espace mémoire. La largeur du bus de données dépend naturellement fortement de l'architecture du µP, mais également des objectifs en coûts matériel que les fabricants souhaitent atteindre (nombre de broches, encombrement, prix, etc.). Il est assez courant que des systèmes basés sur une architecture 32 bits ne disposent que d'un bus mémoire à 16 bits. Cependant et pour maximiser les performances globales du système, les fabricants préféreront un bus de données de même taille que celui de l'architecture, voire plus large. Sur les systèmes sur puce (SoC - *System on Chip*) équipés d'une logique programmable (FPGA - *Field Programmable Gate Arrays*), il est courant que les bus de données internes soient très larges, 256 bits, 512 bits, voire même 1024 bits. Indépendamment de la largeur du bus de données, le µP peut effectuer des transferts de tailles variables, 8 bits, 16 bits, 32 bits voire 64 bits.

Le bus de contrôle (*ctrl*) regroupe tous les signaux nécessaires à la gestion des transferts entre le µP et ses composants. Les signaux principaux sont :

— Type de transfert, lecture ou écrire (R/W_ - *Read/Write_*)
— Taille du transfert (SZ - *Size*)
— Lignes de sélection des contrôleurs de périphériques (CS - *Chip Select*)
— Lignes d'interruptions (IRQ-Lines - *Interrupt Request Lines*)

Le décodeur d'adresses (*address decoder*) sert à adresser séparément chacun des composants en espace mémoire. En effet, pour des raisons de coût et de simplicité, les différents composants placés dans l'espace mémoire du µP ne disposent que des lignes du bus d'adresses correspondant à la taille (en puissance de 2) de l'ensemble de leurs registres pour des contrôleurs de périphériques ou la capacité (également en puissance de 2) pour des mémoires. Selon un schéma pré-établi, le décodeur d'adresses génère un *chip select* afin d'activer le contrôleur de périphériques ou la mémoire correspondant à l'adresse mise sur le bus.

2.7.4 Alignement

Afin d'augmenter les performances du µP, la taille du bus de données correspond généralement à l'architecture du processeur, typiquement pour un µP avec une architecture à 32 bits, son bus de données sera également de 32 bits. Si le µP doit accéder une donnée de 4 octets (32 bits) et que son adresse est divisible par 4, on parle d'un accès aligné (*aligned access*) (figure 2.23) et dans le cas contraire on parle d'accès non aligné (*unaligned access* ou *misaligned access*).

Selon la conception du µP, des accès non alignés peuvent conduire à :

— Violation de contrainte
— Performances non optimales

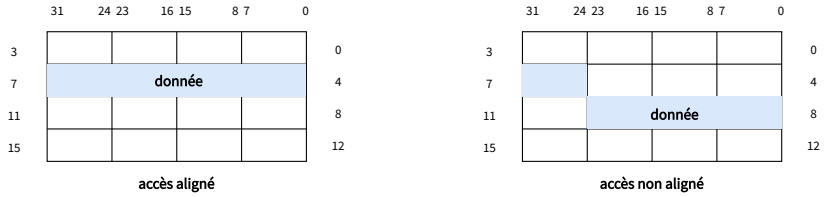


FIGURE 2.23 – Alignement des données en mémoire

Dans le cas d'une violation de contrainte, le µP arrête l'exécution de l'instruction et génère une exception. Dans le cas d'alignement non optimal, le processeur effectuera plusieurs accès alignés (dans l'exemple ci-dessous 2 accès de 4 octets) afin d'extraire les octets correspondants à la requête. Ces multiples accès prennent plus de temps et vont simplement dégrader les performances globales du système.

La règle pour qu'une donnée soit bien alignée en mémoire est que son adresse soit un multiple de sa taille en octets :

— par 8 pour des mots de 64 bits (8 octets)
— par 4 pour des mots de 32 bits (4 octets)
— par 2 pour des mots de 16 bits (2 octets)

Les données d'un seul octet (8 bits) sont quant à elles toujours bien alignées.

2.7.5 Boutisme

En informatique, la représentation de certains types de données nécessite plus d'un octet. On peut citer par exemple les nombres entiers sur 16, 32 ou 64 bits ou les nombres à virgule flottante qui occupent généralement 32 bits pour les *float* ou 64 bits pour les *double*. Le boutisme (*endianness*) désigne l'ordre dans lequel les octets d'une donnée sont stockés dans la mémoire. Il existe deux orientations, lesquelles dépendent de l'architecture du processeur :

— Petit-boutisme (*little-endian*)
— Gros-boutisme (*big-endian*)

Dans l'orientation *petit-boutisme*, pour une donnée dont le type est plus grand que 8 bits, l'octet de poids faible est stocké à l'adresse basse en mémoire, tandis que l'octet de poids fort est placé à l'adresse haute. Si l'on choisit une la valeur entière hexadécimale 0x44332211, l'ordre des octets dans la mémoire est 0x11, 0x22, 0x33 et 0x44 (figure 2.24) en allant de l'adresse basse vers la haute. Les processeurs d'*Intel* et *ARM* utilisent cette convention.

Dans l'orientation *gros-boutisme*, pour la même donnée, l'ordre est inversé, c'est-à-dire l'octet de poids faible est stocké à l'adresse haute en mémoire, tandis que l'octet de poids fort est placé à

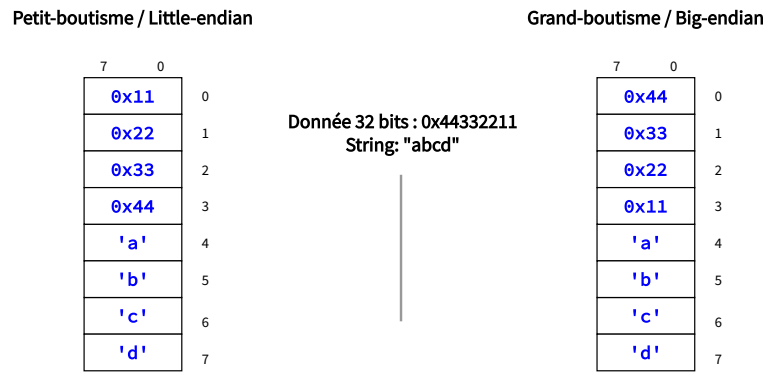


FIGURE 2.24 – Boutisme

l'adresse basse. L'ordre des octets dans la mémoire devient alors 0x44, 0x33, 0x22 et 0x11. Les processeurs *PowerPC* et *MIPS* utilisent cette convention. Il intéressant de noter que les protocoles de communications TCP/IP utilisent également cette convention.

En C/C++, ces conventions ne sont utilisées que pour les types de base, tels que `int`, `float`, `double` ou `enum`. Par contre, pour les types complexes, tels que `struct` ou `array`, l'ordre de déclaration des différents attributs dans la structure, respectivement l'index de l'élément dans le tableau, seront utilisés pour stocker les données en mémoire, en allant toujours des adresses basses vers les hautes.

2.8 Mémoire centrale

Sur les systèmes informatiques, il existe un très grand nombre de moyens pour stocker l'information et les programmes permettant de les traiter. Ces moyens dépendent fortement du type d'information à stocker et de leur volume.

Dans le cadre des systèmes embarqués et des objets de l'Internet basés sur des systèmes sur puce (SoC - *Systems on Chip*), on distingue généralement deux grands types de mémoires :

— Mémoire volatile
— Mémoire non volatile

La mémoire volatile sert à stocker temporairement les données utilisées que pour le traitement de l'information et qui nécessitent d'être conservées que si le système est sous tension et fonctionnel. La mémoire non volatile sert, comme son nom l'indique, à stocker de façon persistante des données, telles que les programmes ou des données de configuration, afin que l'on puisse les récupérer si le système est mis hors tension un certain temps.

2.8.1 Mémoire volatile

Les systèmes embarqués et SoC utilisent de la mémoire vive (RAM - *Random Access Memory*) comme mémoire volatile. Cette mémoire vive offre au µP un accès direct à l'information (également appelé accès aléatoire). Le µP peut lire ou écrire n'importe quelle donnée en se rendant directement à l'adresse (emplacement dans la mémoire) où cette donnée est stockée. La deuxième caractéristique principale de cette mémoire est sa rapidité d'accès. En effet, le temps d'accès aux données stockés dans la mémoire a un impact considérable sur les performances des µP.

Il existe deux types principaux de mémoires vives :

— les mémoires vives statiques (SRAM - *Static RAM*)
— les mémoires vives dynamiques (DRAM - *Dynamic RAM*)

Static RAM (SRAM)

La figure 2.25 représente une cellule d'un bit de mémoire vive SRAM. Cette cellule utilise 6 transistors pour stocker un bit.

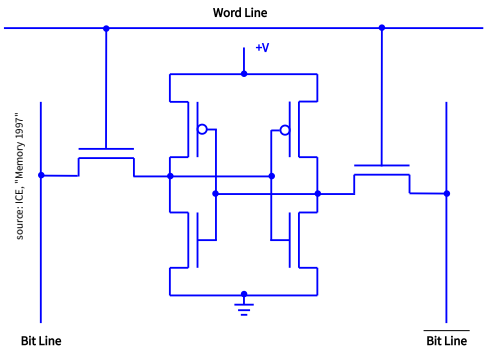
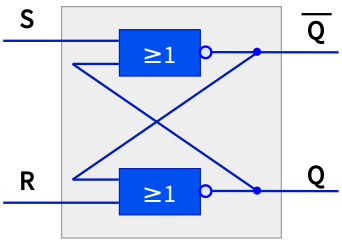


FIGURE 2.25 – SRAM - cellule de 1 bit à 6 transistors

Le fonctionnement de cellule mémoire d'un bit se base sur le fonctionnement d'une bascule de type RS (Reset/Set). La figure 2.26 représente une telle bascule construite avec des portes NOR.

Vu le nombre de transistors nécessaires à la construction d'un bit mémoire, les SRAM ont une densité relativement faible. Cette faible densité fait que l'on ne trouve que des SRAM de taille relativement petite, de quelques kilo-bits à quelques mega-bits. Par contre elles offrent un accès excessivement rapide à l'information, 1 à 2 ns pour les SRAM on-chip et 5 à 10 ns pour des SRAM off-chip. Les SRAM ne nécessitent aucune logique de rafraîchissement, il suffit de les alimenter afin qu'elles conservent leurs contenus. Ces mémoires sont actuellement utilisées comme mémoire interne des µP (System on Chip) ou comme mémoire cache des processeurs d'applications.



S	R	Q	$\overline{Q}$	Remarque
0	0	Q	$\overline{Q}$	Valeur mémorisée
0	1	0	1	Mise à 0
1	0	1	0	Mise à 1
1	1	0	0	Cas particulier

FIGURE 2.26 – Bascule RS et sa table de vérité

Dynamic RAM (DRAM)

Avec une structure très simple pour enregistrer d'un bit d'information, une pico-capacité et un transistor (figure 2.27), les DRAM permettent d'obtenir une grande densité de stockage.

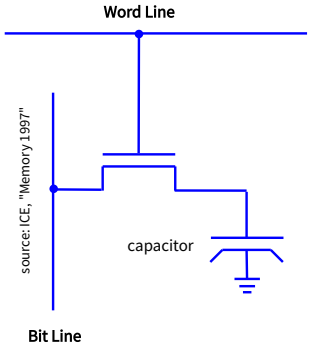


FIGURE 2.27 – DRAM - cellule de 1 bit (1 capacité et 1 transistor)

Cette simplicité souffre d'un inconvénient résidant dans la perte de charge due à des courants de fuite de la pico-capacité. Ces courants de fuite font disparaître l'information. Pour pallier à ce phénomène, il est nécessaire de rafraîchir régulièrement (période de quelques millisecondes) la charge des pico-condensateurs (figure 2.28).

Les mémoires vives dynamiques ont énormément évolué depuis les années 1990. Les DDR4 SDRAM (*Double Data Rate 4th generation Synchronous DRAM*) équipent la plupart des systèmes embarqués de dernière génération comme mémoire centrale externe au µP. Leurs tailles varient entre quelques centaines de MiB à quelques GiB avec un débit de transfert de quelques giga bits par seconde.

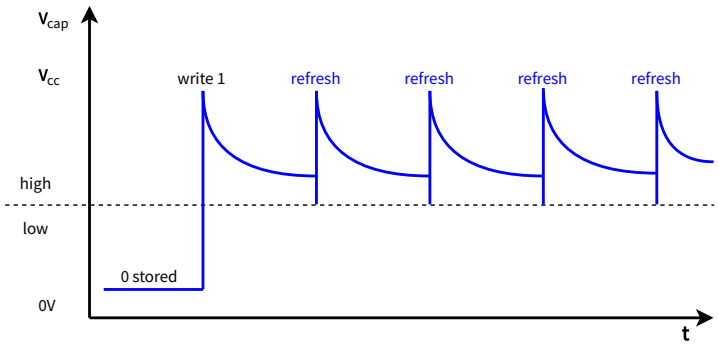


FIGURE 2.28 – DRAM - Refresh

Dual Ported RAM (DPRAM)

Des mémoires vives à accès double (DPRAM - *Dual Ported RAM*) équipent souvent des systèmes à multicoeurs asymétriques. Ces DPRAM sont un type de mémoire vive généralement basée sur de la SRAM offrant un double accès non bloquant en lecture et un seul accès en écriture. Ce type de mémoire est typiquement utilisé par des systèmes à multiprocesseurs nécessitant un canal de communication pour le transfert de données entre deux processeurs indépendants.

2.8.2 Mémoire non volatile

Dans le langage courant, on parle également de mémoire morte désignant une mémoire non volatile dont le contenu ne s'efface pas même si le système est mis hors tension, a contrario de la mémoire vive. Les systèmes embarqués et SoC utilisent des mémoires non volatiles à base de semi-conducteurs. Selon les caractéristiques de l'information à stocker (microcode/microprogramme, programmes ou données de configuration) ainsi que des besoins d'accès à cette information, on utilise différents types de mémoires :

— des mémoires ROM (*Read Only Memory*)
— des mémoires EEPROM (*Electrically Erasable Programmable ROM*)
— des mémoires NVRAM (*Non Volatile RAM*)
— des mémoires Flash

Mémoires ROM

Les mémoires ROM ne sont qu'accessibles en lecture. C'est durant le processus de production que leurs fabricants programment leur contenu. Longtemps utilisées pour stocker les fonctions du BIOS (BIOS - *Basic Input Output System*) des ordinateurs, remplacées à l'heure actuelle par des mémoires

Flash, elles sont toujours d'actualité pour les codes d'amorçage (BROM - *Boot ROM*) des µP de dernière génération.

Mémoires EEPROM

Les mémoires EEPROM, appelées également E2PROM, sont des mémoires non volatiles effaçables et programmables électriquement. Ce type de mémoire, d'une taille de quelques centaines de bytes à quelques KiB, sert généralement à stocker des données de configuration ou d'identification (numéro de série, type de système, etc.). Bien qu'il soit possible d'accéder des EEPROM avec une interface parallèle, c'est plus souvent un bus série qui sert d'interface. Hormis les bus SPI (*Serial Peripheral Interface*) et 1-Wire, le bus I2C (*Inter-Integrated Circuit*) est l'un des plus courants.

En lecture, le µP spécifie l'adresse du premier octet de données stocké dans l'EEPROM et cette dernière lui retourne un nombre d'octets correspondant au nombre de cycles de lecture que le µP effectue. En écriture, les EEPROM offrent souvent deux modes d'opération, "byte write" et "page write". En mode "byte write", le µP transfère sur le bus l'adresse et l'octet de données à stocker. En mode "page write", le µP place sur le bus l'adresse de départ des données ainsi que le nombre de données correspondant à la taille de la page (par exemple 64 octets). Une fois le transfert de données effectué (byte ou page write), l'EEPROM procède à l'écriture des données. Cette opération dure généralement entre 5 et 10 ms. Les EEPROM proposent généralement un protocole d'acquittement afin de permettre au µP de déterminer la fin du cycle d'écriture et d'en optimiser ainsi l'accès.

Mémoires NVRAM

Les mémoires NVRAM sont des mémoires RAM qui retiennent leur contenu même si elles ne sont plus alimentées. Ce type de mémoires est généralement utilisé pour stocker des données processus, par exemple des valeurs de calibration ou des données d'échantillonnage, que l'on souhaite ne pas perdre si le système est mis hors tension un certain temps dû par exemple à une coupure d'alimentation inopinée.

Il existe différentes technologies pour réaliser des mémoires RAM non volatile, mais la technique la plus courante consiste à utiliser une mémoire vive statique (SRAM) avec une capacité. On parle dans ce cas de pseudo-NVRAM.

Selon l'application et le temps de stockage, cette capacité peut être une pile, une batterie ou un condensateur. Sur les µP utilisés dans les objets de l'Internet ou les systèmes embarqués, il est usuel de trouver de telles réalisations. Une de celles-ci est l'horloge temps réel (RTC - *Real Time Clock*).

Mémoires Flash

Les mémoires non volatiles de type Flash apparaissent vers la fin des années 1980. Cette nouvelle technologie révolutionne la micro-informatique. Il devient enfin possible de réaliser des mises à jour logicielles sans devoir envoyer sur site des équipes de maintenance. Les corrections de logiciels ainsi que les nouvelles versions des applications peuvent être téléchargées à distance.

Selon le type de porte logique utilisée comme cellules de stockage des bits d'information, les mémoires Flash se déclinent en deux types :

— les Flash NOR
— les Flash NAND

Ces cellules servant au stockage des bits d'information sont construites avec des transistors MOS (*Metal Oxide Semiconductor*) à grille flottante (figure 2.29).

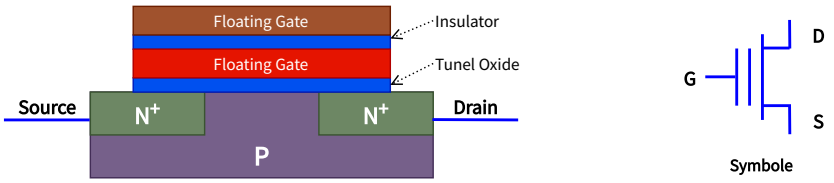


FIGURE 2.29 – Transistor MOS à grille flottante

La grille flottante piège les charges électriques et permet ainsi de stocker l'information. Afin de pouvoir écrire des données dans une mémoire Flash, il est nécessaire d'effacer préalablement la cellule. Cette opération consiste à charger la grille flottante, c'est-à-dire à injecter des électrons dans la grille. Pour cela il faut appliquer une tension de 12V sur la grille de contrôle. Une fois chargée, il est possible de procéder à l'écriture des données. Ces opérations d'effacements et d'écritures sont excessivement agressives pour la grille flottante et affectent la durée de vie du transistor. La rétention de l'information n'est pas illimitée. Elle varie également avec la température d'opération de la mémoire ainsi que du type de portes logiques utilisé.

Flash NOR

Les Flash NOR garantissent un stockage de l'information à 100% sur une période supérieure à 20 ans avec plus 100'000 cycles "effacement/écriture". Leur densité relativement faible limite leur taille de quelques KiB à quelques MiB. Elles offrent un accès direct et aléatoire à chacune des positions mémoire contenant les données organisées généralement en mots de 8 ou 16 bits. Avec cette interface, le µP peut exécuter le code de programmes informatiques directement depuis la mémoire Flash sans devoir le copier préalablement dans la RAM. Cette caractéristique se nomme XIP (*eXecute In Place*).

Les Flash NOR sont spécialement bien adaptées aux systèmes sur puce (SoC) à faible consommation et avec des capacités mémoire restreintes et intégrées directement à l'intérieur du µP.

Les Flash NOR ont une organisation par bloc de quelques KiB, lesquels peuvent être effacés et programmés indépendamment les uns des autres. Selon le type d'applications, ces blocs peuvent avoir la même taille ou des tailles différentes. Des tailles de bloc différentes, comme pour la Flash NOR M29FxxxFT/B de Micron (figure 2.30) permet de stocker des types de données différentes, par exemple un code d'amorçage (*bootcode*) sur les blocs de 16KiB et et 32KiB, des données de configuration sur les blocs de 8KiB et les applications et systèmes de fichiers sur les blocs de 64KiB.

Grâce à un accès direct aux cellules mémoire de la Flash NOR, aucun protocole de lecture n'est nécessaire. Par contre, l'enregistrement de données nécessite un protocole, lequel consiste à écrire des commandes à des adresses spécifiques dans la Flash NOR. Ce protocole est standardisé et connu sous le nom de CFI (*Common Flash Interface*). Ce standard sert d'interface pour la réalisation de systèmes de fichiers, tels que FAT (*File Allocation Table*) ou JFFS2 (*Journalling Flash File System 2*).

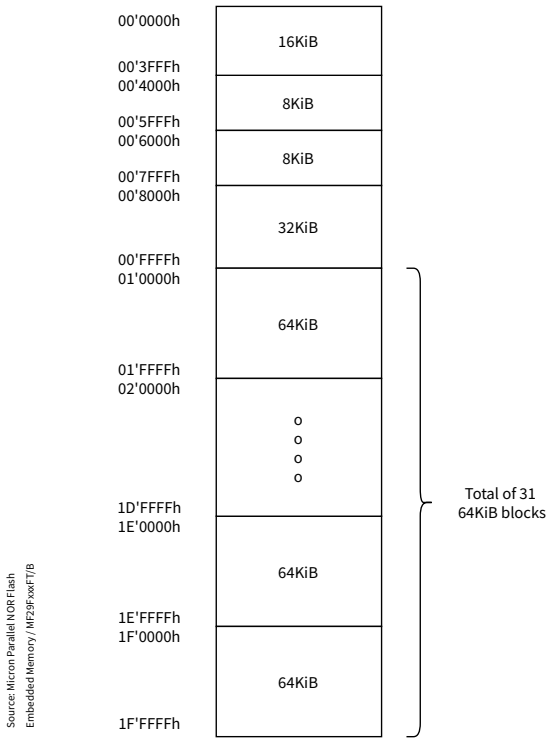


FIGURE 2.30 – Organisation des blocs de la Flash NOR M29FxxxFT/B

Avant de pouvoir écrire des données dans un bloc de la mémoire, il faut préalablement procéder à l'effacement du bloc. Cette procédure a pour effet de placer tous les bits au niveau logique "1". L'algorithme ci-dessous montre l'effacement d'un bloc :

```c
void flash_block_erase(offset_t offset) {
    uint8_t* cmd_555 = (uint8_t*)(FLASH_BASE_ADDR + 0x555);
    uint8_t* cmd_2aa = (uint8_t*)(FLASH_BASE_ADDR + 0x2aa);
    uint8_t* address = (uint8_t*)(FLASH_BASE_ADDR + offset);

    *cmd_555 = 0xaa;  // initiate block erasing sequence
    *cmd_2aa = 0x55;
    *cmd_555 = 0x80;
    *cmd_555 = 0xaa;
    *cmd_2aa = 0x55;
    *address = 0x30;  // erase the block corresponding to the address

    while(1) {        // wait until block is completely erased
        uint8_t status = *address
        if (status == *address) break;
    }
}
```

Une fois le bloc effacé, il devient possible d'écrire des données. Avec cette procédure il est seulement possible de placer au niveau logique "0" des bits de données. S'il est nécessaire de placer au niveau logique "1" de bits de données, il faudra alors procéder à l'effacement du bloc. L'algorithme ci-dessous décrit l'écriture de données dans un bloc.

```c
void flash_program_byte(offset_t offset, uint8_t data) {
    uint8_t* cmd_555 = (uint8_t*)(FLASH_BASE_ADDR + 0x555);
    uint8_t* cmd_2aa = (uint8_t*)(FLASH_BASE_ADDR + 0x2aa);
    uint8_t* address = (uint8_t*)(FLASH_BASE_ADDR + offset);

    *cmd_555 = 0xaa;  // initiate data programming sequence
    *cmd_2aa = 0x55;
    *cmd_555 = 0xa0;
    *address = data;  // program the data byte

    while(1) {        // wait until data is completely programed
        uint8_t status = *address
        if (status == *address) break;
    }
}
```

Les Flash NOR ont de bons temps d'accès en mode lecture (quelques dizaines de ns), par contre les temps d'effacement (entre 0.5s et 5s) et d'écriture (entre 11µs et 200µs) sont relativement longs.

NAND-Flash

Par leur structure interne, les Flash NAND offrent une plus grande densité, de quelques 100MiB à quelques GiB, à un coût par bit plus bas que les Flash NOR. Les Flash NAND ont une organisation par blocs et pages (figure 2.31). Le bloc est la plus petite unité effaçable, tandis que la page est la plus

petite unité lisible et programmable.

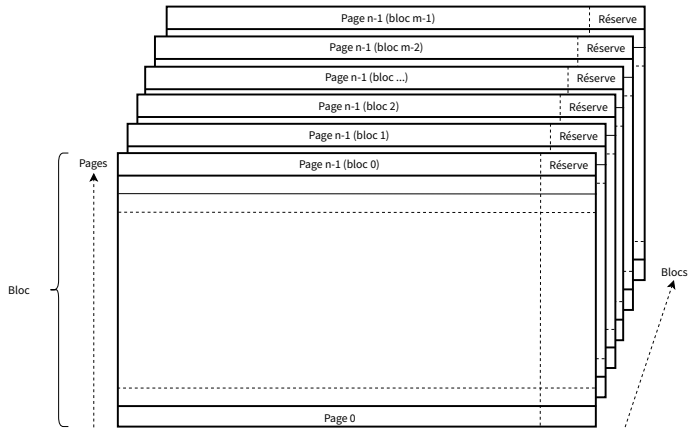


FIGURE 2.31 – Organisation des Flash NAND

Le nombre de pages par bloc ainsi que la taille d'une page peut varier selon les fabricants et les types de Flash. Cependant deux organisations sont courantes, les Flash à petits blocs (32 pages à 512 octets de données) et les Flash à larges blocs (64 pages à 2048 octets de données.)

Contrairement aux Flash NOR, les Flash NAND ne garantissent pas un stockage à 100% de l'information. Pour détecter ces erreurs et dans la mesure du possible les corriger, chaque page d'un bloc dispose, en plus des bits de données, de bits en réserve pour stocker un code de correction d'erreurs (ECC - *Error Correction Code*). Sur la durée de vie des Flash NAND, il arrive que certains blocs se dégradent de telle sorte qu'il ne soit plus possible de corriger ces erreurs (quelques pour cent). Dans ce cas, les blocs concernés sont marqués comme défectueux (*bad blocks*) et remplacés par des blocs de réserve. Les processus de fabrication produisent également un certain nombre de blocs défectueux.

La rétention des données est également plus courte que celle des Flash NOR, environ 10 ans. Par contre, elles permettent plus de cycles "effacement/écriture", environ 1'000'000. Le bloc est la plus petite unité effaçable. L'effacement d'un bloc ne prend que quelque 500µs.

Par leur architecture, les Flash NAND ne disposent que d'une interface série pour accéder à leur contenu. Cette interface nécessite la mise en place d'un contrôleur. Hormis le transfert des pages de données avec la mémoire Flash (écriture, lecture, effacement), ce contrôleur est en charge de gérer :

— les codes de correction d'erreurs
— les blocs défectueux
— l'usure uniforme des blocs (*wear leveling*)

Les Flash NAND sont disponibles sous différents types de boîtier. Dans le domaine des systèmes embarqués, la technologie eMMC (*embedded MMM - embedded Multi Media Card*) est très courante, car

elle permet de souder le chip directement sur la carte électronique. Les cartes mémoire SD, ou micro SD, servent souvent de support si le système ou l'application nécessite une carte amovible.

2.9 Périphériques

Les périphériques d'entrées/sorties sont des composants indispensables des processeurs. Ils leur servent à interfacer et interagir avec les systèmes physiques externes.

2.9.1 Architecture

L'architecture d'un périphérique d'entrées/sorties distingue deux parties principales (figure 2.32) :

— Contrôleur (CTRL)
— Chip physique (PHY)

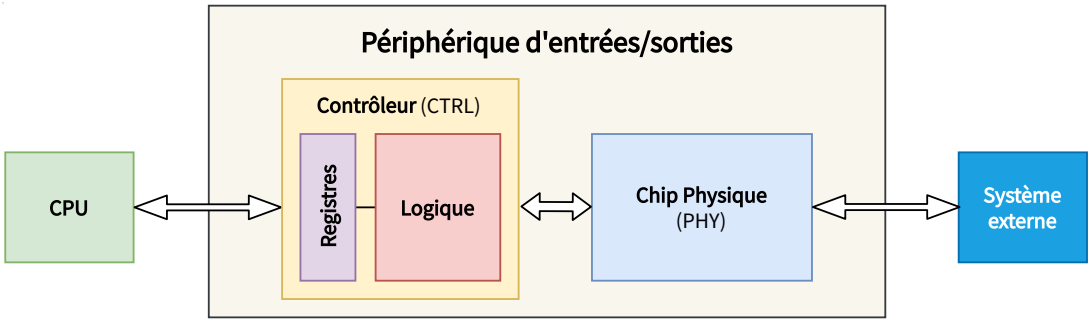


FIGURE 2.32 – Architecture d'un périphérique d'entrées/sorties

Le **contrôleur** (CTRL - *Controller*) implémente la logique du périphérique. Cette logique réalise la fonction du périphérique. Pour interfacer avec le logiciel, nommé pilote de périphérique (*Device Driver*), le contrôleur expose les données pour sa configuration et sa gestion via une banque de registres.

Le **chip physique** (PHY - *Physical Layer*) implémente la couche physique du périphérique. Il permet d'interfacer le périphérique physique externe au processeur avec le contrôleur interne. Contrairement au contrôleur, le chip physique se place généralement à l'extérieur de la puce du processeur, sur la carte électronique (PCB - *Printed Circuit Board*) et à proximité du connecteur.

2.9.2 Registres

Le pilote utilise les registres pour opérer le périphérique. Généralement, ces registres se déclinent en trois grandes catégories (figure 2.33) :

— Registres d'état
— Registres de contrôle
— Registres de données

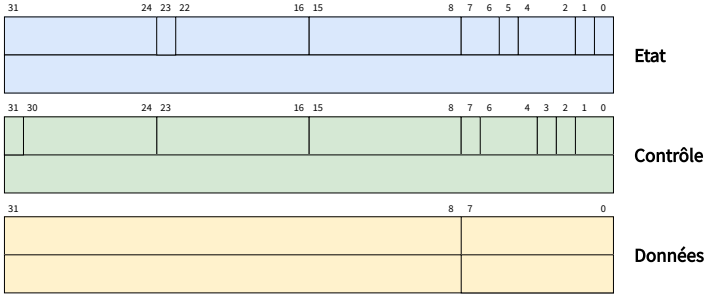


FIGURE 2.33 – Registres d'un contrôleur

Les **registres d'état** (*status registers*) donnent des informations sur l'état de fonctionnement actuel du périphérique, tels que sa disponibilité à émettre des données ou sur la réception de données, sur des erreurs ou malfonctions du périphérique. Ils fournissent également un feedback sur des opérations ou commandes appliquées au périphérique via les registres de contrôle.

Les **registres de contrôle** ou registres de commande (*control registers*) servent à la configuration et gestion du périphérique, tels qu'initialisation du périphérique (*reset*), mode de fonctionnement, contrôle des interruptions, gestion des canaux de transmissions/réceptions.

Les **registres de données** (*data registers*) se déclinent en principe en un registre de transmission (TX) et un registre de réception (RX). Derrière ces registres se cachent généralement deux tampons, un tampon d'émission de données (TX - *Transmit Buffer*) et un tampon de réception de données (RX - *Receipt Buffer*). Ces tampons permettent de réduire les contraintes temporelles sur le pilote, plus de temps avant de réagir lors de la réception de données ou possibilité d'émettre une rafale de données sans blocages.

Selon le type de périphériques, la taille de ces registres varie entre 8 bits, 16 bits et 32 bits. Si le périphérique est embarqué sur la même puce que le CPU, sa taille est généralement similaire à celle de l'architecture du processeur. Le regroupement de plusieurs bits ou champs de bits d'état, respectivement de contrôle, sur un seul registre est une pratique usuelle des concepteurs de périphériques. Cette pratique leur permet de réduire généralement la quantité de logique nécessaire au traitement des requêtes du pilote.

Contrairement aux cellules des mémoires, qui ne stockent que la donnée, un accès effectué sur un des registres d'un contrôleur engendre automatiquement une action sur périphérique, ceci indépendamment que l'accès soit une lecture ou une écriture. Il est donc essentiel de ne pas faire transiter les données de ces registres par les mémoires cache du processeur. Côté logiciel, il est également impor-

tant d'éviter toute optimisation du compilateur. Sans ces précautions, le comportement souhaité du périphérique risque d'être altéré.

2.9.3 Interfaces

Bien que l'architecture interne des contrôleurs de périphériques d'entrés/sorties soit similaire, leur interface avec le processeur varie selon le type de périphérique et de son niveau d'intégration avec le processeur (sur la puce du processeur ou placé à l'extérieur). Cependant, pour interfacer les registres des contrôleurs avec le CPU, les processeurs distinguent généralement trois types principaux de périphériques (figure 2.34) :

— MMIO (*Memory-Mapped I/O*)
— PMIO (*Port-Mapped I/O*)
— Coprocesseur

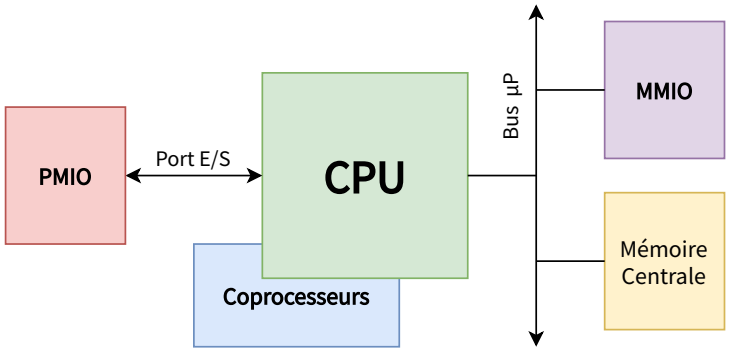


FIGURE 2.34 – Types de périphériques

MMIO

Les registres des contrôleurs de ces périphériques d'entrées/sorties mappés en espace mémoire sont directement accessibles dans l'espace d'adressage du processeur. Le transfert de données entre les registres du CPU et ceux de ce type de contrôleurs s'effectue strictement de la même façon que celui de données avec la mémoire centrale. Par la simplicité d'accès à leurs registres, ce type de contrôleurs équipent la majorité des µP et µC.

La figure ci-dessus (figure 2.35) représente l'interface d'un périphérique élémentaire. Celui-ci dispose d'un registre d'état, d'un registre de contrôle et de deux registres pour le transfert de données (rx et tx). Tous les registres ont une taille fixe de 32 bits. Pour l'exemple, le périphérique est placé en mémoire, adresses `0x4000_1000` à `0x4000_1027`.

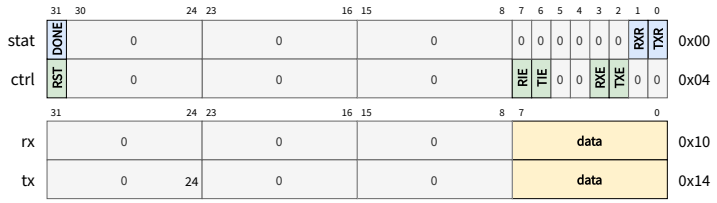


FIGURE 2.35 – Périphérique

La description d'un périphérique en langage de programmation se réalise généralement en C. Cette description passe par la déclaration des bits et champs contenus dans les différents registres ainsi que d'une structure contenant tous les registres comme membre de cette structure. Pour rappel, l'emplacement en mémoire des membres d'une structure correspond à l'ordre des déclarations et à la taille de chaque membre. Il est important de bien contrôler l'offset de chacun des membres et de rajouter des "padding" si nécessaire.

```
#define STAT_TXR  (1 <<  0)
#define STAT_RXR  (1 <<  1)
#define STAT_DONE (1 << 31)

#define CTRL_TXE  (1 <<  2)
#define CTRL_RXE  (1 <<  3)
#define CTRL_TIE  (1 <<  6)
#define CTRL_RIE  (1 <<  7)
#define CTRL_RST  (1 << 31)

struct ctrl {
   uint32_t stat;
   uint32_t ctrl;
   uint32_t pad[2];
   uint32_t tx;
   uint32_t rx;
};
```

Un pointeur, initialisé avec l'adresse de base du périphérique en mémoire, donne accès aux registres du contrôleur. La directive **`volatile`** est indispensable afin d'éviter toute optimisation du compilateur. Il est en effet essentiel que chaque accès aux registres du contrôleur soit effectué et non optimisé. Il est ensuite possible de piloter le périphérique. Dans l'exemple ci-dessous, le périphérique est d'abord initialisé, puis le code attend que l'initialisation se termine.

```
static volatile struct ctrl* io = (struct ctrl*)0x40001000;

io->ctrl = CTRL_RST;
while ((io->stat & STAT_DONE) == 0);
```

PMIO

Les registres des contrôleurs de ces périphériques d'entrées/sorties mappés dans l'espace des entrées/sorties (I/O), ne sont accessibles que dans un espace d'adressage distinct de celui de la mémoire centrale. Des instructions spécifiques (par exemple `IN` et `OUT` pour le jeu d'instructions x86) permettent d'accéder les registres de ces contrôleurs et de manipuler ainsi leur contenu. Ce type de contrôleurs se trouvent principalement sur des processeurs de l'architecture `x86` ou `x86_64`.

Coprocesseurs

Les processeurs intègrent certains de leurs périphériques internes en tant que coprocesseurs. Par exemple les processeurs ARM du profil A interfacent les registres des contrôleurs de leurs mémoires caches L1 et L2, ainsi que de la MMU via un coprocesseur, le coprocesseur 15 (CP15). Dans de tels cas, le CPU dispose d'instructions spécialisées pour accéder ces registres (sur ARM, les instructions `MCR` et `MRC`).

2.10 Placement d'un programme en mémoire

Pour qu'un µP puisse exécuter un programme informatique, il est indispensable de le charger préalablement dans la mémoire centrale. Son emplacement dans cette mémoire dépend naturellement fortement de l'architecture du système, du type de mémoires et de leur capacité. Un µP équipé d'une unité de gestion ou de protection de la mémoire (MMU/MPU - *Memory Management Unit / Memory Protecting Unit*) ainsi que le type du système d'exploitation mis en oeuvre ont également leur importance.

2.10.1 Layout d'un programme

Afin de permettre une gestion optimale de l'espace mémoire, le code et les différents types de données sont regroupés dans des sections distinctes (figure 2.36).

Un programme distingue au minimum quatre sections principales :

- La section **.text**, qui regroupe toutes les instructions du programme, le code.
- La section **.rodata**, qui regroupe toutes les constantes ou variables du programme avec seulement un accès en lecture (non modifiable).
- La section **.data**, qui regroupe toutes les données ou variables globales du programme qui sont initialisées avec une valeur différente de zéro (0).

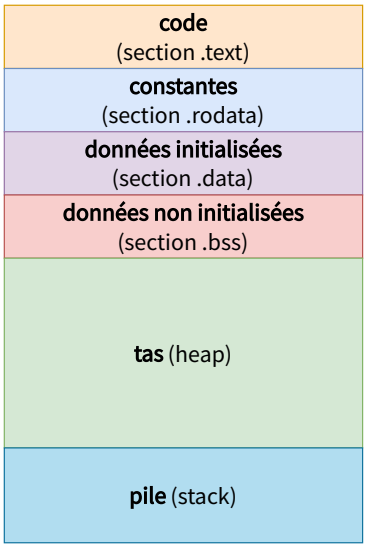


FIGURE 2.36 – Layout d'un programme en mémoire

— La section **.bss**, qui regroupe toutes les données ou variables globales du programme qui sont soit initialisée à zéro ou nul (0), ainsi que les variables non initialisées, lesquelles seront mises à zéro lors du lancement du programme.

La taille de ces quatre sections est déterminée lors de la génération du programme et découle directement du développement de l'application, c'est-à-dire du nombre de lignes de code ainsi que du nombre de constantes et de variables globales initialisées ou non.

A ces quatre sections, deux sections sont encore nécessaires pour l'exécution du programme par le µP :

— La **pile** (*stack*) permet de stocker le contexte d'une fonction (adresse de retour, sauvegarde de certains registres du µP, etc.), les arguments ou paramètres d'appel de la fonction ainsi les variables locales de la fonction.

— Le **tas** (*heap*) permet à l'application d'allouer dynamiquement des zones mémoires pour le stockage d'information ou la création d'objets.

La taille de la pile et du tas dépendent fortement du type de système déployé. Si le système met en oeuvre un système d'exploitation (OS - *Operating System*) ou d'un OS temps réel (RTOS - *Real Time Operating System*), ces tailles sont définies par l'OS ou le RTOS. Dans le cas d'application `bare metal`, c'est-à-dire sans OS/RTOS, la taille de la pile est généralement fixée statiquement lors de la génération du programme. Quant au tas, il occupe l'espace restant entre le sommet de la pile et le sommet de la section .bss.

L'éditeur de liens (*linker*) effectue le placement du code et des données dans ces différentes sections.

Le fichier ci-dessous présente un exemple élémentaire d'un fichier de configuration de l'éditeur de liens.

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)

MEMORY {
    DDR0 : o = 0x80000000, l = 0x00f00000
}

SECTIONS {
    .text : {
        *(.text_startup)
        *(.text*)
    } > DDR0

    .rodata : {
        . = ALIGN(0x100);
        *(.rodata*)
    } >DDR0

    .data : {
        . = ALIGN(0x100);
        *(.data*)
    } >DDR0

    .bss (NOLOAD): {
        . = ALIGN (0x100);
        __bss_start__ = . ;
        *(.bss*)
        *(COMMON)
        __bss_end__ = .;
    } >DDR0

    _heap_bottom  = ALIGN(0x100);
    _stack_bottom = ORIGIN(DDR0) + LENGTH(DDR0) - 0x80000;
    _stack_top    = ORIGIN(DDR0) + LENGTH(DDR0);
}
```

L'organisation du code et des données en différentes sections permettent également d'augmenter la robustesse et la fiabilité du logiciel. En effet, si le processeur dispose d'une MMU ou d'une MPU, il devient possible d'éviter des accès intempestifs dans des zones mémoires inappropriées, par exemple exécution de code dans une zone de données, modification du code, etc.

2.10.2 Organisation pour des systèmes embarqués

Les systèmes embarqués disposent généralement d'un µP performant équipé d'une MMU et de suffisamment de mémoire Flash et RAM (plusieurs centaines de MiB, voire même de GiB). Les mémoires Flash sont principalement de type NAND, tandis que la RAM est de type DRAM. Ces ressources autorisent la mise en oeuvre d'un OS tel que GNU/Linux et d'un système de fichiers, par exemple `ext4`, sur la Flash permettant ainsi d'exécuter plusieurs applications sous forme de processus indépendants.

La mémoire Flash sert à stocker de façon permanente les applications et les données de configuration. Lors du lancement d'une application, l'OS copie l'application (contenu des sections .text, .rodata et .data) de la Flash dans la RAM (figure 2.37). Cette opération terminée, le µP pourra finalement l'exécuter.

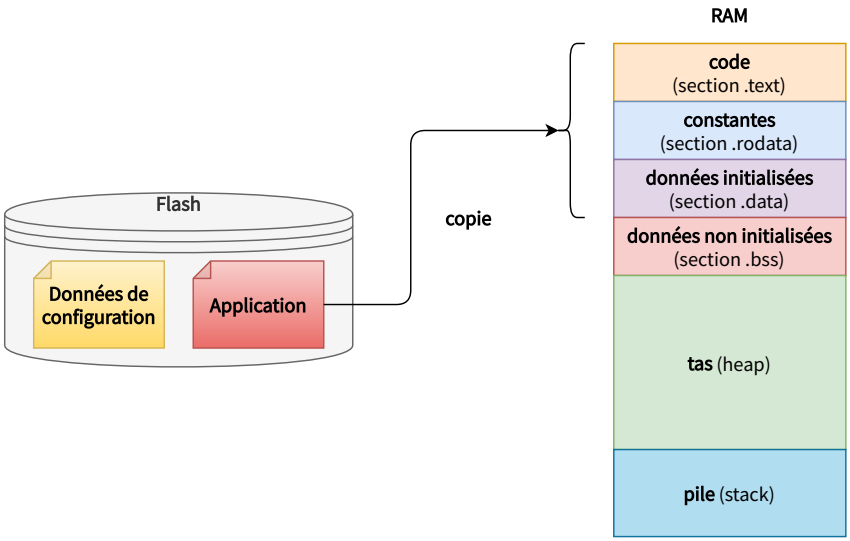


FIGURE 2.37 – Organisation sur des systèmes embarqués

2.10.3 Organisation pour des systèmes sur puce (SoC)

Les systèmes sur puce (SoC) ne disposent généralement que de très peu de mémoire Flash et RAM (quelques dizaines de KiB). Sur les SoC, la Flash ainsi que la RAM sont intégrées directement dans la puce du µP. Pour ce faire, elles sont de type NOR pour la Flash et de type SRAM pour la RAM. Ces ressources limitées n'autorisent pas la mise en oeuvre d'OS tel que GNU/Linux. Par contre, le déploiement de RTOS, tel que MBedOS, FreeRTOS et bien d'autres, est très courant. Ces RTOS permettent d'exécuter une seule et unique application, mais avec plusieurs tâches (*threads*).

La mémoire Flash sert à stocker de façon permanente les applications et les données de configuration. Les Flash de type NOR offrant un accès direct et aléatoire au contenu, le µP peut ainsi exécuter directe-

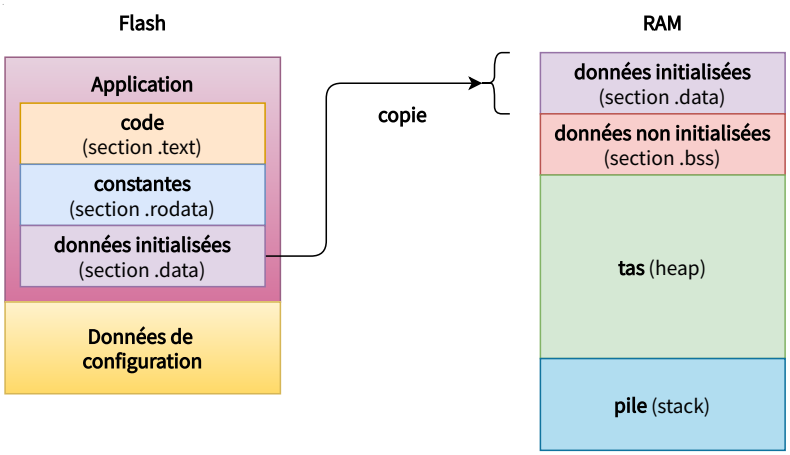


FIGURE 2.38 – Organisation sur des systèmes sur puce

ment les instructions de l'application de la mémoire Flash sans devoir préalablement la copier dans la RAM. Pour cela, le code binaire de l'application est stocké directement dans les blocs de la Flash (section .text). Il en va de même des constantes (section .rodata) et des données avec une valeur initiale (section .data). Lors du lancement de l'application, le contenu de la section .data est copié de la Flash dans la RAM (figure 2.38). Cette opération terminée, le µP poursuit l'exécution du programme.

2.11 Exercices

Quelques exercices pour assimiler la matière de ce chapitre.

2.11.1 Exercice 1 : Composants d'un CPU

Dessinez l'architecture générale de l'unité centrale de traitement avec accès au code et aux données. Quelle est la fonction de chacun de ses composants ?

2.11.2 Exercice 2 : Traitement de l'information

Citez les 3 cycles principaux du traitement de l'information.
Quelles opérations sont effectuées dans chacun de ces cycles ?

2.11.3 Exercice 3 : Classification de Flynn

Quelles sont les caractéristiques principales des quatre architectures selon la classification de Flynn ?

2.11.4 Exercice 4 : Performances d'un µP

Quels facteurs influencent les performances d'un processeur (Iron Law) ?
Comment peut-on les influencer ?

2.11.5 Exercice 5 : Représentation de l'information

Comment se représentent des données en mémoire ?
Quelle influence a l'alignement des données en mémoire ?
Que signifie boutisme ?

2.11.6 Exercice 6 : Placement des données en mémoire

Pour la structure C ci-dessous, donnez la représentation en hexadécimale de son instanciation pour une mémoire organisée en octet.

```
1 struct S {
2     int32_t  a;
3     int8_t   b;
4     uint64_t c;
5     uint16_t d;
6     char*    e;
7 } s = { .a=1, .b=-2, .c=0xabcdef12345678, .d=0777, .e=0x20004000, };
```

2.11.7 Exercice 7 : Caractéristiques des mémoires

Quels types de mémoire centrale existe-t-il et quelles sont leurs principales caractéristiques ?
Quelles différences existe-t-il entre une SRAM et une DRAM et où sont-elles utilisées ?
Quelles différences existe-t-il entre une mémoire Flash NOR et une NAND et où sont-elles utilisées ?

2.11.8 Exercice 8 : Architecture d'un périphérique d'entrées/sorties

Dessinez l'architecture d'un périphérique d'entrées/sorties.
Quels sont les registres principaux d'un contrôleur ?

Quels types d'interfaces existe-t-il avec le CPU ?

2.11.9 Exercice 9 : Pilote de périphérique

Pour s'assurer qu'une application ne reste pas dans une boucle de traitement particulier bloquant le reste du système, les µP modernes disposent de chiens de garde (WDOG - *Watchdog Timer*). Ces chiens de garde sont des minuteries (*timer*) qui généralement redémarrent le système s'ils ne sont pas réarmés après un certain laps de temps.

Les processeurs d'applications i.MX7ULP fabriqués par NXP sont équipés de trois de ces chiens de garde. La figure ci-dessus (figure 2.39) présente sommairement les registres de ces contrôleurs et donne leur emplacement en mémoire.

WDOG Memory Map

WDOG0 base address: 4102_5000h
WDOG1 base address: 403D_0000h
WDOG2 base address: 4043_0000h

Offset	Register	Width (in bits)	Access	Reset value
0h	Watchdog Control and Status Register (CS)	32	RW	0000_0980h
4h	Watchdog Counter Register (CNT)	32	RW	0000_0000h
8h	Watchdog Timeout Value Register (TOVAL)	32	RW	0000_0400h
Ch	Watchdog Windows Register (WIN)	32	RW	0000_0000h

Watchdog Control and Status Register (CS)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0																WIN	FLG	CMD32EN	PRES	ULK	RCS	CLK		EN	INT	UPDATE	TST		DBG	WAIT	STOP
W																		W1C														
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0

FIGURE 2.39 – i.MX7ULP WDOG

Questions :

a. Définissez l'interface C/C++ (structure, constantes, etc.) pour un WDOG
b. Déclarez le tableau permettant d'accéder aux registres des 3 WDOG et initialisez-le
c. Ecrivez un code pour mettre à 0 le bit EN (bit 7) du 2^e WDOG, ceci sans modifier les autres bits du registre
d. Ecrivez un test permettant d'attendre jusqu'à ce que le bit ULK (bit 11) du 2^e WDOG soit mis à 1
e. Ecrivez un code pour effacer (clear) le bit FLG (bit 14) du 2^e WDOG sans modifier les autres bits du registre

Remarque :

- Les registres de ces contrôleurs ne peut être accédés qu'avec des mots de 32 bits
- `W1C` indique qu'il faut écrire un 1 pour effacer (clear) le bit
- Seuls les bits utilisés pour solutionner les questions doivent être déclarés (soit les bits `FLG`, `ULK` et `EN`)

2.11.10 Exercice 10 : Layout d'un programme

Quelles sont les sections principales d'une application ?
Quelle est son organisation pour un système embarqué ?
Quelle est son organisation pour un système sur puce (SoC) ?