Architecture avancée : Fiche

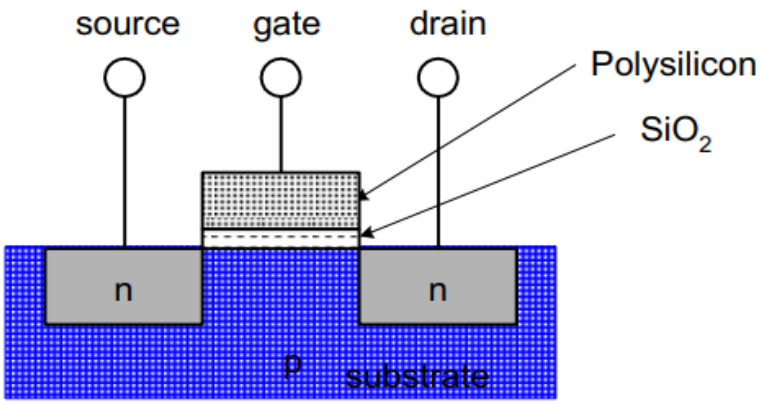
0. Rappels

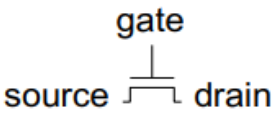
- $2^{10} = 1$ kilo
- $2^{20} = 1$ mega
- $2^{30} = 1$ giga
- *nibble* : demi-octet
- bit de poids faible : bit le plus à droite
- bit de poids fort : bit le plus à gauche

0.1. Transistor

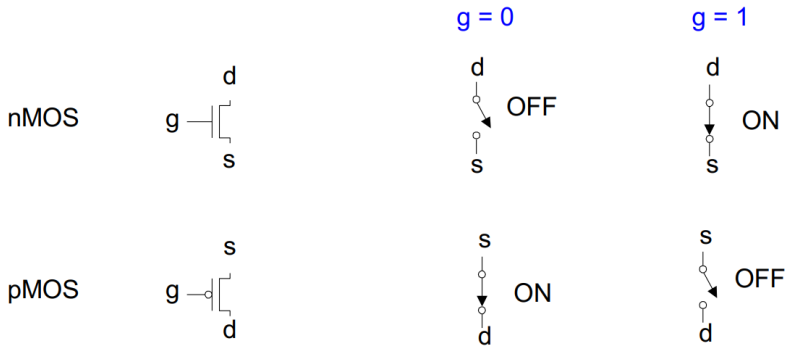
0.1.1. Principe :

- Un transistor est composé de la source, d'un *gate* (grille) et du *drain*. Nous venons sur l'entrée *gate* pour soit connecter la source et le *gain*, soit les déconnecter.

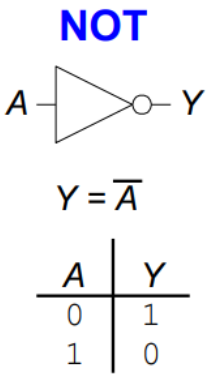


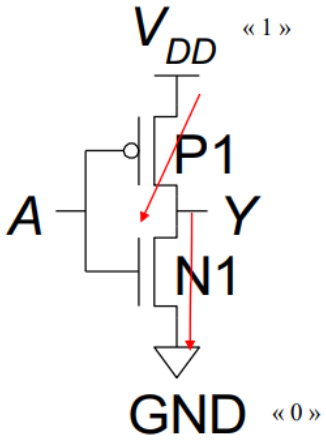


nMOS



0.1.2. Porte NON





A	P1	N1	*Y*
0	ON	OFF	1
1	OFF	ON	0

---

1. Mémoire

- Hierarchie de la mémoire :

    1. Registres (le plus rapide, le moins volumineux, le plus cher à l'octet)

2. Cache
3. Mémoire centrale
4. Disque dur (le moins rapide, le plus volumineux, le moins cher à l'octet)

1.2. Registres *(en. Registers)*

- Le processeur ARM a **16 registres**
- Chaque registre est composé de **32 bits** (ARM est une "architecture 32-bits" car opère sur des données 32 bits)

Certains registres ont des spécificités particulières (ex : `R0` passe des arguments à une fonction, de `R4` à `R15` c'est là où on sauvegarde les données quand on appelle une fonction...)

> En code assembleur ARM, pour initialiser des variables on utilise `MOV` :

- **C :**

```c
//int: 32-bit signed word

int a = 23;
int b = 0x45;
```

- **ARM assembly code :**

```
; R0 = a, R1 = b

      MOV   R0, #23
      MOV   R1, #0x45
```

1.3. Mémoire centrale *(en. Memory)*

- Stocke plus de données que les registres mais moins rapides
- On y stocke les données qu'on utilise souvent
- Chaque donnée a une adresse unique, on utilise les octets. Chaque mot représente 4 octets.
- On ne peut pas utiliser directement les données stockées dans la mémoire, il faut passer par les registres.

> En code assembleur ARM, pour récupérer des données on utilise `LDR` (load):

- **ARM assembly code :**

```
      LDR   R0, [R1, #12]
```

- On calcule une adresse $=$ `R1` $+12$
  - `R1` est la *base address*

    - 12 est l'*offset*
- `R0` contient la donnée stockée à l'adresse mémoire $(\texttt{R1} + 12)$

> En code assembleur ARM, pour écrire des données on utilise `STR` (store):

- Imaginons que nous voulons écrire la valeur stockée dans `R7` dans le mot mémoire 21.
    - L'adresse mémoire $= 4 \times 21 = 84 = 0x54$
- **ARM assembly code :**

```
MOV   R5, #0
STR   R7, [R5, #0x54]
```

---

2. ARM

2.1. Instructions logiques *(en. Logical instructions)*

- `AND` : Utile pour les masques (`0xF234012F AND 0x000000FF = 0x0000002F`)
- `ORR`
- `EOR` : *XOR*
- `BIC` : *Bit Clear*, Utile pour les masques (`0xF234012F BIC 0xFFFFFF00 = 0x0000002F`)
- `MVN` : *MoVe and NOT*

2.1.1. Exemple

Instruction	Registre	Valeur
/	R1	0100 0110 1010 0001 1111 0001 1011 0111
/	R2	1111 1111 1111 1111 0000 0000 0000 0000
AND R3, R1, R2	R3	0100 0110 1010 0001 0000 0000 0000 0000
ORR R4, R1, R2	R4	1111 1111 1111 1111 1111 0001 1011 0111
EOR R5, R1, R2	R5	1011 1001 0101 1110 1111 0001 1011 0111
BIC R6, R1, R2	R6	0000 0000 0000 0000 1111 0001 1011 0111
MVN R7, R2	R7	0000 0000 0000 0000 1111 1111 1111 1111

2.2. Instructions de décalage *(en. Shift instructions)*

- `LSL` : *Logical Shift Left*
- `LSR` : *Logical Shift Right*
- `ASR` : *Arithmetic Shift Right*

- `ROR` : *ROtate Right*

2.2.1. Exemple

Instruction	Registre	Valeur
/	R1	1111 1111 0001 1100 0001 0000 1110 0111
`LSL R2, R1, #7`	R2	1000 1110 0000 1000 0111 0011 1***000 0000***
`LSR R3, R1, #17`	R3	***0000 0000 0000 0000 0***111 1111 1000 1110
`ASR R4, R1, #3`	R4	***111***1 1111 1110 0011 1000 0010 0001 1100
`ROR R5, R1, #21`	R5	***1110 0000 1000 0111 0011 1***111 1111 1000

2.3. Multiplication

- Multiplications 32 bits $\times$ 32 bits.
- `MUL`: *MULtiply*
  - Résultat sur 32 bits (`MUL R1, R2, R3` $\Rightarrow$ `R1 = R2` $\times$ `R3`)
- `UMULL`: *Unsigned MULtiply Long*
  - Résultat sur 64 bits (`MUL R1, R2, R3, R4` $\Rightarrow$ `{R1, R2} = R3` $\times$ `R4` non-signé)
- `SMULL`: *Signed MULtiply Long*
  - Résultat sur 64 bits (`MUL R1, R2, R3, R4` $\Rightarrow$ `{R1, R2} = R3` $\times$ `R4` signé)

2.4. Conditions

2.4.1. Description

Flag	Nom	Description
N	**N**egative	Le résultat de l'instruction est négatif
Z	**Z**ero	Le résultat de l'instruction est nul
C	**C**arry	L'instruction nécessite une retenue
V	o**V**erflow	L'instruction cause un overflow (nombre trop large pour le PC)

2.4.2. Initialisation

- `CMP R5, R6` : Exécute `R5 - R6` sans enregistrer le résultat mais en metant à jour les flags.
- `ADDS R1, R2, R3` : Exécute `R2 + R3`, enregistre le résultat dans `R1` et met à jour les flags.

2.4.3. Utilisation

- Nous pouvons imposer des conditions pour l'exécution d'une instruction en accolant à celle-ci un *condition mnemonic* :

```
      CMP   R1, R2
      SUBNE R3, R5, R8      ; Ne s'exécute que si R1 != R2
```

- Liste complète des *condition mnemonics* en annexe.

2.5. Structures de haut-niveau

- L'instruction `B` permet d'effectuer un saut à l'adresse spécifié.

2.5.1. `if`

```
// En C
if (i == j)
  f = g + h;

else
  f = f - i;



; En ARM
; R0 = f, R1 = g, R2 = h, R3 = i, R4 = j

      CMP   R3, R4          ; défini les flags
      BNE   sinon           ; si i != j, on va à sinon
      ADD   R0, R1, R2      ; f = g + h
      B     suite
sinon
      SUB   R0, R0, R3      ; f = f - I
suite
      .....
```

2.5.2. `while`

```
// En C
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

```
; En ARM
; R0 = pow, R1 = x

      MOV   R0, #1          ; pow = 1
      MOV   R1, #0          ; x = 0

WHILE
      CMP   R0, #128        ; pow - 128
      BEQ   DONE            ; if (pow == 128) quitte le while

      LSL   R0, R0, #1      ; pow = pow * 2
      ADD   R1, R1, #1      ; x = x + 1
      B     WHILE           ; saut vers WHILE

suite
      .....
```

2.5.3. for

```
// En C
for (i = 1; i != 10; i = i + 1)
  sum = sum + i;


; En ARM
; R0 = i, R1 = sum

      MOV   R1, #0          ; sum = 0
      MOV   R0, #1          ; i = 1

FOR
      CMP   R0, #10         ; i - 10
      BEQ   DONE            ; if (i == 10)
      ADD   R1, R1, R0      ; sum = sum + i
      ADD   R0, R0, #1      ; i = i + 1
      B     FOR             ; saut vers FOR

DONE
      .....
```

2.5.4. Tableaux

```c
// En C
int tab[5];

tab[0] = tab[0] * 8;
tab[1] = tab[1] * 8;
```

```arm
; En ARM
; R0 = Adresse du premier élément du tableau (base address)

      MOV   R0, #0x60000000 ; R0 = 0x60000000

      LDR   R1, [R0]        ; R1 = tab[0]
      LSL   R1, R1, #3      ; R1 = R1 << 3 (= MUL R1, R1, #8 donc R1 * 8)
      STR   R1, [R0]        ; tab[0] = R1

      LDR   R1, [R0, #4]    ; R1 = tab[1]
      LSL   R1, R1, #3      ; R1 = R1 << 3 (= MUL R1, R1, #8 donc R1*8)
      STR   R1, [R0, #4]    ; tabs[1] = R1
```

2.5.5. Fonctions

- Nous allons utiliser le registre PC qui est incrémenté automatiquement par le microprocesseur entre chaque instruction. Ce registre permet au microprocesseur de savoir quelle instruction exécuter. C'est la position actuelle dans le programme $+8$ car le microprocesseur prévoit ce qui va s'exécuter.
    - On va *move* le *link register* dans le PC, c'est-à-dire qu'on va dire au microprocesseur de revenir dans le main après l'exécution de la fonction.
- Il faut faire attention à la ré-écriture, nous ne voulons pas impacter les registres dans le *Callee*. Pour ça, on utilise le ***Stack*** (pile) qui est une mémoire dynamique utilisée pour stocker des variables temporaires. Il fonctionne sur le principe du LIFO (Last In First Out).
    - Pour accéder au haut de la pile, on utilise SP, le *Stack Pointer*

```c
// En C
int diffofsums(int f, int g, int h, int i) {    // Callee
  int result;
  result = (f + g) - (h + i);
  return result;
}

int main() {
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);                   // Caller
```

```
  ...
}
```

2.5.5.1. Méthode 1

```
; R4 = y

MAIN
      ...
      MOV   R0, #2          ; argument 0 = 2
      MOV   R1, #3          ; argument 1 = 3
      MOV   R2, #4          ; argument 2 = 4
      MOV   R3, #5          ; argument 3 = 5
      BL    DIFFOFSUMS      ; appelle la fonction
      MOV   R4, R0          ; y = valeur retournée
      ...

; R4 = result

DIFFOFSUMS

; STR R4, [SP, #-4]! : Mode d'adressage : pre-indexé.
; Charge la valeur trouvé dans R2 dans l'adresse mémoire SP - 4.
; SP est modifié: SP = SP - 4 grâce au '!'

      STR   R4, [SP, #-4]!  ; empile R4 sur le stack
      STR   R5, [SP, #-4]!  ; empile R5 sur le stack
      STR   R6, [SP, #-4]!  ; empile R6 sur le stack

      ADD   R4, R0, R1      ; R8 = f + g
      ADD   R5, R2, R3      ; R9 = h + i
      SUB   R6, R8, R9      ; result = (f + g) - (h + i)
      MOV   R0, R6          ; met la valeur à retourner dans R0

; LDR   R6, [SP], #4  : Mode d'adressage : post-indexé.
; Charge la valeur trouvé dans SP dans l'adresse mémoire R6.
; SP est modifié: SP = SP + 4

      LDR   R6, [SP], #4    ; recharge R6
      LDR   R5, [SP], #4    ; recharge R5
      LDR   R4, [SP], #4    ; recharge R4

      MOV   PC, LR          ; retourne dans le main
```

2.5.5.1. Méthode 2

```
DIFFOFSUMS
        PUSH  {R4 - R6}       ; empile les registres 4 à 6 sur le stack

        ADD   R4, R0, R1      ; R8 = f + g
        ADD   R5, R2, R3      ; R9 = h + i
        SUB   R6, R8, R9      ; result = (f + g) - (h + i)
        MOV   R0, R6          ; met la valeur à retourner dans R0

        POP   {R4 - R6}       ; recharge les registres 4 à 6

        MOV   PC, LR          ; retourne dans le main
```

---

3. Encodage des instructions (sur 32 bits)

3.1. Format des instructions de traitement de données (en. Data-processing format)

- **Champs de contrôle** (12 bits)
  - ***cond*** : condition d'exécution (4 bits) (voir annexes)
  - ***op*** : *opcode* (2 bits $= 00_2$ en *data-processing*)
  - ***funct*** : fonction à exécuter (6 bits)
    - ***I*** : (1 bit)
      - $= 0$ si *Src2* est un registre
      - $= 1$ si *Src2* est un immédiat
    - ***cmd*** : instruction à exécuter (4 bits) (voir annexes)
    - ***S*** : (1 bit)
      - $= 0$ si l'instruction ne définit pas les *flags*
      - $= 1$ si l'instruction définit les *flags*
- **Opérandes** (20 bits)
  - ***Rn*** : premier registre source (4 bits)
  - ***Rd*** : registre de destination (4 bits)
  - ***Src2*** : deuxième registre source ou immédiat (12 bits)
    - Si c'est un immédiat :
      - ***rot*** : rotation vers la droite d'*imm8* de $2\times$ rot (4 bits)
      - ***imm8*** : valeur de l'immédiat (8 bits)
    - Si c'est un registre :
      - ***shamt5*** : le nombre de fois que *Rm* est décalé (5 bits)

        - ***sh*** : type de shift (>>, <<, >>>, ROR) $(2$ bits $)$
        - ***0*** : bit nul $(1$ bit $)$
        - ***Rm*** : deuxième registre source $(4$ bits $)$
    - (Autre cas non-traité : *Register-shifted Regsiter*)

3.2. Format des instructions de mémoire (en. Memory instruction format)

- **Champs de contrôle** $(12$ bits $)$
    - ***cond*** : condition d'exécution $(4$ bits $)$ (voir annexes)
    - ***op*** : *opcode* $(2$ bits $= 01_2$ en *memory instruction* $)$
    - ***funct*** : fonction à exécuter $(6$ bits $)$
        - $\bar{I}$: $(1$ bit $)$
            - $= 0$ si *Src2* est un immédiat
            - $= 1$ si *Src2* est un registre
        - ***P*** : $(1$ bit $)$
            - $= 0$ si le mode d'indexage est en *Postindex* ou en *Not Supported*
            - $= 1$ si le mode d'indexage est en *Preindex* ou en **Offset**
        - ***U*** : $(1$ bit $)$
            - $= 0$ si l'offset doit être soustrait
            - $= 1$ si l'offset doit être ajouté
        - ***B*** : $(1$ bit $)$
            - $= 0$ si **STR** ou **LDR**
            - $= 1$ si *STRB* ou *LDRB*
        - ***W*** : $(1$ bit $)$
            - $= 0$ si le mode d'indexage est en *Postindex* ou en **Offset**
            - $= 1$ si le mode d'indexage est en *Preindex* ou en *Not Supported*
        - ***L*** : $(1$ bit $)$
            - $= 0$ si **STR** ou *STRB*
            - $= 1$ si **LDR** ou *LDRB*
- **Opérandes** $(20$ bits $)$
    - ***Rn*** : $(4$ bits $)$
        - Source en Load
        - Destination en Store
    - ***Rd*** : $(4$ bits $)$
        - Destination en Load
        - Source en Store
    - ***Src2*** : Offset $(12$ bits $)$
        - Si c'est un immédiat :
            - ***imm12*** : valeur de l'immédiat $(12$ bits $)$

        - Si c'est un registre :
            - ***shamt5*** : le nombre de fois que *Rm* est décalé $(5$ bits $)$
            - ***sh*** : type de shift (>>, <<, >>>, ROR) $(2$ bits $)$
            - ***0*** : bit nul $(1$ bit $)$
            - ***Rm*** : deuxième registre source $(4$ bits $)$
        - (Autre cas non-traité : *Register-shifted Regsiter*)

3.3. Format des instructions de branchement (en. Branch instruction format)

- **Champs de contrôle** $(8$ bits $)$
    - ***cond*** : condition d'exécution $(4$ bits $)$ (voir annexes)
    - ***op*** : *opcode* $(2$ bits $= 10_2$ en *branch instruction* $)$
    - ***funct*** : fonction à exécuter $(2$ bits $)$
- ***BTA*** (*Branch Target Address* : l'adresse de la cible) $(24$ bits $)$
    - ***imm24*** : Nombre de mots séparant `PC + 8` et `BTA` $(24$ bits $)$

3.4. Exemple

```
BOUCLE  CMP     R0, #0
        BNE     SINON
        ADD     R1, R1, #1
        B       FIN

SINON   SUB     R1, R1, #1
        LDR     R1, [R2, #4]
        STR     R3, [R7, R8]

FIN     BL      BOUCLE
```

`BOUCLE CMP R0, #0`

Cond	OP	I	Cmd	S	Rn	Rd	Src2
1110	00	1	1010	0	0000	0000	000000000000

`BNE SINON`

Cond	OP	1L	imm24

Cond	OP	1L	imm24
0001	10	10	00000000000000000000001

`ADD R1, R1, #1`

Cond	OP	I	Cmd	S	Rn	Rd	Src2
1110	00	1	0100	0	0001	0001	000000000001

`B FIN`

Cond	OP	1L	imm24
1110	10	10	00000000000000000000010

`SINON SUB R1, R1, #1`

Cond	OP	I	Cmd	S	Rn	Rd	Src2
1110	00	1	0010	0	0001	0001	000000000001

`LDR R1, [R2, #4]`

Cond	OP	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2
1110	01	0	1	1	0	0	1	0010	0001	000000000100

`STR R3, [R7, R8]`

Cond	OP	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2
1110	01	1	1	1	0	0	0	0111	0011	000000001000

`FIN BL BOUCLE`

Cond	OP	1L	imm24
1110	10	10	11111111111111111110111

($11111111111111111110111 = -9$ en complément à 2)

---

4. Micro-architecture

4.1. Implémentations

- *Single-cycle* (Cycle unique) : Chaque instruction s'exécute d'une traite.

- *Multicycle* : Chaque instruction est divisée en plusieurs étapes.
- *Pipelined* (pipeliné) : Chaque instruction est divisée en plusieurs étapes et plusieurs instructions s'exécutent en même temps.

---

5. Mémoire Cache

5.1. Préambule

- Viens du mot "cacher"
- Le cache contient des copies des blocs de données (*cache line*) qui sont stockées dans la mémoire centrale. Toutes les lectures de la mémoire centrale redirigent vers le cache pour voir si la donnée y est contenue.
    - La donnée y est : ***cache hit***, la donnée requise est lue dans le cache.
    - La donnée n'y est pas : ***cache miss***, la donnée requise est envoyée de la mémoire centrale au cache, puis lue.
- Les caches permettent de réutiliser les données récemment extraites. Les caches reposent sur le principe de 2 localité :
    - Localité temporelle - l'information qui vient d'être utilisée est susceptible d'être réutilisée dans le futur.
    - Localité spatiale - l'information adjacente à celle utilisée est susceptible d'être utilisée dans le futur.
        - Nécessite de larges extractions de données de la mémoire vers le cache : bloc de données)
- Les blocs cache font la même taille que les blocs mémoire. A chaque bloc est associé :
    - *Tag* : dérivé de l'adresse du cache qui est utilisée pour déterminer si un accès au cache a eu lieu ou non
    - *Valide Bit* : ligne de cache pleine (valide) ou libre (non valide)
    - *Modified Bit* : indiquant si la ligne de cache a été modifiée (écriture)

5.2. Mappage de la mémoire centrale au cache (en. Mapping)

- Schéma simple : ***DMC*** (*Direct Mapped Cache*) limite l'endroit où une adresse mémoire donnée peut se trouver dans le cache. Utilise l'adressage modulo.
- Schéma plus complexe ***FA*** (Fully Associative cache) utilise l'adressage associatif qui permet aux données de n'importe quelle adresse en mémoire d'être placées dans n'importe quel bloc du cache.

5.2.1. Cas du DMC (Correspondance directe)

> Très restrictif donc beaucouop de recherche dans la mémoire centrale

- Cas d'un DMC avec un cache de taille $16\ Ko$, un bloc est de taille $16\ o$ donc il y a 1024 bloc
- L'adresse d'une donnée est divisée en trois parties : $(32$ bits $)$
    - ***Tag*** : indique quel bloc mémoire est mappé $(18$ bits $)$
    - ***Index*** : index du bloc dans lequel la donnée est stockée $(10$ bits $)$
    - ***Offset*** : offset dans la *cache line* $(4$ bits $)$

5.2.1.1. Exemple

- Soit le cache pour l'instant vide, nous voulons lire des données :

Index	V	Tag	0xf-c	0xb-8	0x7-4	0x3-0
0						
1						
2						
3						
4						
5						
6						

- `00000000000000000 0000000001 0100`
    - A l'index 1, il n'y a pas de donnée : ***Cold Miss***.
    - Nous l'ajoutons donc, puis nous la lisons suivant l'offset (**b**) :

Index	V	Tag	0xf-c	0xb-8	0x7-4	0x3-0
0						
1	1	0	d	c	**b**	a
2						
3						
4						
5						
6						

- `00000000000000000 0000000001 0001`
    - A l'index 1, il y a une donnée et le tag correspond : ***Hit***.
    - Nous lisons suivant l'offset (**a**)

Index	V	Tag	0xf-c	0xb-8	0x7-4	0x3-0
0						
1	1	0	d	c	b	a
2						
3						
4						
5						
6						

- `00000000000000010 0000000001 0100`
    - A l'index 1, il y a une donnée mais le tag ne correspond pas : ***Hot Miss***.
    - Nous l'ajoutons donc, puis nous la lisons suivant l'offset (**f**) :

Index	V	Tag	0xf-c	0xb-8	0x7-4	0x3-0
0						
1	1	2	h	g	f	e
2						
3						
4						
5						
6						

5.2.2. Cas du FA (Correspondance associative)

> Beaucoup d'opérations

- Pas de notion d'index, le tag est comparé à toutes les entrées.
- Cas d'un FA avec un cache de taille $16\ Ko$, un bloc est de taille $16\ o$ donc il y a 1024 bloc
- L'adresse d'une donnée est divisée en deux parties : $(32$ bits $)$
    - ***Tag*** : indique quel bloc mémoire est mappé $(28$ bits $)$
    - ***Offset*** : offset dans la *cache line* $(4$ bits $)$

5.2.2.1. Exemple

- Soit le cache pour l'instant vide, nous voulons lire des données :

Index	V	Tag	0xf-c	0xb-8	0x7-4	0x3-0
0						
1						
2						
3						
4						
5						
6						

- `00000000000000000000000000000000 0100`
    - Pas de donnée correspondant au tag : ***Miss***.
    - Nous l'ajoutons donc, puis nous la lisons suivant l'offset (**b**) :

Index	V	Tag	0xf-c	0xb-8	0x7-4	0x3-0
0	1	0	d	c	**b**	a
1						
2						
3						
4						
5						
6						

- `00000000000000000000000000000000 0001`
    - Un tag correspond : ***Hit***.
    - Nous lisons suivant l'offset (**a**)
- `00000000000000000000000000000010 0100`
    - Pas de donnée correspondant au tag : ***Miss***.
    - Nous l'ajoutons donc, puis nous la lisons suivant l'offset (**f**) :

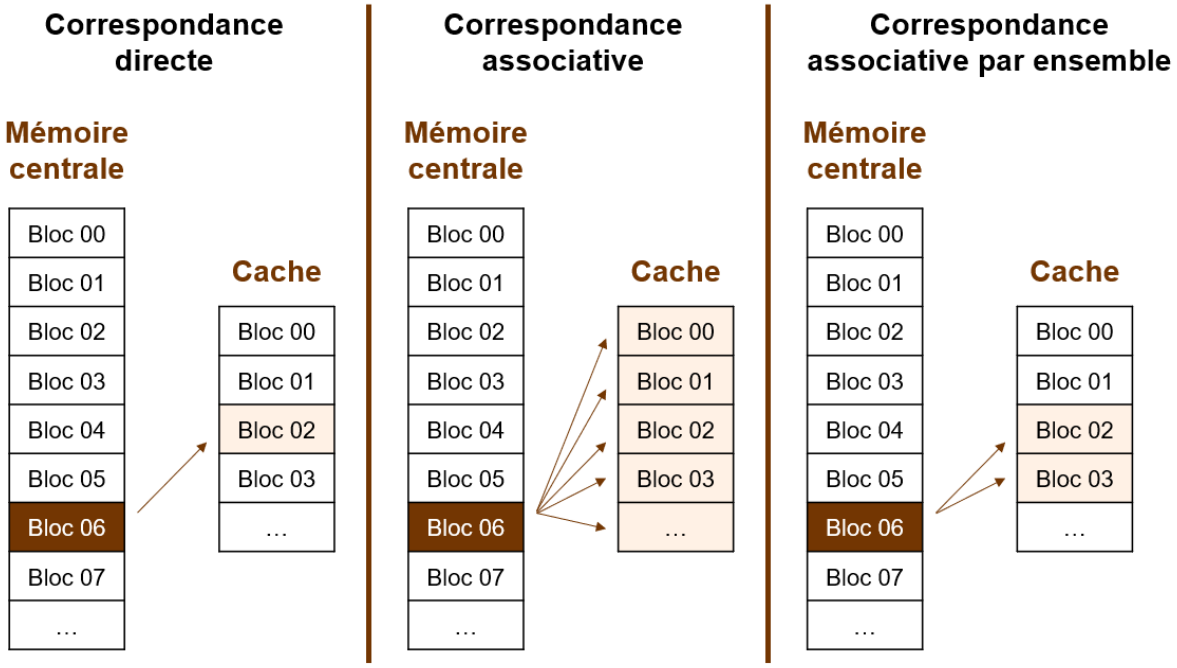
Index	V	Tag	0xf-c	0xb-8	0x7-4	0x3-0
0	1	0	d	c	b	a
1	1	2	h	g	**f**	e
2						
3						
4						
5						
6						

5.2.3. Compromis : *N-way Set Associative Cache* (Correspondance associative par ensemble)

- N entrées à chaque ensemble d'index, donc pour un ensemble i, N tags peuvent correspondre

- Plus permissif que la **DMC**, moins d'opérations que la **FA**

5.2.4. **Résumé** (Schéma)



5.2.5. Normes de remplacement

- En correspondance associative par ensemble ou en correspondance associative, lorsqu'il faut expulser un bloc de cache pour y mettre le bloc à lire, il y a plusieurs possibilités.
    - Ejecter le bloc le moins récemment utilisé : ***LRU*** *(Least Recently Used)*.
        - Dur à implémenter, il faut enregistrer chaque fois qu'un bloc est *hit*.
    - Ejecter au hasard.
        - Bien plus simple, pas besoin d'enregistrer.

5.3. Ecriture

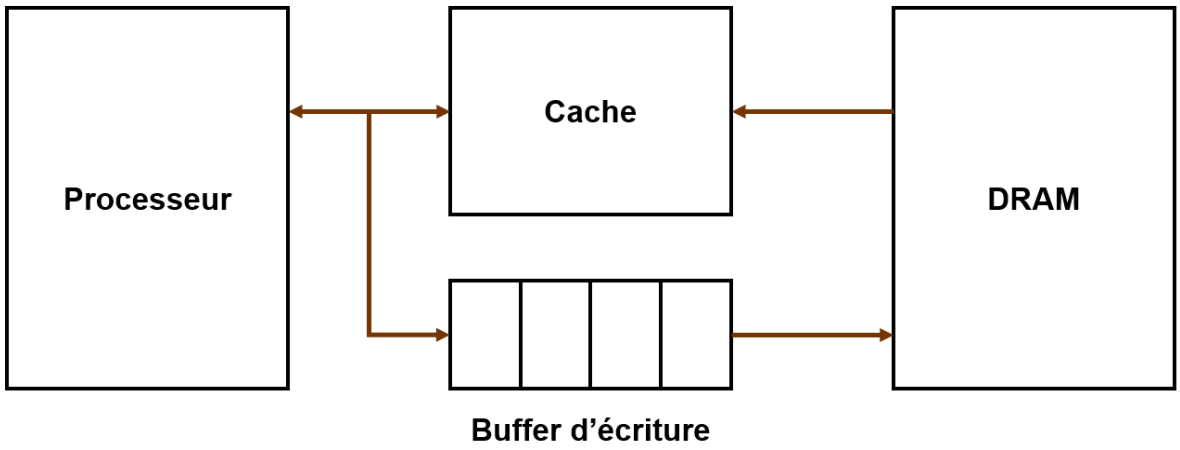
- ***Write Through*** *(WT)*: Les informations sont écrites à la fois dans le bloc du cache et dans le bloc de la mémoire centrale.
- ***Write back*** *(WB)*: les informations sont écrites uniquement dans le bloc du cache. Le bloc de cache modifié n'est écrit dans la mémoire centrale que lorsqu'il est remplacé.

5.3.1. Avantages et inconvénients

- *Write Through*
    - Les *Read Misses* ne peuvent pas entraîner d'écriture
    - Risque de saturation de la mémoire
- *Write back*
    - Pas d'écriture lorsqu'il y a plusieurs écritures à la suite.

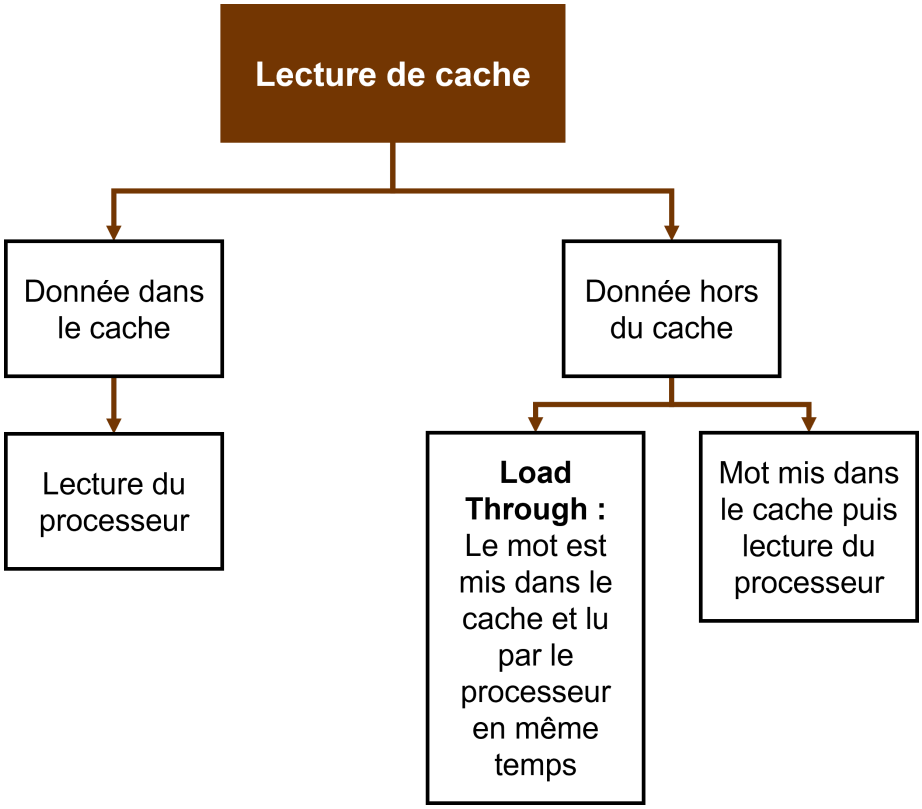
5.3.2. *Write Buffer*

- Les *WT* sont toujours combginé avec des buffers d'écriture (en *FIFO*) pour accélérer le processus. Il y a souvent 4 entrées.

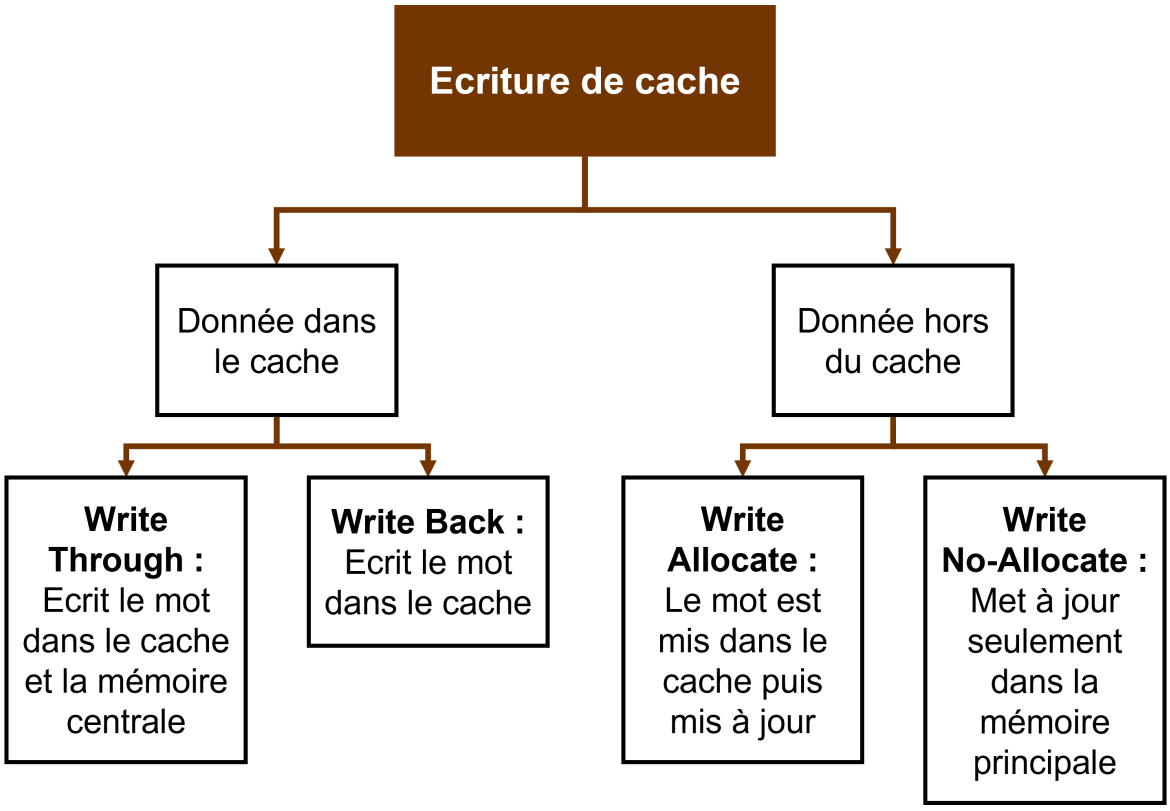


5.4. Politique de lecture et d'écriture (Schéma)

5.4.1. Lecture



5.4.2. Ecriture



Annexes

Hexadécimal - Décimal - Binaire

Hexa	Dec	Bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101

Hexa	Dec	Bin
E	14	1110
F	15	1111

Condition mnemonics

Mnemonic	cond	Description	Flags testés
EQ	0000	*EQual* - égal	Z == 1
NE	0001	*Not Equal* - inégal	Z == 0
CS ou HS	0010	*Carry Set* ou *unsigned Higer or Same* - retenue	C == 1
CC ou LO	0011	*Carry Clear* ou *unsigned LOwer* - pas de retenue	C = 0
MI	0100	*MInus* - négatif	N == 1
PL	0101	*PLus* - positif ou nul	N == 0
VS	0110	*V Set* - overflow	V == 1
VC	0111	*V Clear* - pas d'overflow	V == 0
HI	1000	*HIgher* - non-signé : plus grand	(C == 1) && (Z == 0)
LS	1001	*Lower or Same* - non-signé : plus petit ou égal	(C == 0) || (Z == 1)
GE	1010	*Greater than or Equal* - signé : plus grand ou égal	N == V
LT	1011	*Less Than* - signé - plus petit	N != V
GT	1100	*Greater Than* - signé : plus grand	(Z == 0) && (N == V)
LE	1101	*Less than or Equal* - signé - plus petit ou égal	(Z == 1) || (N != V)
AL (ou rien)	1110	*ALways executed* - pas de condition	/

Champ CMD

Opération	cmd	Description	Détail
AND	0000	*logical AND* - et logique	Rd := Rn AND Src2
EOR	0001	*logical Exclusive OR* - ou exclusif logique	Rd := Rn EOR Src2
SUB	0010	*SUBtract* - soustraction	Rd := Rn - Src2

Opération	cmd	Description	Détail
RSB	0011	*Reverse SuBtract* - soustraction inverse	`Rd := Src2 - Rn`
ADD	0100	*ADDition* - addition	`Rd := Rn + Src2`
ADC	0101	*ADdition with Carry* - addition avec retenue	`Rd := Rn + Src2 + Carry Flag`
SBC	0110	*SuBtract with Carry* - soustraction avec retenue	`Rd := Rn - Src2 + Carry Flag`
RSC	0111	*Reverse Subtract With Carry* - soustraction inverse avec retenue	`Rd := Src2 - Rn + Carry Flag`
TST	1000	*TeST* - test	Met à jour les flags après `Rn AND Src2`
TEQ	1001	*Test EQuivalence* - test d'équivalence	Met à jour les flags après `Rn EOR Src2`
CMP	1010	*CoMPare* - comparaison d'une soustraction	Met à jour les flags après `Rn - Src2`
CMN	1011	*CoMpare Negated* - comparaison d'une somme	Met à jour les flags après `Rn + Src2`
ORR	1100	*logical OR* - ou logique	`Rd := Rn OR Src2`
MOV	1101	*MOVe* - attribution	`Rd := Src2`
BIC	1110	*BIt Clear*	`Rd := Rn AND NOT (Src2)`
MVN	1111	*MoVe Not* - non	`Rd := NOT (Src2)`