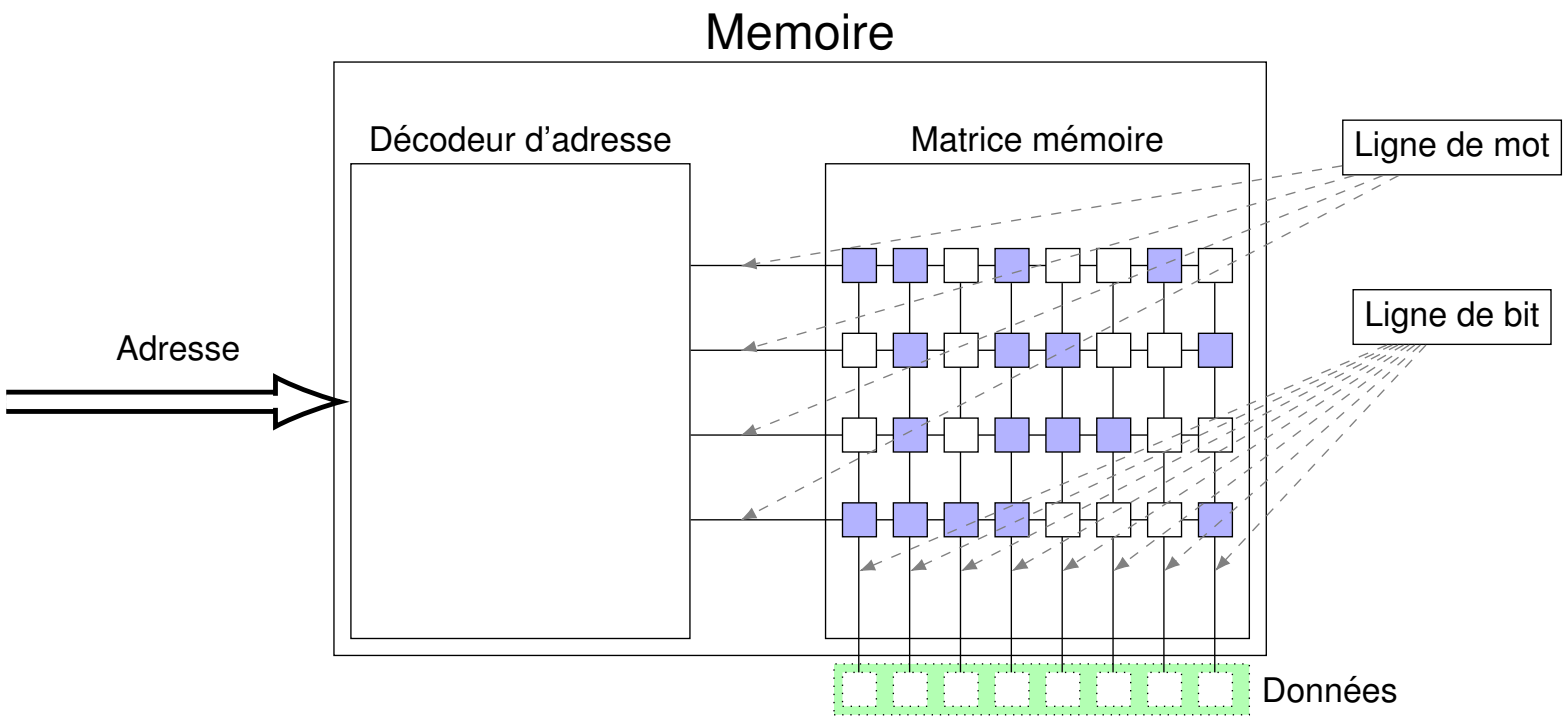
Mémoire physique

Vincent Migliore

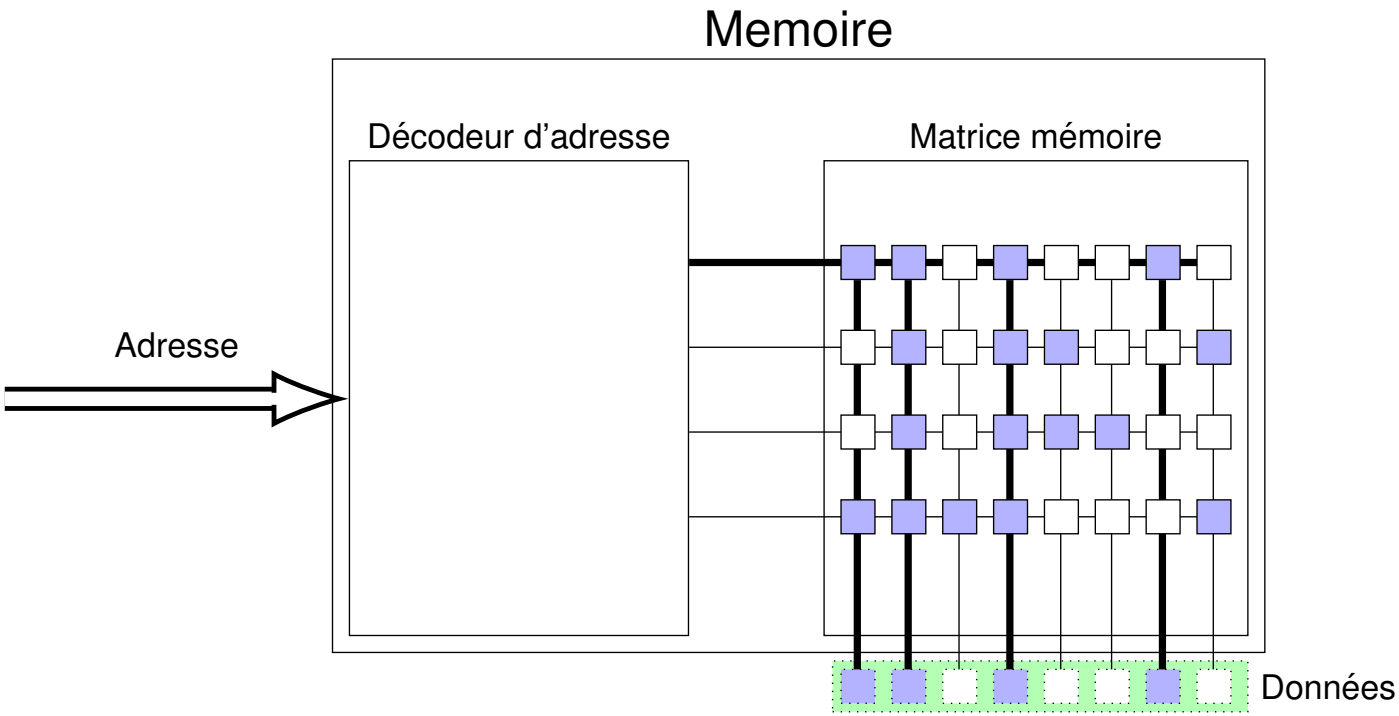
vincent.migliore@insa-toulouse.fr

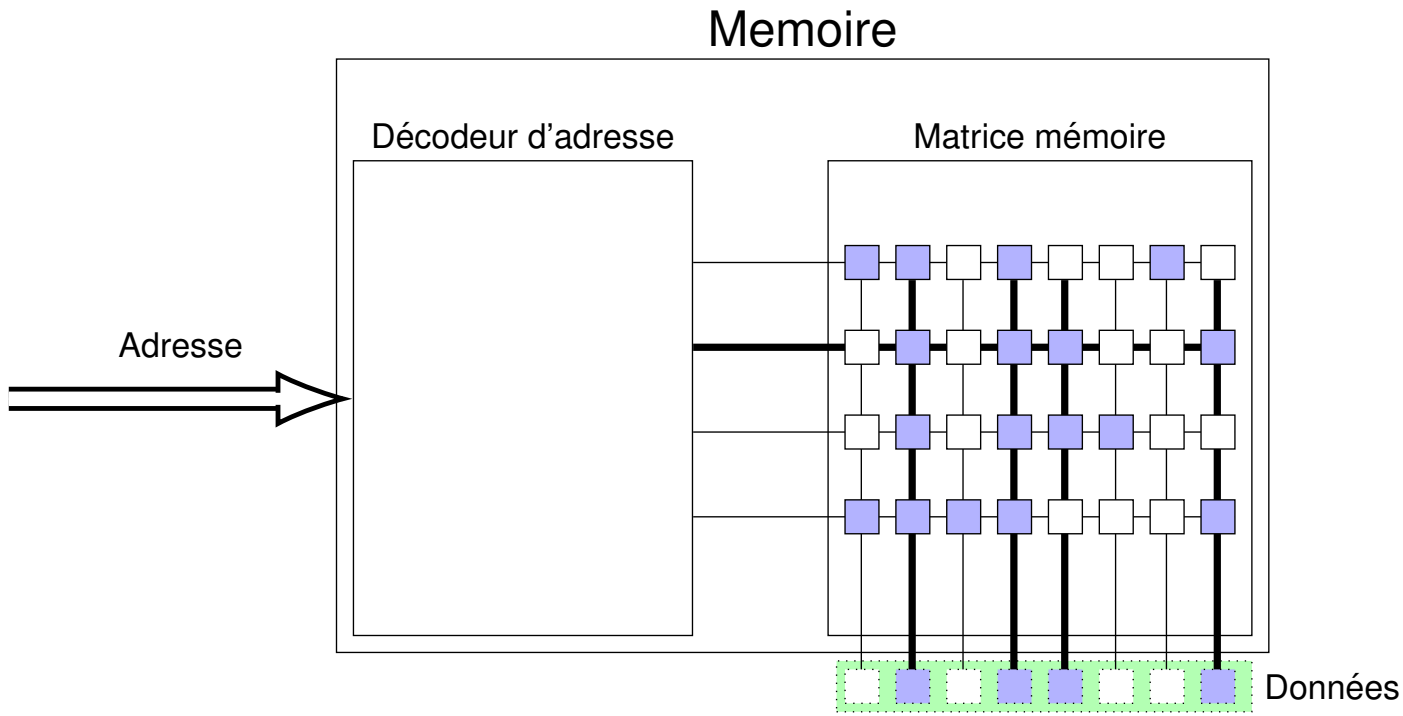
Fonctionnement d'une mémoire

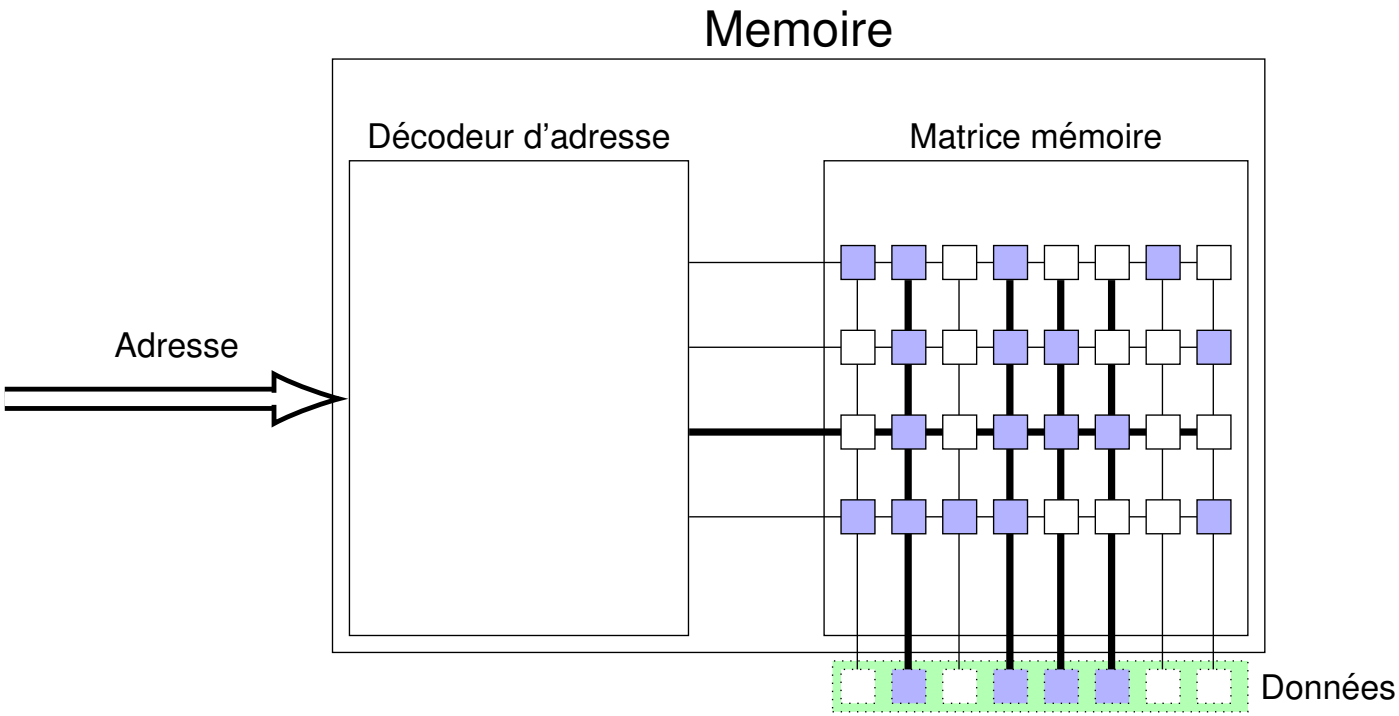


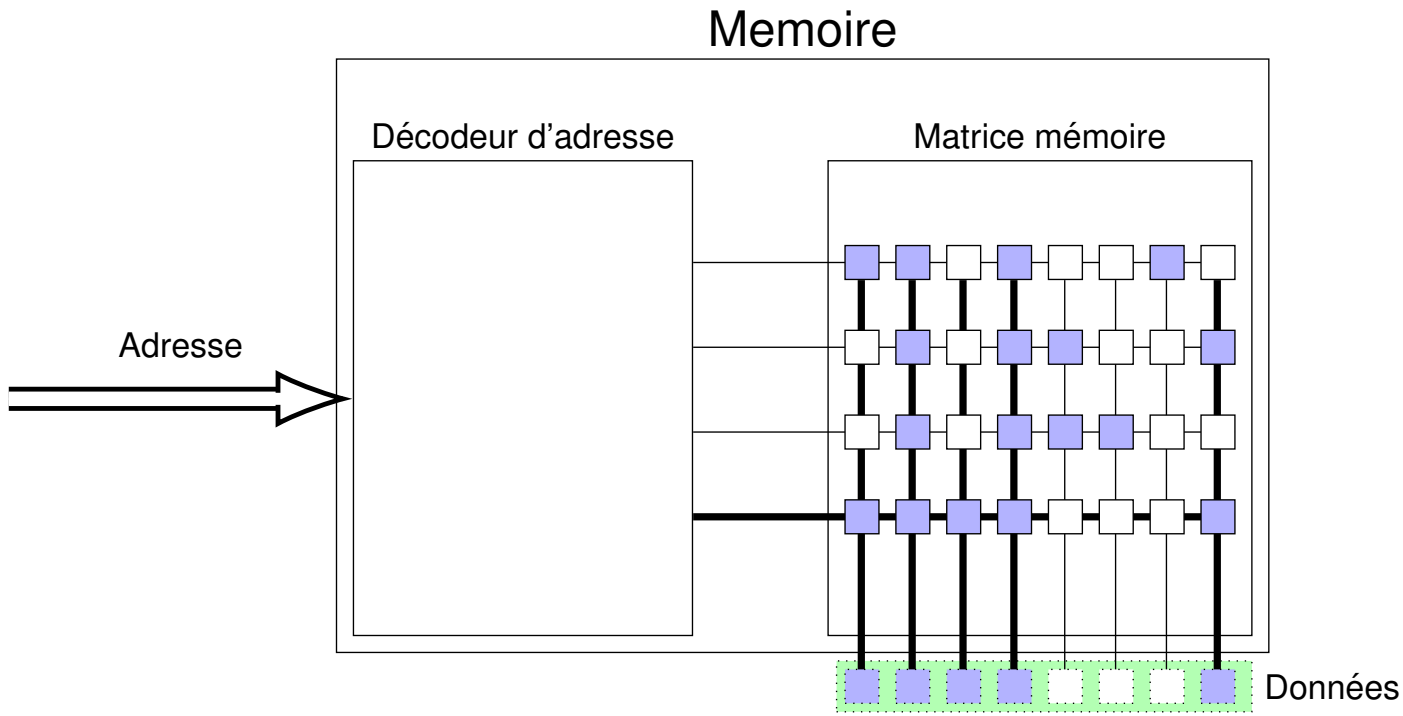
Entrées/sorties et structure interne

Une mémoire est composée d'une entrée d'adresse et une sortie de donnée. Une adresse est un entier représentant le numéro de ligne à lire dans la mémoire interne de la mémoire nommée matrice mémoire. Le décodeur d'adresse active la ligne demandée et désactive les autres, ce qui a pour conséquence de copier la valeur des bits associés à cette adresse dans la sortie donnée.

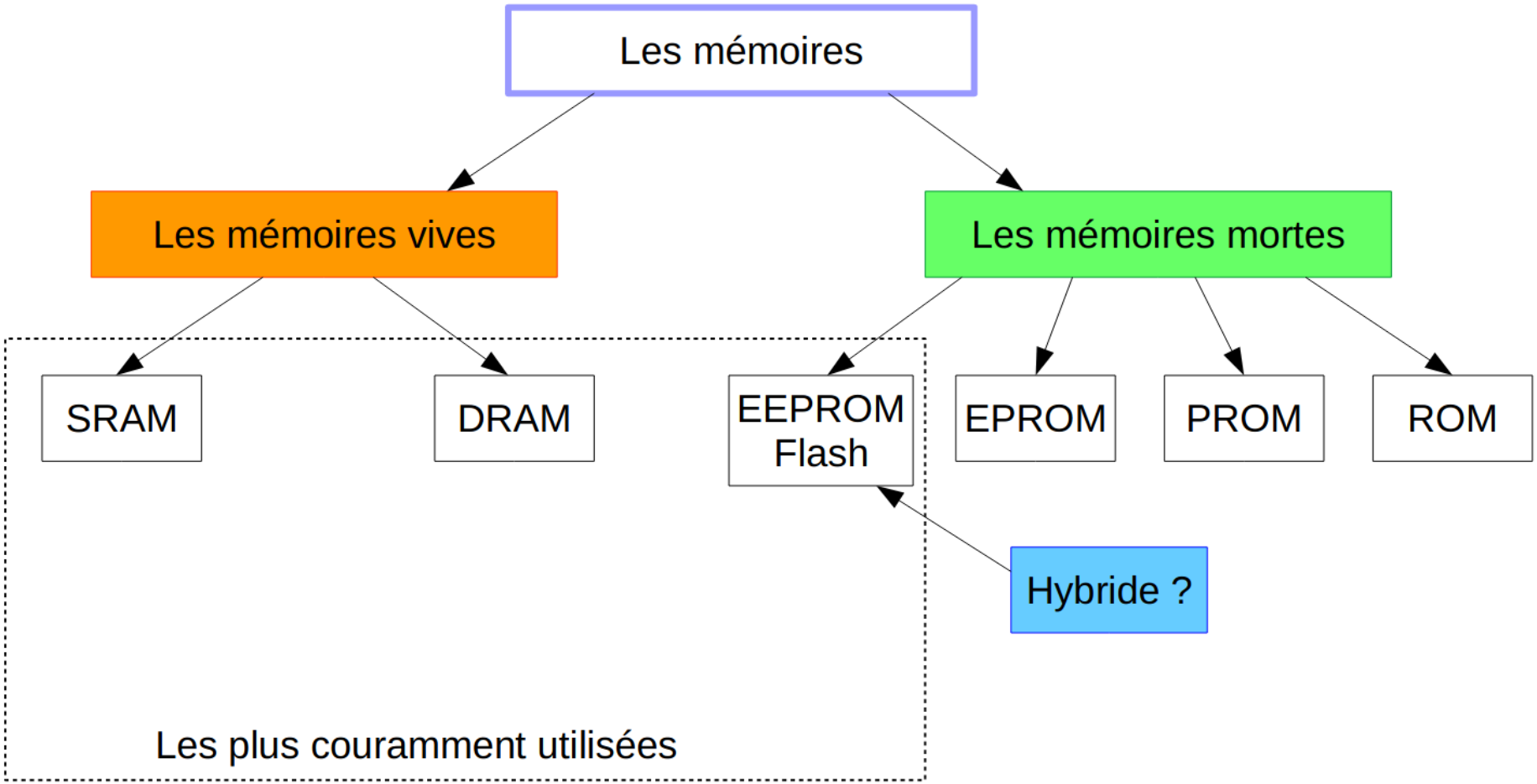

Memoire
Décodeur d'adresse
Matrice mémoire
Adresse
Données


Memoire
Décodeur d'adresse
Matrice mémoire
Adresse
Données


Memoire
Décodeur d'adresse
Matrice mémoire
Adresse
Données


Memoire
Décodeur d'adresse
Matrice mémoire
Adresse
Données

Technologies de mémoire



Définition

Une mémoire ROM (Read Only Memory) est une mémoire, qui historiquement, est accessible uniquement en lecture. Les technologies ont évoluées pour permettre une réécriture de la mémoire. C'est une mémoire dont les données persistent après extinction de son alimentation, c'est donc une mémoire **non volatile**.

Mémoire Flash

Les technologies mémoires des composants modernes sont principalement de type Flash. Ce sont des mémoires ROM réinscriptibles électriquement permettant ainsi sa réécriture directement depuis l'ordinateur.

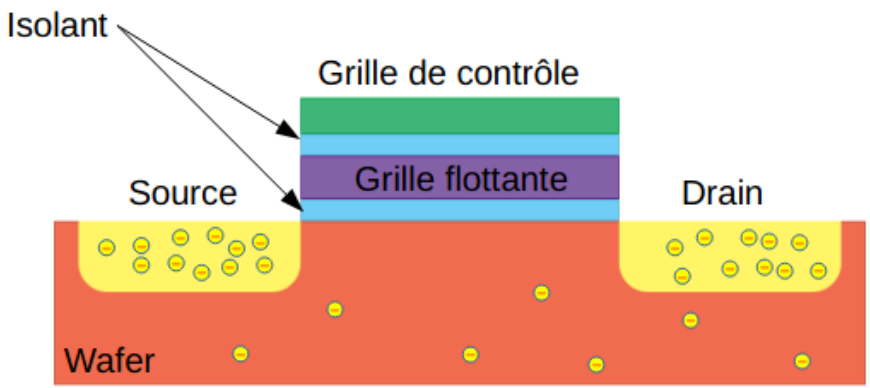


Figure: Représentation d'une mémoire de type Flash. Le stockage de l'information se fait par le stockage dans la grille flottante d'électrons issus de la zone orange (faiblement dopée en électron).

Définition

Une mémoire ROM (Read Only Memory) est une mémoire, qui historiquement, est accessible uniquement en lecture. Les technologies ont évoluées pour permettre une réécriture de la mémoire. C'est une mémoire dont les données persistent après extinction de son alimentation, c'est donc une mémoire **non volatile**.

Exemples d'utilisation de ROMs

Les mémoires de masse des ordinateurs, les clés USB, cartes SD, le bootloader (composant contenant le programme de démarrage de l'ordinateur), ...

Remarque

Les mémoires de masses de technologie précédente utilisaient un disque magnétisé pour stocker l'information, c'est pour cela que l'on nomme par abus de langage la mémoire de masse le disque dur.

Définition

Une mémoire RAM (Random Access Memory) est une mémoire qui est accessible indifféremment en lecture et en écriture. Il existe 2 grandes familles : les Static RAMs (SRAM) constitués de 6 transistors montés en tête-beche, et les Dynamic RAMs (DRAM) constitués d'un condensateur en série avec un transistor. C'est une mémoire dont les données s'effacent après extinction de son alimentation, c'est donc une mémoire **volatile**.

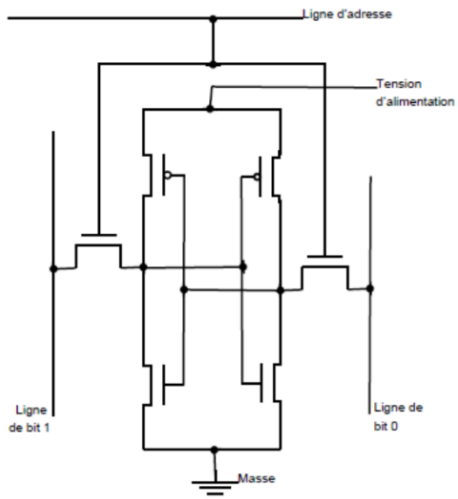


Figure: Représentation d'une cellule mémoire SRAM.

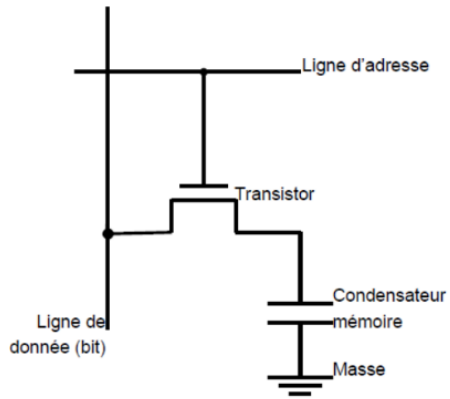


Figure: Représentation d'une cellule mémoire DRAM.

Définition

Une mémoire RAM (Random Access Memory) est une mémoire qui est accessible indifféremment en lecture et en écriture. Il existe 2 grandes familles : les Static RAMs (SRAM) constitués de 6 transistors montés en tête-beche, et les Dynamic RAMs (DRAM) constitués d'un condensateur en série avec un transistor. C'est une mémoire dont les données s'effacent après extinction de son alimentation, c'est donc une mémoire **volatile**.

Comparaison des deux technologies

Caractéristique	SRAM	DRAM
Surface d'une cellule	Plus grande (6 transistors)	Plus petite (1 transistor et 1 condensateur)
Vitesse d'accès	Plus rapide (pas d'élément capacitif)	Plus lent (élément capacitif)

Exemples d'utilisation de RAMs

Les principaux composants utilisant des RAMs sont les registres du processeur, les caches et la mémoire centrale. La technologie diffère pour chaque cas :

Composant	Type de technologie
Registres	SRAM (optimisation de la vitesse)
Caches	SRAM (optimisation de la vitesse)
Mémoire centrale	DRAM (optimisation de la surface) sauf gamme micro-contrôleurs car le besoin en mémoire est plus faible (quelques Mo suffisent)

Quelques mots sur les unités utilisées dans les mémoires

Convention sur les tailles des données

Préfixes décimaux		
Nom	Symbole	Valeur
Octet	o	1 octet
Kilo octet	Ko	10^4 octets
Mega octet	Mo	10^8 octets
Giga octet	Go	10^{12} octets
Tera octet	To	10^{16} octets

Préfixes Binaires		
Nom	Symbole	Valeur
Octet	o	1 octet
Kibi octet	Kio	2^{10} octets
Mébi octet	Mio	2^{20} octets
Gibi octet	Gio	2^{30} octets
Tébi octet	Tio	2^{40} octets

Composants Mémoires		
Nom	Symbole	Valeur
Octet	o	1 octet
Kilo octet	Ko	2^{10} octets
Mega octet	Mo	2^{20} octets
Giga octet	Go	2^{30} octets
Tera octet	To	2^{40} octets

Conventions existantes

Il existe 3 conventions sur les tailles des données, à savoir les préfixes décimaux dont le passage au préfixe suivant se fait en multipliant la valeur par 1000, les préfixes binaires dont le passage au préfixe suivant se fait en multipliant la valeur par 1024, et la convention pour les composants mémoire qui est une convention hybride : on utilise le nom des préfixes décimaux tout en utilisant les valeurs des préfixes binaires. Ainsi, un kilo octet correspondra à 1024 octets.

Exemple d'utilisation des différentes conventions

Pour les tailles de mémoire centrale, de cache ou de composants mémoire internes aux système, c'est la convention hybride qui est utilisée. Pour les mémoires centrales, il faut faire attention, c'est en principe la convention des préfixes décimaux qui est en général employé, car cela permet de faire gonfler artificiellement les chiffres : 1 Go en utilisant les préfixes décimaux correspond à 1 milliard d'octets, alors qu'il correspond à 1.073 milliard d'octets pour la convention des composants mémoire.

Adressage de la mémoire

Chaque adresse correspond à un ensemble de 8 cellules mémoires, soit un 1 octet.

Les données que peut traiter un processeur ne sont pas nécessairement sur 1 octet, ainsi on parle souvent de mots.

octet	(byte)	=	8 bits
mot	(word)	=	16 bits
double mot	(dword)	=	32 bits
quadruple mot	(qword)	=	64 bits

<table>
<tr><td colspan="8">qword</td></tr>
<tr><td colspan="4">dword</td><td colspan="4">dword</td></tr>
<tr><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td></tr>
<tr><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td></tr>
</table>

Un processeur moderne d'architecture 64 bits peut donc traiter des quadruples mots (qword), mais également des doubles mots, des mots et des octets.

Question

On suppose que l'on possède une mémoire de 2 Mo, calculez :

1. Le nombre d'adresses
2. Le nombre de bits nécéssaires pour adresser toute la mémoire

Question

On suppose que l'on possède une mémoire de 2 Mo, calculez :

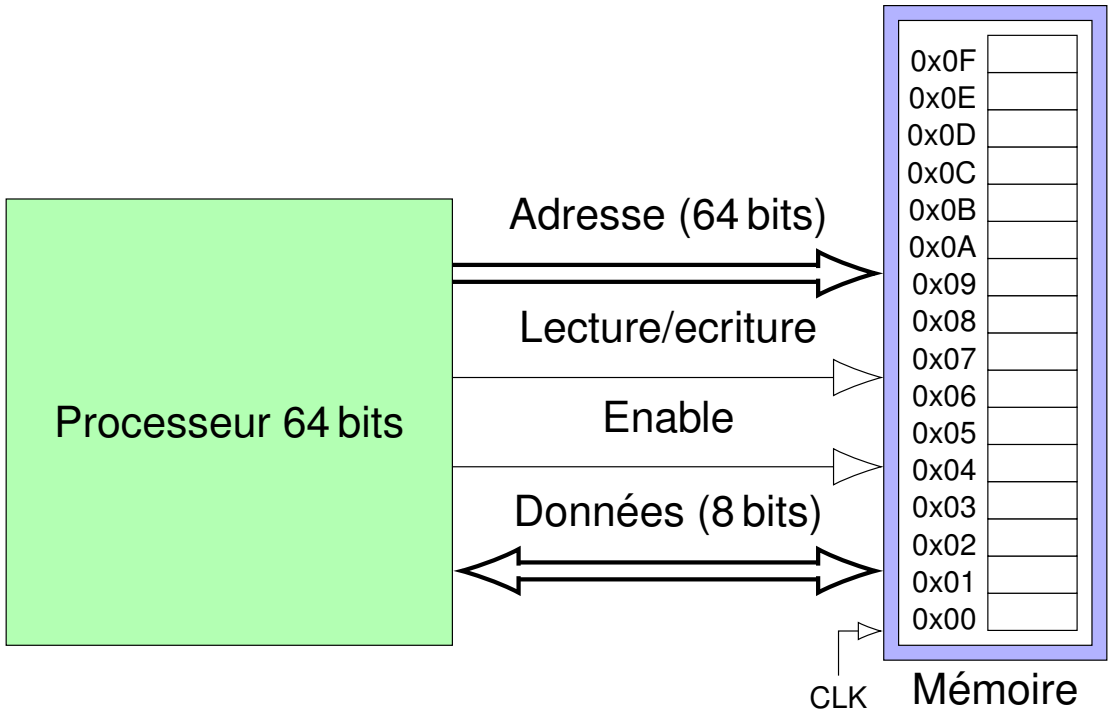
1. Le nombre d'adresses
   La mémoire est sur 2 Mo, chaque adresse est sur 1 octet
   $\rightarrow$ 2 Mo / 1 o = $2 \times 2^{20} / 1 = 2^{21}$ adresse.
2. Le nombre de bits nécéssaires pour adresser toute la mémoire

Question

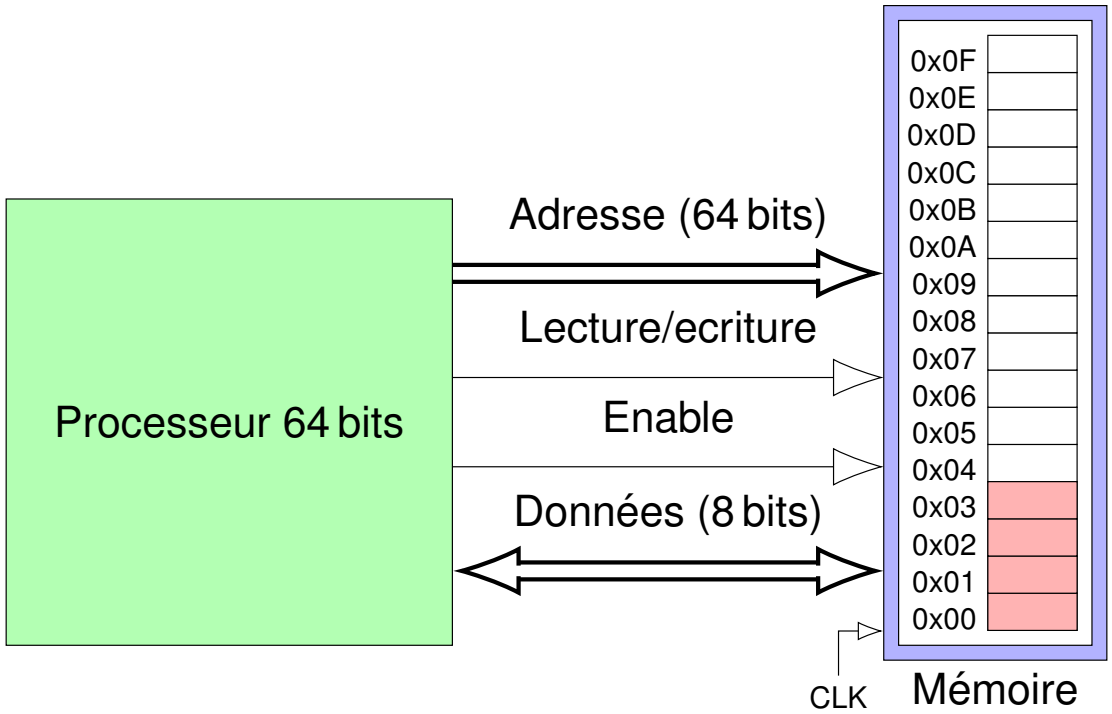
On suppose que l'on possède une mémoire de 2 Mo, calculez :

1. Le nombre d'adresses
   La mémoire est sur 2 Mo, chaque adresse est sur 1 octet
   $\rightarrow$ 2 Mo / 1 o = $2 \times 2^{20}/1 = 2^{21}$ adresses.
2. Le nombre de bits nécéssaires pour adresser toute la mémoire
   Nous avons 2^{21} adresses, il nous faut donc
   $\lceil \log_2 2^{21} \rceil$ bits = 21 bits.

Adressage de la mémoire

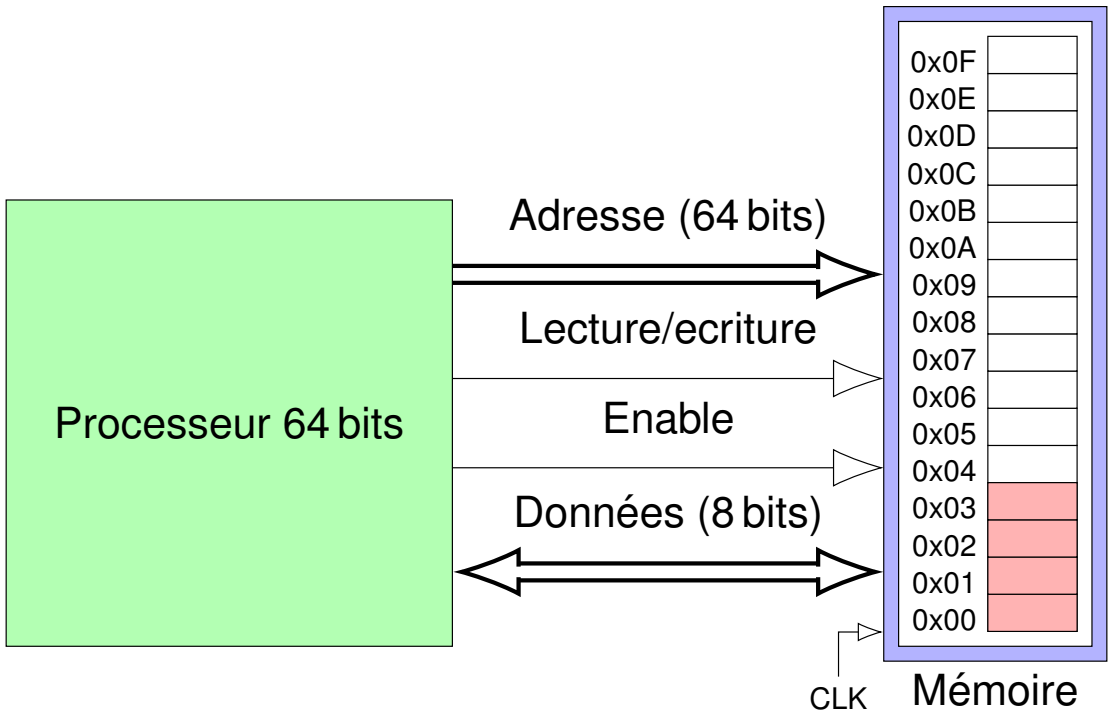


Les mémoires possèdent un certain nombre de broches qui permettent de lire et écrire ses cellules mémoires. Elles possèdent également un signal d'horloge, se sont des composants séquentiels. Chaque adresse pointe vers un octet, il faut donc plusieurs accès successifs pour lire de plusieurs octets. Nous allons faire l'hypothèse pour le moment que le bus de données est sur 8 bits (donc on peut lire/ecrire un seul octet à la fois), en principe, plusieurs composants mémoires sont positionnés en parallèle sur une même mémoire pour lire plusieurs octets à la fois.



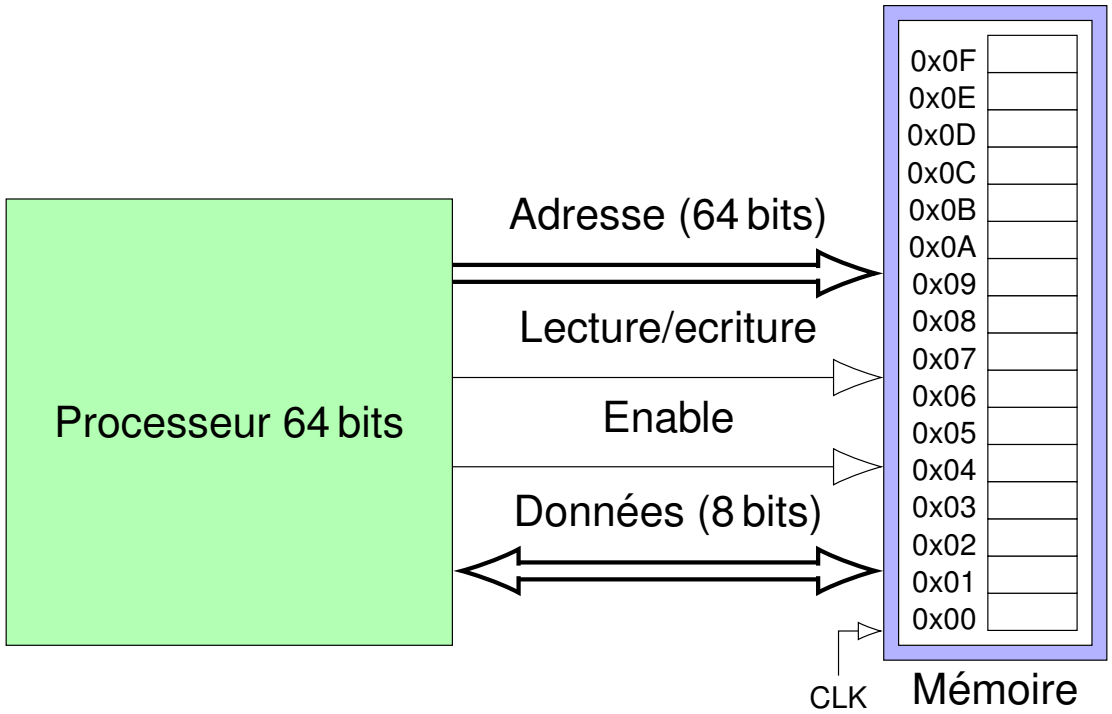
Question

Supposons que nous avons stocké un entier de 32 bits en mémoire (représenté en rouge). Quelles adresses et combien d'accès mémoire sont nécéssaires ?



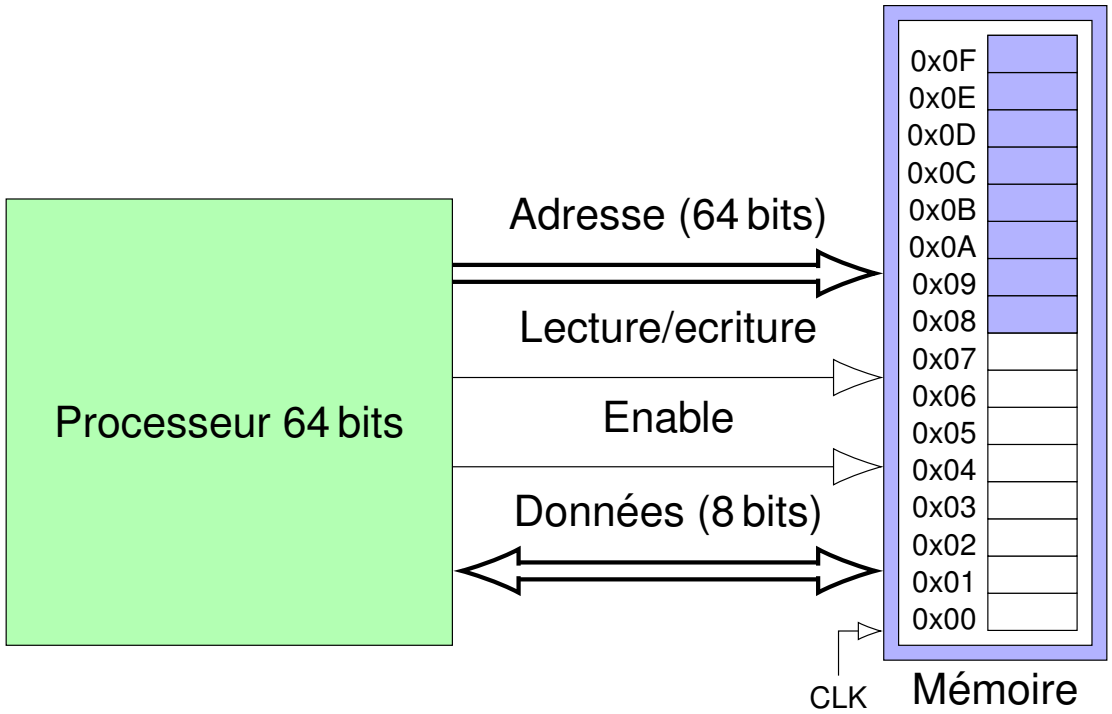
Question

Il faudra 4 accès mémoire, aux adresses 0x0, 0x1, 0x2, 0x3. Pour le premier octet, le contrôleur mémoire du processeur positionnera le bus d'adresse à 0x0, Enable à 1, et Lecture/Ecriture à 0 pour lire l'octet au prochain coup d'horloge.



Question

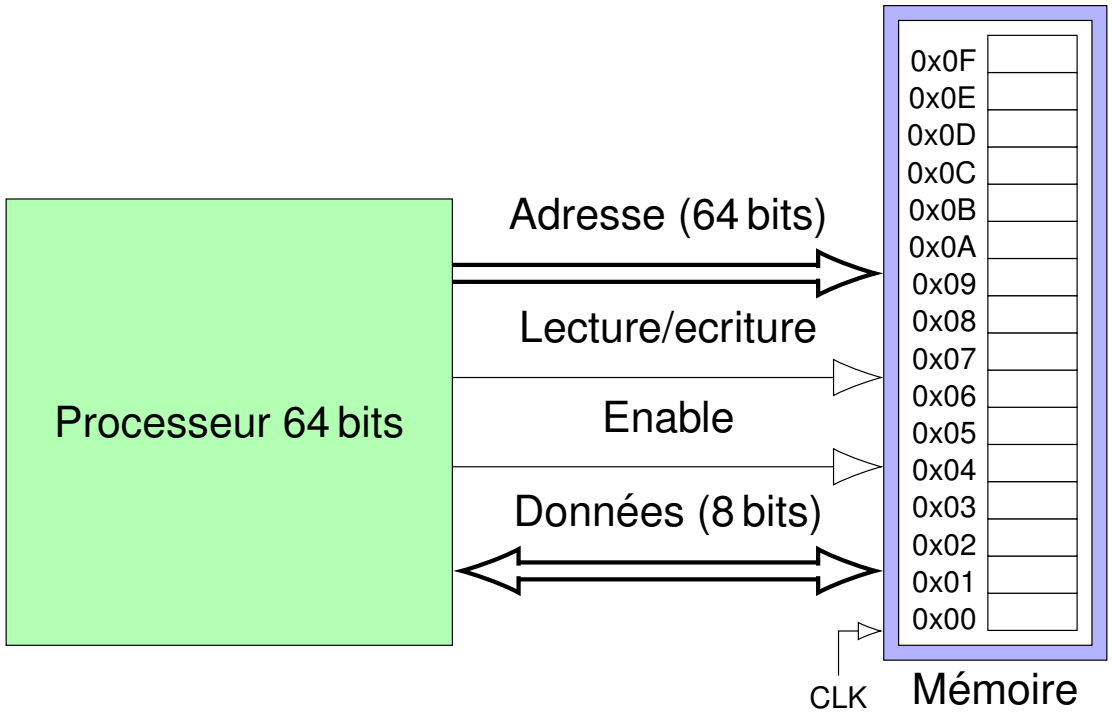
Autre question. Le processeur souhaite stocker une adresse à partir de l'adresse 0x8 pour la réutiliser plus tard. Combien d'accès à la mémoire sera-t-il nécessaire ?



Question

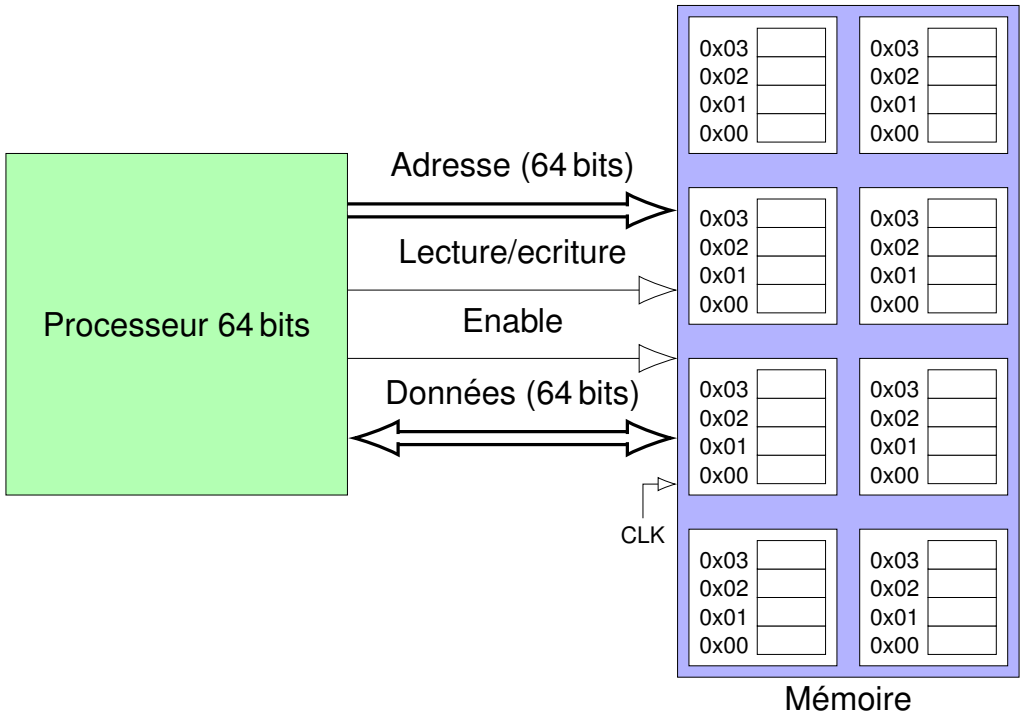
Il faudra 8 accès mémoire, aux adresses 0x8, 0x9, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF. Pour le premier octet, le contrôleur mémoire du processeur positionnera le bus d'adresse à 0x8, Enable à 1, et Lecture/Ecriture à 1 pour écrire l'octet au prochain coup d'horloge.

Adressage de la mémoire



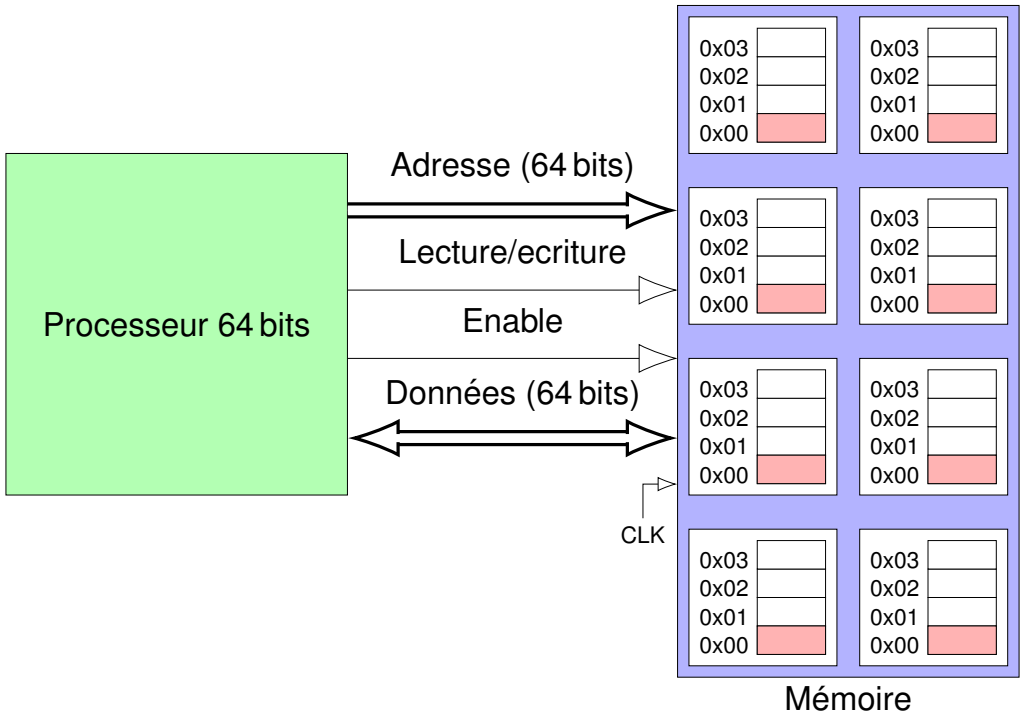
Compte tenu que le processeur traite des données sur 64 bits, un bus de données sur 8 bits est insuffisant.

Adressage de la mémoire



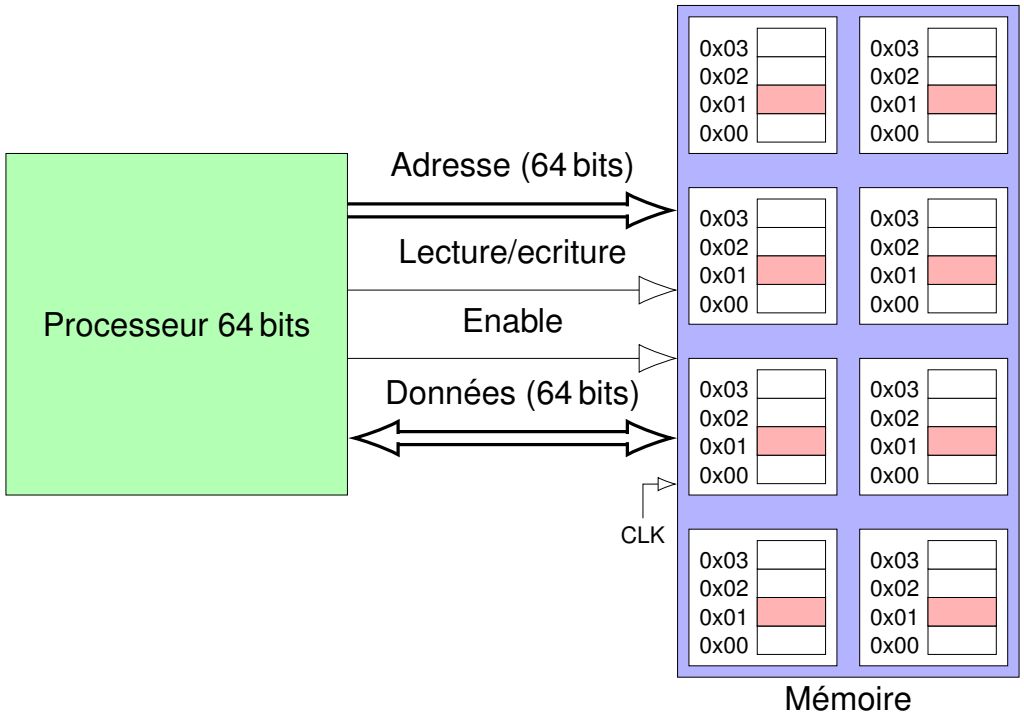
L'idée est d'utiliser plusieurs composants mémoires en parallèle. Sur les mémoires DDR moderne, elles sont ou nombre de 8, permettant d'obtenir un bus de données sur 64 bits. Il faut garder en tête que l'adressage se fait toujours à l'octet, mais que plus de données sont lues/écrites à chaque opération.

Adressage de la mémoire



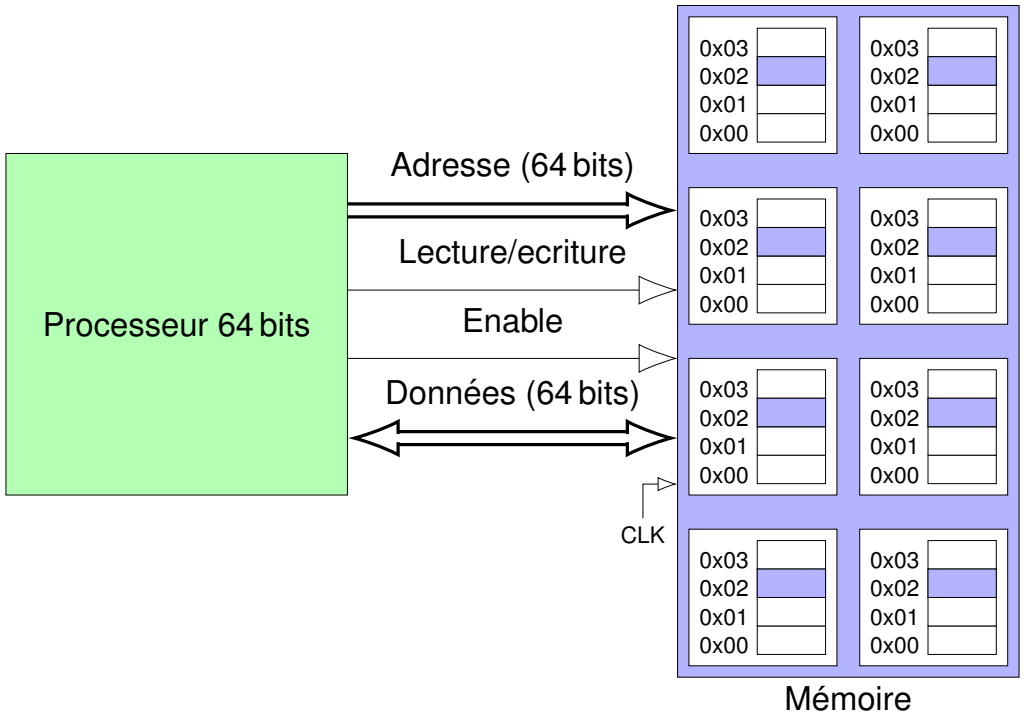
Cette fois-ci, lorsque l'adresse 0x0 sera demandée, chaque cellule mémoire sera accédée en parallèle.

Adressage de la mémoire

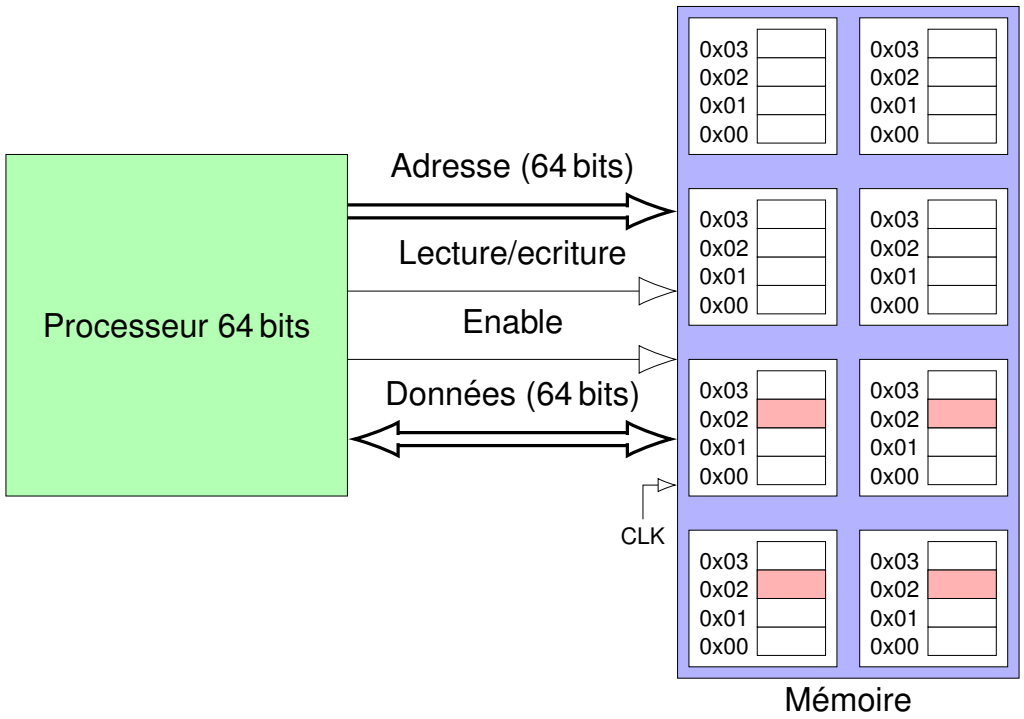


De même lors de l'accès à l'adresse 0x8, qui se traduira par l'accès à l'adresse 0x2 de chaque composant mémoire.

Adressage de la mémoire

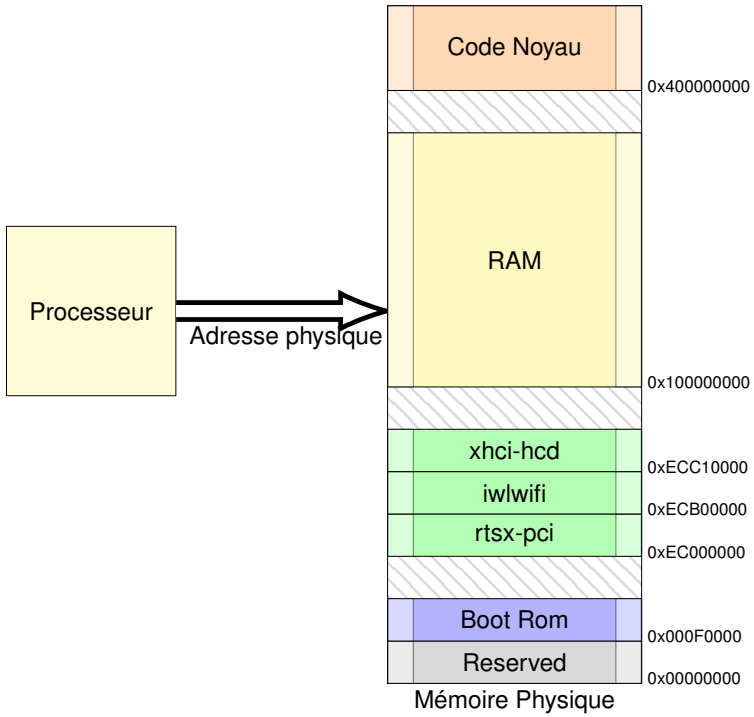


Un dernier exemple dans le cas où le processeur veut stocker 64 bits à l'adresse 0xC.



Remarque

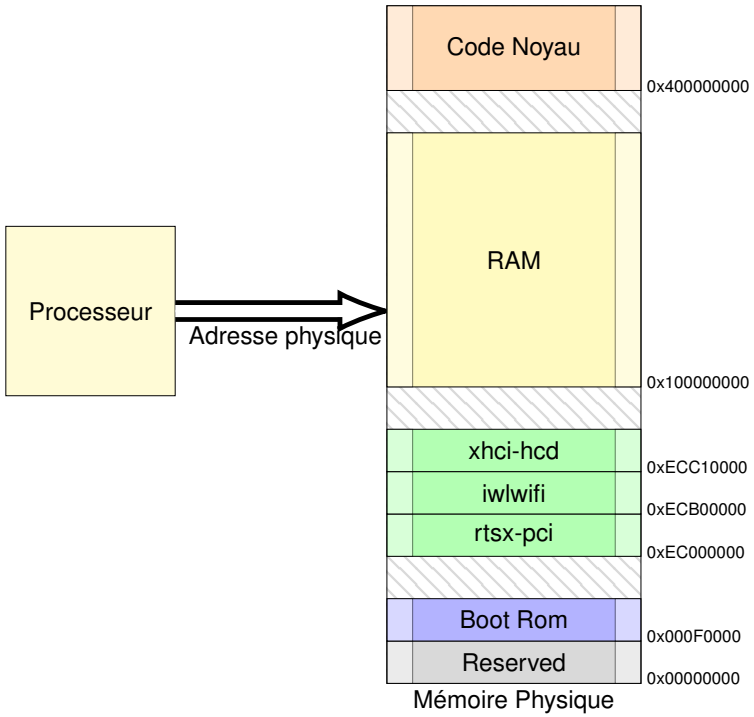
Il est tout à fait possible de stocker des données qui font moins que 64 bits, via l'utilisation d'une broche non représentée ici qui s'appelle "masque". Seule la partie non masquée sera écrite en mémoire.

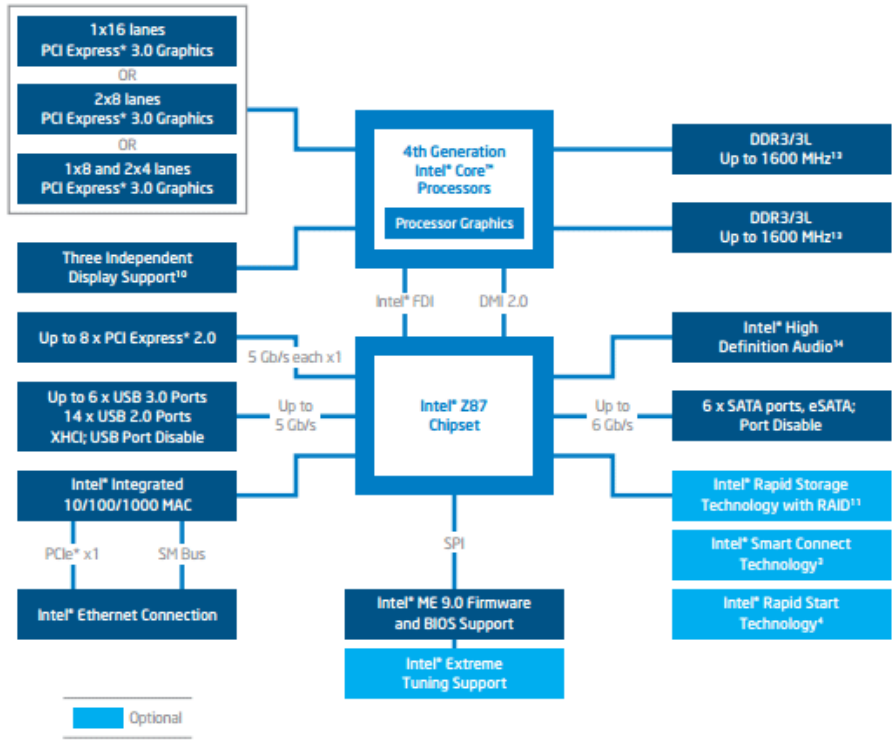


Mémoire physique

La mémoire physique est un espace mémoire qui permet au processeur de communiquer avec tous les périphériques présents sur la machine, que ce soit un composant mémoire ou non (carte réseau, périphérique USB, carte SD, ...). Dans ce dernier cas, le processeur communiquera avec eux en lisant et écrivant à des adresses dédiées.

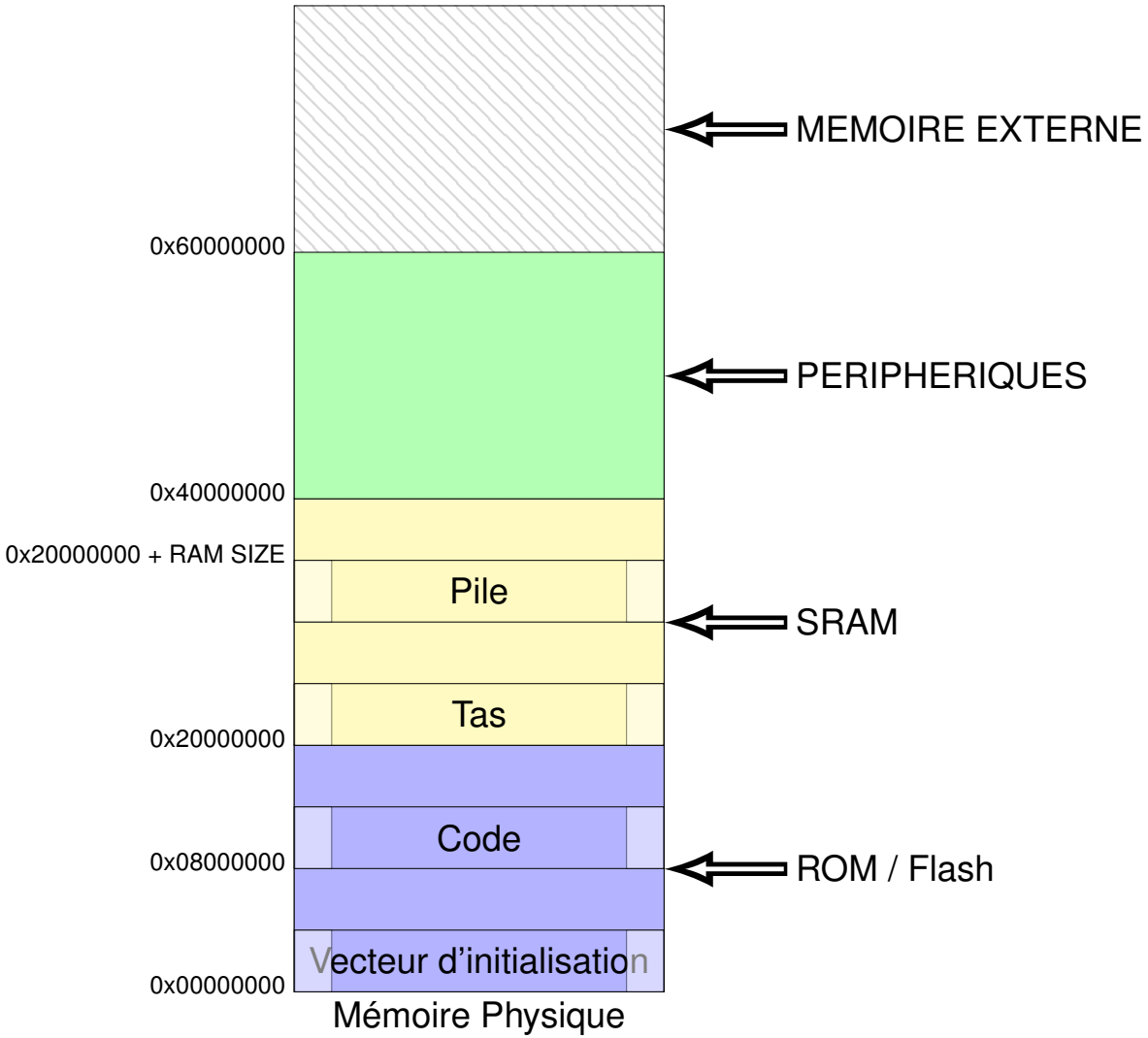
Espace d'adressage physique





Exemple pour les processeurs intel.

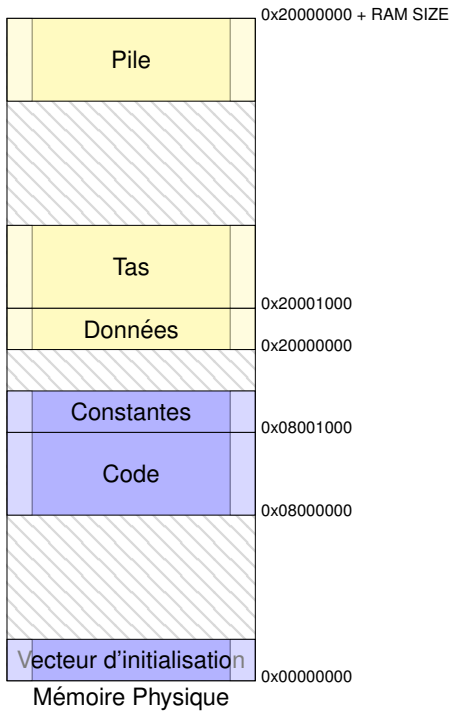
Exemple de l'ARM Cortex-M



- 0x0 est l'adresse du début du code pour démarrer la machine.
- 0x08000000 est l'adresse du début du code principal.
- 0x20000000 est l'adresse du début de l'espace de stockage des données.
- 0x40000000 est l'adresse du début de l'espace des périphériques.

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

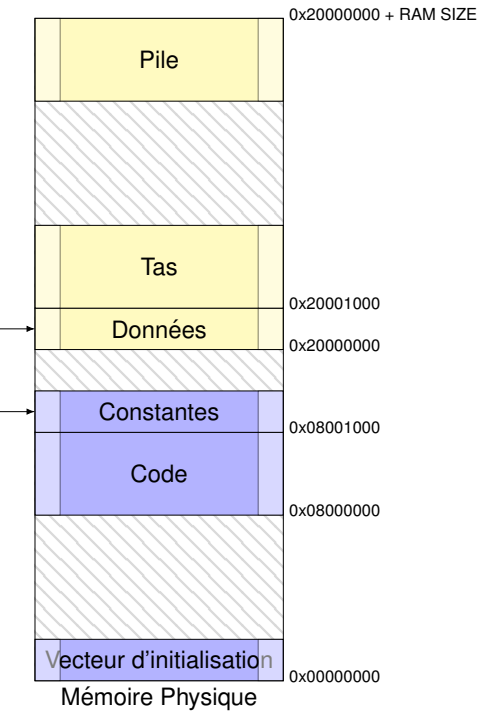


Stockage des données en mémoire

Les constantes et variables sont des données qui sont manipulées par le processeur et qui en fonction de leur nature, vont être stockées sur des segments différents de la mémoire.

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

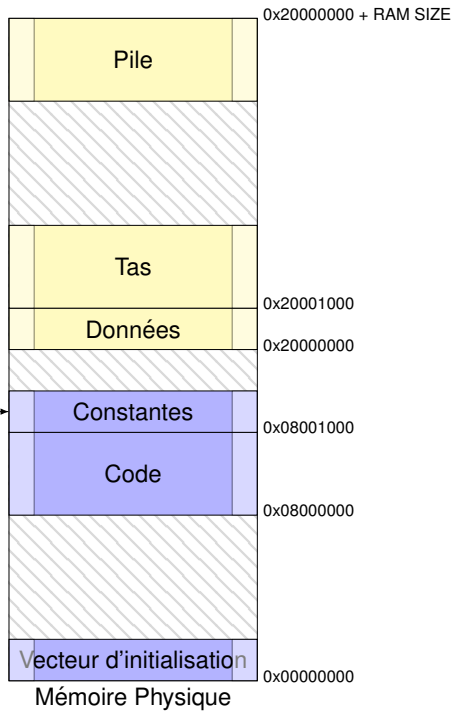


Constantes et variables globales

Les constantes et variables globales sont des données déclarées en dehors des fonctions (souvent en en-tête des fichiers). Elles sont accessibles par toutes les fonctions du programme. Elles existent du début à la fin de l'exécution (se sont des variables statiques).

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

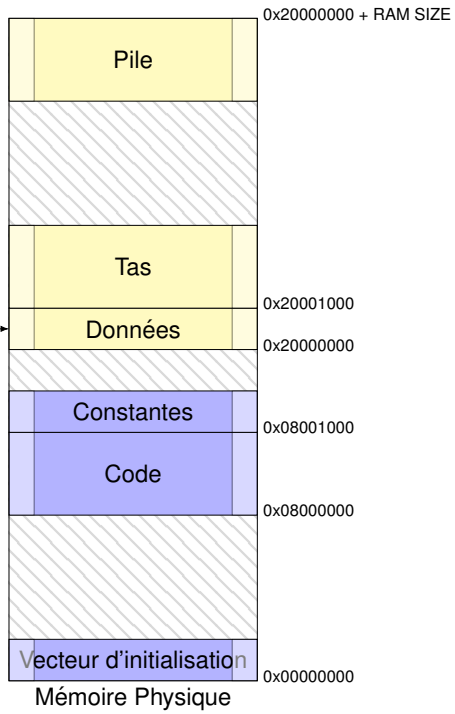


Constantes globales

Les constantes globales sont stockées juste après le code du programme (également appelé le segment text).

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

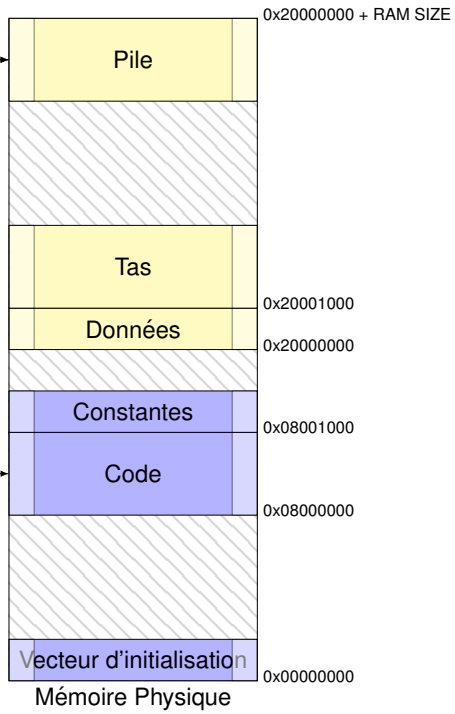


Constantes globales

Les variables globales sont stockées dans le segment des données (également appelé le segment data).

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

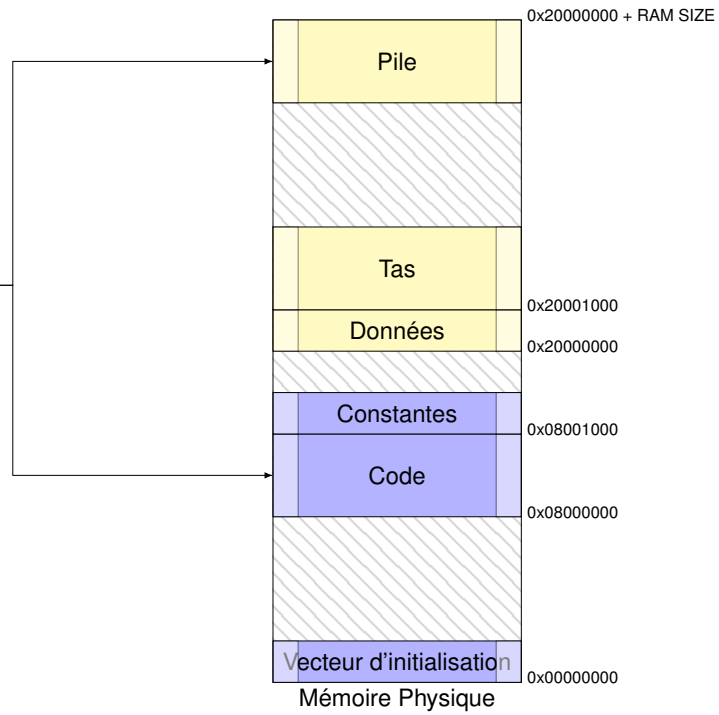


Constantes et variables locales

Les constantes et variables locales sont des données déclarées dans les fonctions. Elles ne sont accessibles que par les fonctions où elles ont été déclarées. Elles n'existent que durant l'exécution de la fonction (se sont des variables dynamiques).

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

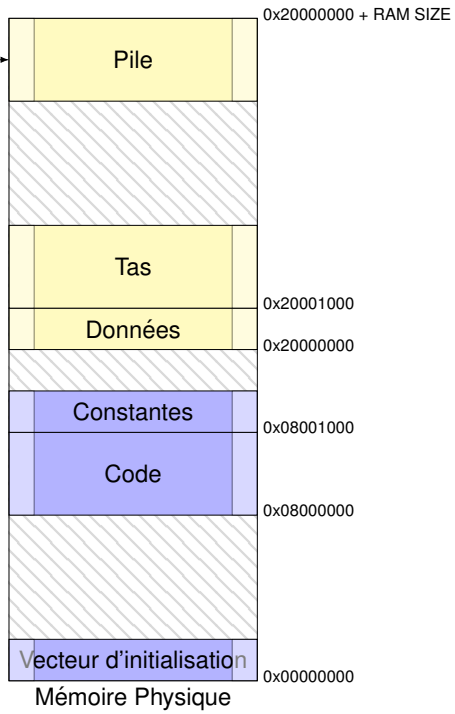


Constantes locales

Les constantes locales sont stockées dans le code un niveau d'une instruction particulière qui vient copier cette constante dans un registre. Notons que comme la taille des instructions est limitée, les plus grosses constantes sont stockées sur plusieurs instructions avec des opérations arithmétiques pour la reconstruire. Elles sont également stockées dans la pile au début du code de la fonction.

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

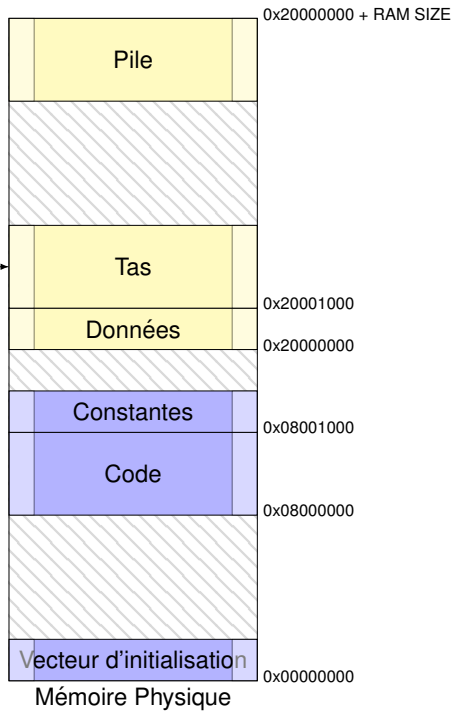


Variables locales

Les variables locales sont stockées dans la pile au début du code de la fonction.

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

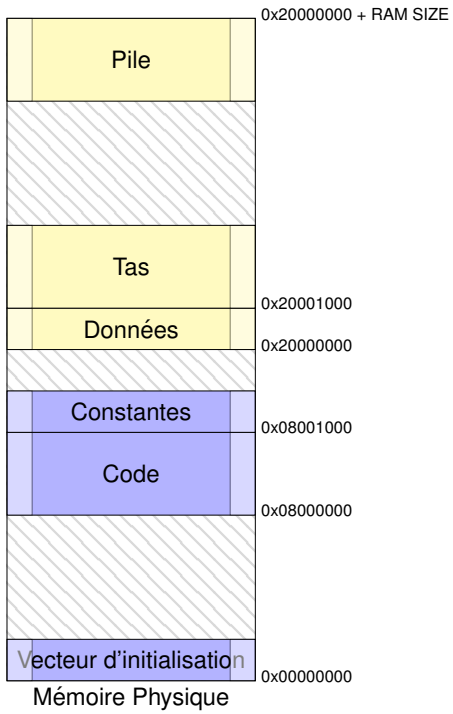


Variables locales allouées dynamiquement

Il existe un dernier type de variables qui sont dites allouées dynamiquement. Elles sont crées (malloc, new) et détruites (free, delete) par le programmeur via des fonctions spécifiques. Elles sont stockées dans le tas. Le tas est de taille variable, et grandi vers le haut en général.

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```

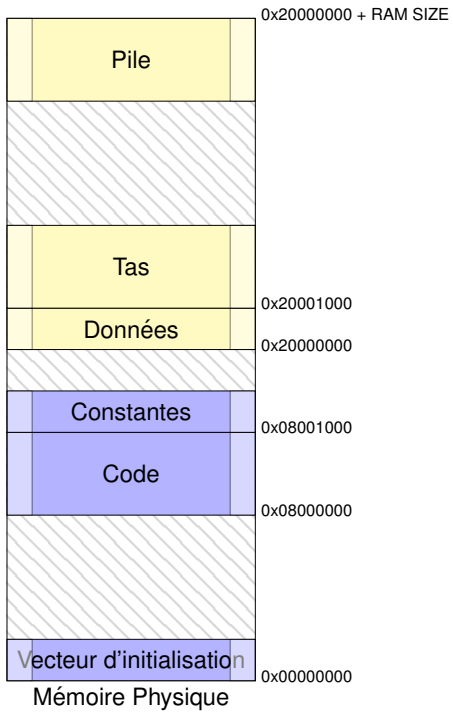


Piscine de données (literal pool)

Il existe une dernière zone mémoire qui s'appelle la piscine de données. Elle est située juste après le code d'une fonction et permet de stocker des constantes qui font référence à des adresses mémoires lointaines (par exemple l'adresse du segment de données, ou l'adresse d'un périphérique).

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```



Question

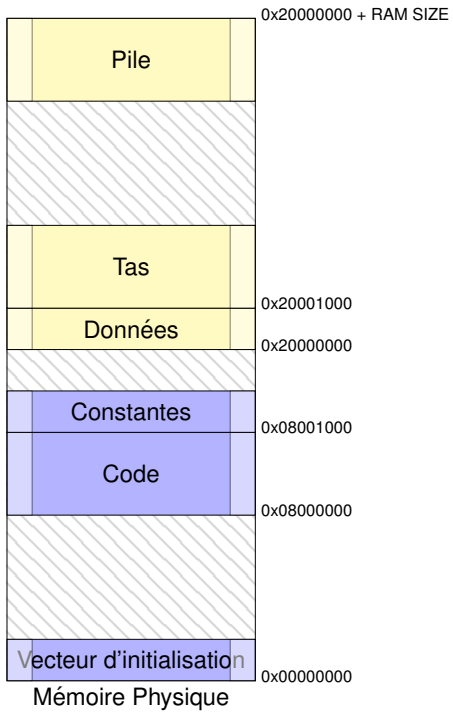
A votre avis, où sont stockés les arguments d'une fonction ?

A. Segment du code (text).
B. Segment de la pile.
C. Segment des données (data).
D. Segment du tas.
E. Les registres du processeur.

L'appel de fonction

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```



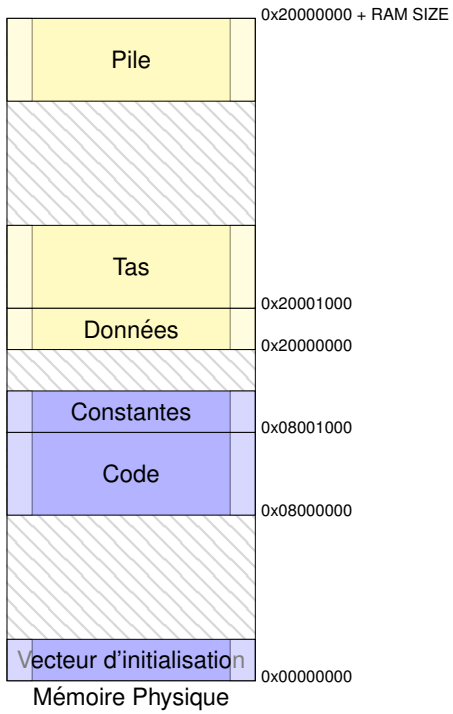
Question

A votre avis, où sont stockés les arguments d'une fonction ?

A. Segment du code (text).
B. Segment de la pile.
C. Segment des données (data).
D. Segment du tas.
E. Les registres du processeur.

```
const int64_t constante_globale = 1234;
int64_t variable_globale = 5678;

int main() {
    const int64_t constante_locale = 4321;
    int64 variable_locale = 8765;
    int64 donnees_dynamiques = malloc(16);
    ma_fonction(constante_locale,
            variable_locale,
            constante_globale,
            variable_globale);
    return 0;
}
```



Remarques

Le stockage dans les registres dépendent de ce que l'on appelle la convention d'appel de fonction :

- Pour les architectures 32 bits d'Intel (x86), tout se fait par la pile.
- Pour les architectures 64 bits d'Intel (x86-64), tout se fait par registres.
- Pour les architectures 32 bits d'Arm, c'est d'abord les registres, puis la pile.

Comment le processeur gère-t-il les périphériques ?

Comme dit précédemment, l'accès aux périphériques fonctionne en lisant et écrivant à des adresses spécifiques.

Certains périphériques ne demandent pas d'attention particulière de la part du processeur et se contentent d'exécuter des taches sur demande (mémoire RAM, souris/claviers, hauts-parleurs, ...).

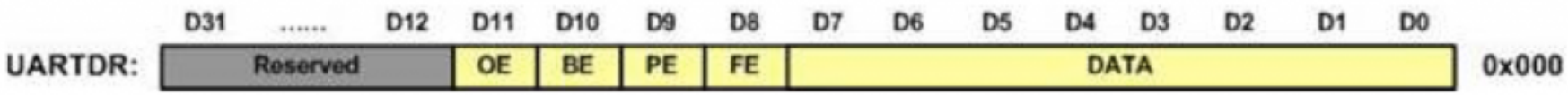
D'autres en revanche fonctionnent par évènements qui doivent être traités par le processeur. On peut citer :

- Le téléchargement de gros fichiers par la carte réseau ;
- La copie de fichiers depuis/vers la mémoire de masse ;
- Le branchement/débranchement d'un périphérique USB ;
- ...

Le polling, également appelé la méthode des drapeaux, consiste à venir lire l'état d'un périphérique par le Processeur de manière régulière.

Le processeur reçoit une suite de bit dont certains bits donnent de l'information sur l'état interne du périphérique. On appel ce type de bit un drapeau (ou flag en anglais).

Exemple du data register de l'uart du cortex-m



La lecture de ce registre (qui est à une adresse physique spécifique), le processeur peut récupérer 1 octet qui a été transmis par la liaison serie (uart) d'un Arm Cortex-M.

Méthode 1 : Le polling

Le polling, également appelé la méthode des drapeaux, consiste à venir lire l'état d'un périphérique par le Processeur de manière régulière.

Le processeur reçoit une suite de bit dont certains bits donnent de l'information sur l'état interne du périphérique. On appel ce type de bit un drapeau (ou flag en anglais).

Remarque

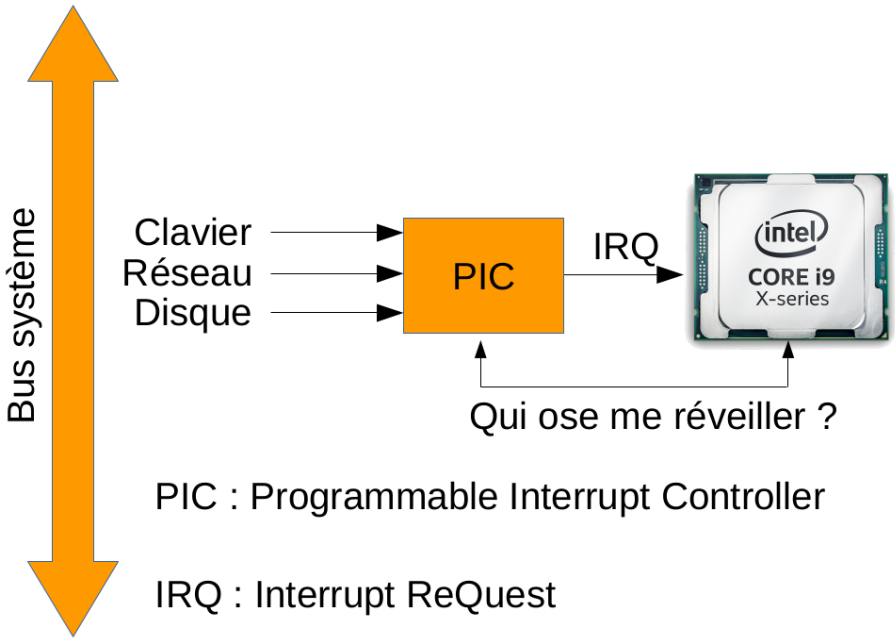
Cette stratégie peut entraîner une surcharge du système, en forçant le processeur à vérifier l'état de chaque périphérique, même si aucune action était réellement nécessaire.

Méthode 2 : Les interruptions

Une interruption est un mécanisme matériel qui permet à un périphérique d'indiquer au processeur la nécessité de traiter un évènement.

Les interruptions sont hiérarchisées afin de donner des priorités à certaines.

Elles peuvent être masquées, c'est à dire temporairement désactivées par le processeur pour ne pas être interrompu.



- Un PIC (Processor Interrupt Controller) est un composant matériel qui permet de gérer les différentes interruptions.
- Chaque interruption est numérotée et associée à une adresse mémoire qui contient le code à exécuter en cas d'interruption.