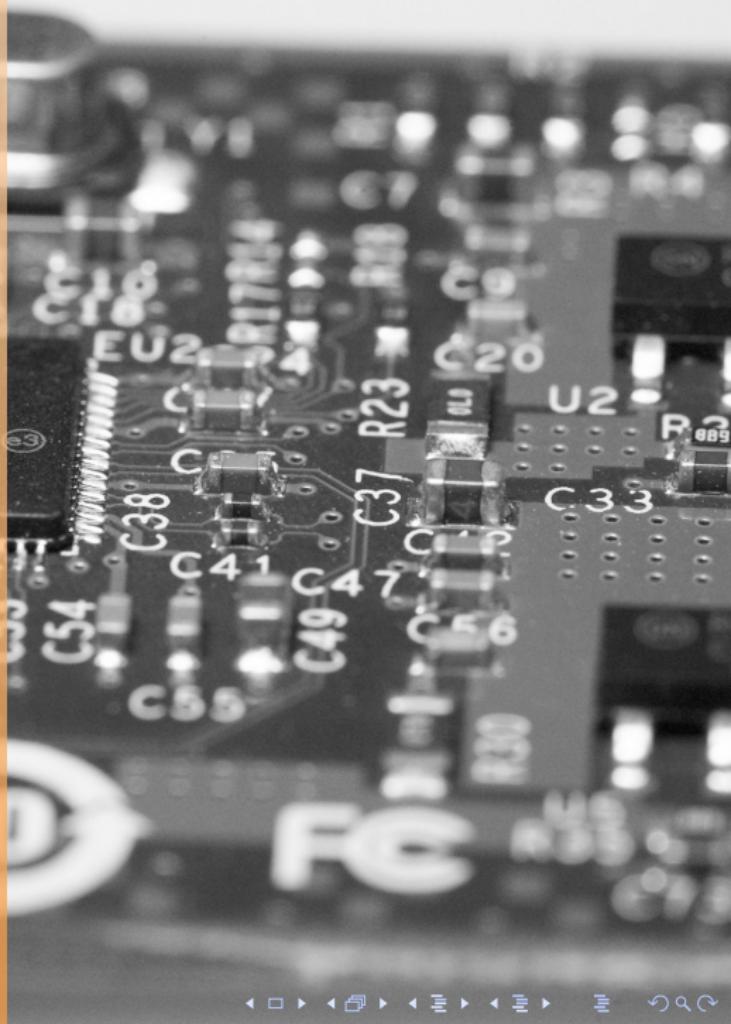


Architecture des processeurs

T. Carle, H. Cassé, C. Rochange, P. Sainrat
2022-2027



Objectifs

- Comprendre comment fonctionne un processeur
- Comprendre la différence entre jeu d'instructions et microarchitecture
- Savoir écrire et comprendre des programmes assembleur ARM
- Concevoir un processeur simple au niveau circuit
- Comprendre les principes de l'organisation micro-architecturale des processeurs modernes
- Comprendre le fonctionnement de la mémoire et des bus

Plan du cours

- Introduction
- Jeu d'instructions et programmation assembleur ARM
 - Structures algorithmiques
 - Mémoire et adressage
 - Sous-programmes
 - Entrées/Sorties
- Organisation micro-architecturale
 - Principaux composants d'un microprocesseur
 - Organisation en pipeline
- Mémoire cache

Introduction

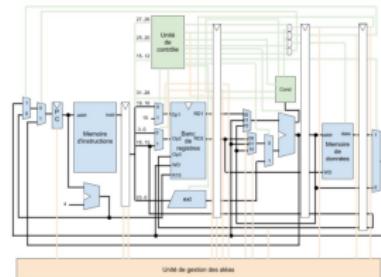
LOGICIEL

Langage de programmation

int x = x + 16 ;

Architecture - ISA

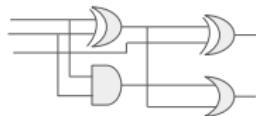
ADD R0, R0, #16 → 1110001010000000000000000000000010000



MATÉRIEL

Microarchitecture

Circuit logique



Transistors

Introduction

Un processeur est un circuit qui exécute des **instructions**. Chaque instruction composant un programme configure le processeur pour effectuer une opération particulière.

Introduction

Un processeur est un circuit qui exécute des **instructions**. Chaque instruction composant un programme configure le processeur pour effectuer une opération particulière.

- Les instructions sont chargées dans une **mémoire** (e.g. la RAM) connectée au processeur.
 - Les instructions sont stockées **en séquence** dans la mémoire.
 - Les instructions ARM que nous considérons sont codées sur **32 bits**.

Introduction

Un processeur est un circuit qui exécute des **instructions**. Chaque instruction composant un programme configure le processeur pour effectuer une opération particulière.

- Les instructions sont chargées dans une **mémoire** (e.g. la RAM) connectée au processeur.
 - Les instructions sont stockées **en séquence** dans la mémoire.
 - Les instructions ARM que nous considérons sont codées sur **32 bits**.
- Le processeur exécute les instructions en séquence.
 - Lorsqu'il n'y a plus d'instruction, le processeur continue d'exécuter le contenu de la mémoire.

Introduction

Un processeur est un circuit qui exécute des **instructions**. Chaque instruction composant un programme configure le processeur pour effectuer une opération particulière.

- Les instructions sont chargées dans une **mémoire** (e.g. la RAM) connectée au processeur.
 - Les instructions sont stockées **en séquence** dans la mémoire.
 - Les instructions ARM que nous considérons sont codées sur **32 bits**.
- Le processeur exécute les instructions en séquence.
 - Lorsqu'il n'y a plus d'instruction, le processeur continue d'exécuter le contenu de la mémoire.
- Les instructions opèrent sur les **registres** du processeur. Les registres sont des mémoires présentes dans le processeur, utilisées pour manipuler et stocker les **opérandes** des instructions.
 - Les processeurs ARM sont équipés de 16 registres généraux (r0 à r15) et de registres à usages particuliers (e.g. CPSR).

Du langage C aux instructions ARM



`i=i+1;`



Compilateur



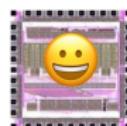
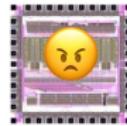
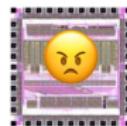
`add r1, r1, #1`



Assembleur



`1110 0010 0000 0100 0100 0000 0000 0001`



Et en hexadécimal, ça fait combien ?

Du langage C aux instructions ARM



i = i + 1;



Compilateur



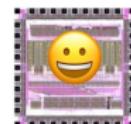
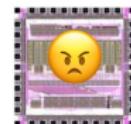
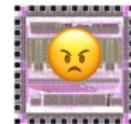
add r1, r1, #1



Assembleur



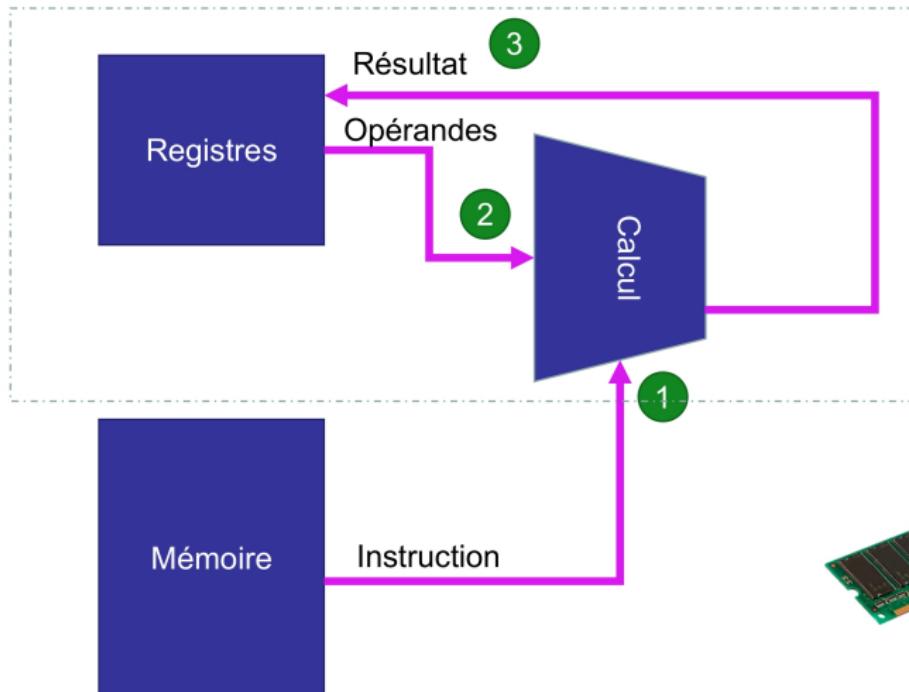
1110 0010 0000 0100 0100 0000 0000 0001



1110 0010 0000 0100 0100 0000 0000 0001 => 0xE2044001

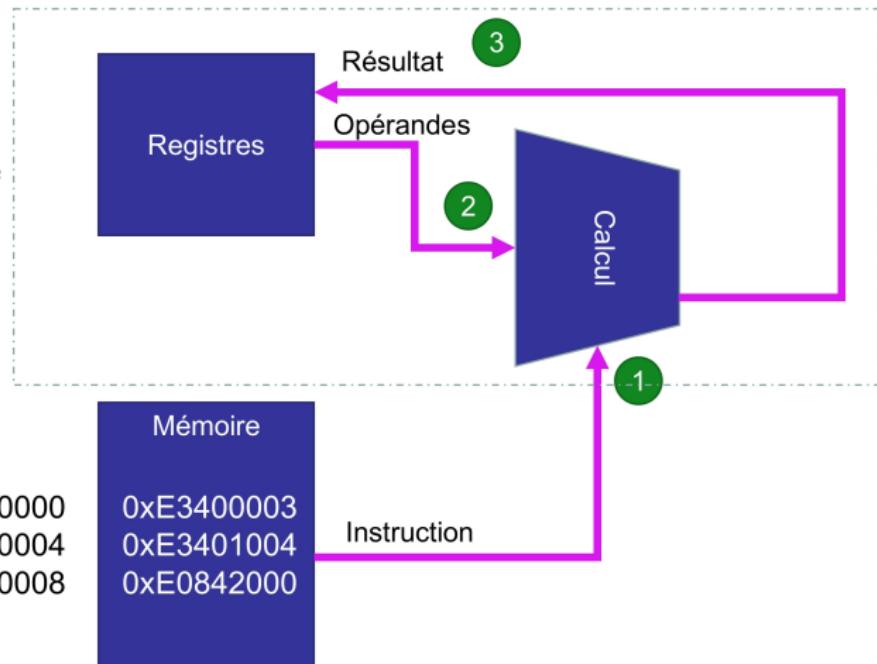
1110 0000 1000 0100 0010 0000 0000 0000 => 0xE0842000

Exécution de add r1, r1, #1



Programme simple

```
mov    r4,#3  
mov    r5,#4  
add    r6,r5,r4
```



Quelques instructions de calcul

Syntaxe	Sémantique
add rd, rs1, rs2	$\text{@ } rd \leftarrow rs1 + rs2$
add rd, rs1, #valeur	$\text{@ } rd \leftarrow rs1 + \text{valeur}$
sub rd, rs1, rs2	$\text{@ } rd \leftarrow rs1 - rs2$
and rd, rs1, rs2	$\text{@ } rd \leftarrow \text{ET bit à bit de } rs1 \text{ et } rs2$
mul rd, rs1, rs2	$\text{@ } rd \leftarrow rs1 * rs2$
mov rd, rs1	$\text{@ } rd \leftarrow rs1$
mov rd, #valeur	$\text{@ } rd \leftarrow \text{valeur}$

rd, rs1, rs2 = un registre (r0, r1, ..., r15)
valeur = un nombre (13, 0xd, 0b01101)

Exercice 1

Que fait le programme suivant ?

```
mov r0, #1  
mov r1, #5  
sub r2, r1, r0
```

Exercice 2

Que fait le programme suivant ?

```
mov r0, #3
add r1, r0, r0
add r1, r1, r1
add r1, r1, #6
```

Exécution conditionnelle : comparaisons

L'**exécution conditionnelle** en assembleur se base sur la comparaison entre deux registres.

L'instruction **cmp rs1, rs2** soustrait rs2 à rs1 et met à jour un registre spécial (le **CPSR**) avec des informations sur le résultat de cette soustraction : le résultat est-il égal à 0 ($rs1 = rs2$) ? Est-il positif ($rs1 > rs2$) ?

Syntaxe	Sémantique
<code>cmp rs1, rs2</code>	@ comparaison de rs1 et rs2
<code>cmp rs1, #value</code>	@ comparaison de rs1 et value

Exécution conditionnelle : comparaisons

L'**exécution conditionnelle** en assembleur se base sur la comparaison entre deux registres.

L'instruction **cmp rs1, rs2** soustrait rs2 à rs1 et met à jour un registre spécial (le **CPSR**) avec des informations sur le résultat de cette soustraction : le résultat est-il égal à 0 ($rs1 = rs2$) ? Est-il positif ($rs1 > rs2$) ?

Syntaxe	Sémantique
<code>cmp rs1, rs2</code>	@ comparaison de rs1 et rs2
<code>cmp rs1, #value</code>	@ comparaison de rs1 et value

Le processeur ne peut pas déterminer si un nombre stocké dans un registre est **signé** ou non.

Les informations mises dans le CPSR permettent de retrouver le résultat de la comparaison dans les deux cas.

Exécution conditionnelle : suffixes

Une fois qu'une comparaison a été réalisée, on peut conditionner l'exécution des instructions suivantes en leur apposant un **suffixe**.

cmp r2, r5	@ comparaison de r2 et r5
addeq r1, r3, r4	@ addition de r3 et r4 seulement si r2 = r5
subne r1, r6, #128	@ soustraction de r6 et 128 seulement si r2 ≠ r5

Exécution conditionnelle : suffixes

Une fois qu'une comparaison a été réalisée, on peut conditionner l'exécution des instructions suivantes en leur apposant un **suffixe**.

cmp r2, r5	@ comparaison de r2 et r5
addeq r1, r3, r4	@ addition de r3 et r4 seulement si r2 = r5
subne r1, r6, #128	@ soustraction de r6 et 128 seulement si r2 ≠ r5

C'est au programmeur (ou au compilateur) de choisir le bon suffixe, et en particulier de faire la différence entre nombres signés et non signés.

Exécution conditionnelle : suffixes

cmp rs1, rs2

Condition	Nombres non signés	Nombres signés
$rs1 = rs2$	eq	eq
$rs1 \neq rs2$	ne	ne
$rs1 > rs2$	hi	gt
$rs1 \geq rs2$	hs	ge
$rs1 < rs2$	lo	lt
$rs1 \leq rs2$	ls	le

eq : equal, ne : not equal, ge : greater or equal
hi : higher, hs : higher or same, lt : lesser than
lo : lower, ls : lower or same, le : lesser or equal
gt : greater than

Exercice 3 : $r3 = \max(r2, r1)$ (entiers non signés)

CPSR et suffixes

suffixe	signification	condition	suffixe	signification	condition
al	always	1	vc	overflow clear	\bar{V}
eq	equal	Z	vs	overflow set	V
ne	not equal	\bar{Z}	hi	higher	$C.\bar{Z}$
pl	plus	\bar{N}	ls	lower or same	$\bar{C} + Z$
mi	minus	N	gt	greater than	$\bar{Z}.N.V + \bar{N}.V$
cc / lo	carry clear	\bar{C}	ge	greater or equal	$N.V + \bar{N}.V$
cs / hs	carry set	C	lt	less than	$N.\bar{V} + \bar{N}.V$
			le	less or equal	$Z + N.\bar{V} + \bar{N}.V$

CPSR :

N	Z	C	V	
---	---	---	---	--

N = Negative

Z = Zero

C = Carry

V = oVerflow

HS ou GE ?

Pour simplifier, on suppose que les registres ne contiennent que 4 bits

cmp r0, r1

r0 1001

r1 0010

Quelle est la valeur de r0 ?

HS ou GE ?

Pour simplifier, on suppose que les registres ne contiennent que 4 bits

cmp r0, r1

r0

r1

Quelle est la valeur de r0 ?

9

HS ou GE ?

Pour simplifier, on suppose que les registres ne contiennent que 4 bits

cmp r0, r1

r0 1001

r1 0010

Quelle est la valeur de r0 ?

9

-7

HS ou GE ?

Pour simplifier, on suppose que les registres ne contiennent que 4 bits

cmp r0, r1

r0 1001

r1 0010

$$\begin{array}{r} 1001 \\ - 0010 \\ \hline 0111 \end{array}$$

CPSR :

0	0	1	1	
N	Z	C	V	

HS ou GE ?

Pour simplifier, on suppose que les registres ne contiennent que 4 bits

cmp r0, r1

r0 1001

r1 0010

$$\begin{array}{r} 1001 \\ - 0010 \\ \hline 0111 \end{array} \quad \begin{array}{r} \text{non signés} \\ 9 \\ - 2 \\ \hline 7 \end{array}$$

CPSR :

0	0	1	1	
N	Z	C	V	

cs / hs	C
---------	---

vérifié $\rightarrow r0 \geq r1$

HS ou GE ?

Pour simplifier, on suppose que les registres ne contiennent que 4 bits

cmp r0, r1

r0 1001

r1 0010

$$\begin{array}{r} 1001 \\ - 0010 \\ \hline 0111 \end{array}$$

non signés

$$\begin{array}{r} 9 \\ - 2 \\ \hline 7 \end{array}$$

signés

$$\begin{array}{r} -7 \\ - +2 \\ \hline \text{Faux} \end{array}$$

CPSR :

0	0	1	1	
N	Z	C	V	

cs / hs C

ge $N.V + \bar{N}.\bar{V}$

vérifié $\rightarrow r0 \geq r1$

non vérifié $\rightarrow r0 < r1$

Exercice 4 : Si ... alors ... sinon ...

C / Algorithme

```
if (i > 0)
    signe = 0;
else
    signe = 1;
signe = signe + 1;
```

Assembleur

```
/* i : entier signé
   i dans r1
   signe dans r2 */
```

Branchements

Le processeur utilise un registre particulier appelé **Program Counter (PC)** pour stocker l'adresse de la prochaine instruction à charger. Pour les processeurs ARM, c'est le registre r15 qui joue ce rôle.

Syntaxe	Sémantique
b étiquette	$\text{@ branchement } (\text{PC} \leftarrow \text{adresse}_{\text{étiquette}})$

Branchements

Le processeur utilise un registre particulier appelé **Program Counter (PC)** pour stocker l'adresse de la prochaine instruction à charger. Pour les processeurs ARM, c'est le registre r15 qui joue ce rôle.

- En temps normal, lors de l'exécution d'une instruction, le PC est **incrémenté** pour pointer vers la prochaine instruction en mémoire.

Syntaxe	Sémantique
b étiquette	$\text{@ branchement } (\text{PC} \leftarrow \text{adresse}_{\text{étiquette}})$

Branchements

Le processeur utilise un registre particulier appelé **Program Counter (PC)** pour stocker l'adresse de la prochaine instruction à charger. Pour les processeurs ARM, c'est le registre r15 qui joue ce rôle.

- En temps normal, lors de l'exécution d'une instruction, le PC est **incrémenté** pour pointer vers la prochaine instruction en mémoire.
- Pour réaliser les structures de contrôle (if, for, while), une instruction de **branchement** permet de charger directement une adresse dans le PC. Cela permet de rediriger l'exécution vers une instruction qui n'est pas la suivante en mémoire.

Syntaxe	Sémantique
b étiquette	$\text{@ branchement } (\text{PC} \leftarrow \text{adresse}_{\text{étiquette}})$

Exemple : la boucle tant que

```
i=0;  
while(i<=50){  
    ...  
    ...  
    i = i + 1;  
}  
  
bcl : mov r1, #0  
      cmp r1, #50  
      bhi fin  
      ...  
      add r1, r1, #1  
      b bcl  
fin :
```

Exemple : la boucle tant que

```
⇒      mov r1, #0      @ r1 ← 0
bcl :  cmp r1, #50
        bhi fin
        ...
add r1, r1, #1
b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
⇒    bcl : cmp r1, #50      @ comparer r1 à 50
        bhi fin
...
        add r1, r1, #1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
⇒      bhi fin          @ brancher à fin si >
...
add r1, r1, #1
b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
⇒      ...
        add r1, r1, #1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
⇒      add r1, r1, #1    @ r1 ← r1 + 1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
add r1, r1, #1
⇒      b bcl          @ brancher à bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
⇒    bcl : cmp r1, #50      @ comparer r1 à 50
        bhi fin
...
        add r1, r1, #1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
⇒      bhi fin          @ brancher à fin si >
...
add r1, r1, #1
b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
⇒      ...
        add r1, r1, #1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
⇒      add r1, r1, #1    @ r1 ← r1 + 1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
add r1, r1, #1
⇒      b bcl          @ brancher à bcl
fin :
```



Exemple : la boucle tant que



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
⇒      add r1, r1, #1    @ r1 ← r1 + 1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
add r1, r1, #1
⇒      b bcl          @ brancher à bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
⇒    bcl : cmp r1, #50      @ comparer r1 à 50
        bhi fin

...
add r1, r1, #1
b bcl

fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
⇒      bhi fin          @ brancher à fin si >
...
add r1, r1, #1
b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
⇒      ...
        add r1, r1, #1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
⇒      add r1, r1, #1    @ r1 ← r1 + 1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
add r1, r1, #1
⇒      b bcl          @ brancher à bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
⇒    bcl : cmp r1, #50      @ comparer r1 à 50
        bhi fin
...
        add r1, r1, #1
        b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
⇒      bhi fin          @ brancher à fin si >
...
add r1, r1, #1
b bcl
fin :
```



Exemple : la boucle tant que

```
        mov r1, #0
bcl :  cmp r1, #50
        bhi fin
...
        add r1, r1, #1
        b bcl
⇒    fin :
```



Exercice 5 : calcul de la somme des 49 premiers entiers

C / Algorithme

```
somme = 0;  
for (i=0 ; i<=48 ; i++)  
    somme = somme + i;
```

Assembleur

Pour passer à l'assembleur :

i : r2

somme : r1

Exercice 6 : If then else avec branchements

Assembleur

C / Algorithme

```
if (i < 0)
    x = i*i + 2*i + 12;
else
    x = i*6 - 5;
x = x + 3;
```

```
/* i : entier signé
   i dans r1
   x dans r2 */
```

Exercice 7 : Gestion des opérateurs logiques dans les conditions

C / Algorithme

```
if (i > 5 && i < 10)
    x = 15 ;
else
    x = 0 ;
```

Assembleur

```
/* i : entier signé
   i dans r1
   x dans r2 */
```

Exercice 8 : Somme des multiples

Ecrire un programme qui parcourt tous les entiers entre 0 et N et calcule les sommes de ceux qui sont des :

- multiples de 2 (dans r1)
- multiples de 4 (dans r2)
- multiples de 8 (dans r3)

On définira N en utilisant la syntaxe suivante pour lui donner la valeur 128 :

.equ N, 128

qui est l'équivalent assembleur de `#define N 128` en C.

Exercice 8 : Somme des multiples

```
#define N 128
i = 0;
somme2 = 0;
somme4 = 0;
somme8 = 0;
while(i <= N){
    if(i%2 == 0){
        somme2 = somme2 + i;
    }
    if(i%4 == 0){
        somme4 = somme4 + i;
    }
    if(i%8 == 0){
        somme8 = somme8 + i;
    }
}
i = i + 1;
}

.equ N,128
r1 : somme2
r2 : somme4
r3 : somme8
r4 : i
```

Exercice 9 : Division entière de r4 par r1 (en-tiers non signés)

Algorithme

```
quotient = 0;  
reste = r4;  
while(reste >= r1){  
    quotient = quotient + 1;  
    reste = reste - r1;  
}
```

```
quotient : r2  
reste : r3
```

Exercice 10 : Calcul itératif de la factorielle d'un entier naturel présent dans r0

Algorithme

```
i = r0;  
fact = 1;  
while(i >= 2){  
    fact = fact * i;  
    i = i - 1;  
}
```

i : r2
fact : r1

Mémoire

- La mémoire est organisée comme une **suite d'octets**
- Chaque octet de mémoire est numéroté. Le numéro d'un octet est **son adresse**
 - La mémoire peut être vue comme un grand tableau d'octets

Mémoire

- La mémoire est organisée comme une **suite d'octets**
- Chaque octet de mémoire est numéroté. Le numéro d'un octet est **son adresse**
 - La mémoire peut être vue comme un grand tableau d'octets
- Le processeur peut accéder à plusieurs octets simultanément
 - Le programmeur peut spécifier la taille de l'accès
 - byte : b → 1 octet
 - half-word : h → 2 octets
 - word : 4 octets

Mémoire

- La mémoire est organisée comme une **suite d'octets**
- Chaque octet de mémoire est numéroté. Le numéro d'un octet est **son adresse**
 - La mémoire peut être vue comme un grand tableau d'octets
- Le processeur peut accéder à plusieurs octets simultanément
 - Le programmeur peut spécifier la taille de l'accès
 - byte : b → 1 octet
 - half-word : h → 2 octets
 - word : 4 octets
 - Le processeur ARM a des limitations
 - L'accès à un demi-mot (h) ne peut se faire qu'à une **adresse paire**
 - L'accès à un mot (w) ne peut se faire qu'à une **adresse multiple de 4**

Mémoire

- La mémoire est organisée comme une **suite d'octets**
- Chaque octet de mémoire est numéroté. Le numéro d'un octet est **son adresse**
 - La mémoire peut être vue comme un grand tableau d'octets
- Le processeur peut accéder à plusieurs octets simultanément
 - Le programmeur peut spécifier la taille de l'accès
 - byte : b → 1 octet
 - half-word : h → 2 octets
 - word : w → 4 octets
 - Le processeur ARM a des limitations
 - L'accès à un demi-mot (h) ne peut se faire qu'à une **adresse paire**
 - L'accès à un mot (w) ne peut se faire qu'à une **adresse multiple de 4**
- Les architectures ARM sont des architecture Load/Store
 - Seules les instructions d'accès mémoire (load et store) peuvent accéder à la mémoire

Réserver de la mémoire pour des variables scalaires

- Représentation des adresses
 - Pour faciliter la programmation, on peut représenter l'adresse de certaines instructions ou variables par une **étiquette** (chaîne de caractères)

Réserver de la mémoire pour des variables scalaires

- Représentation des adresses
 - Pour faciliter la programmation, on peut représenter l'adresse de certaines instructions ou variables par une **étiquette** (chaîne de caractères)
- Directives d'assemblage
 - Définition d'une **variable** sur 4 octets **initialisée** en réservant de la mémoire
étiquette : `.word valeur_initiale`

Exemple :

V : `.word 8`

Autres directives : `.byte`, `.float`, `.fill`, `.ascii`, `.asciz`

Réserver de la mémoire pour des variables scalaires

- Représentation des adresses
 - Pour faciliter la programmation, on peut représenter l'adresse de certaines instructions ou variables par une **étiquette** (chaîne de caractères)
- Directives d'assemblage
 - Définition d'une **variable** sur 4 octets **initialisée** en réservant de la mémoire
étiquette : `.word valeur_initiale`
- Exemple :
`V : .word 8`
Autres directives : `.byte`, `.float`, `.fill`, `.ascii`, `.asciz`
- À ne pas confondre avec :
`.equ V, 8` qui est une facilité d'écriture mais **ne réserve pas de mémoire** : comme avec `#define` en C, le pré-processeur remplace chaque occurrence de "V" par "8" avant de lancer l'assemblage.

Réserver de la mémoire pour un tableau

- En C :

```
int tab1[3] = {0, 0, 0};
```

- En assembleur ARM :

```
tab1 : .fill 3, 4, 0
```

x	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

Réserver de la mémoire pour un tableau

- En C :
`int tab1[3] = {0, 0, 0};`
- En assembleur ARM :
`tab1 : .fill 3, 4, 0`
- En C :
`char tab2[3];`
- En assembleur ARM :
`tab2 : .fill 3, 1`

x	0	0	0	0	0	0	0	0	0	x+4	0	0	x+8	0	0	0	0	x+12	??	x+13	??	x+14	??
	0	0	0	0	0	0	0	0	0														

Réserver de la mémoire pour un tableau

- En C :
`int tab1[3] = {0, 0, 0};`

- En assembleur ARM :
`tab1 : .fill 3, 4, 0`

- En C :
`char tab2[3];`

- En assembleur ARM :
`tab2 : .fill 3, 1`

- En C :
`char tab3[3]= {3, -2, 12};`

- En assembleur ARM :
`tab3 : .byte 0b11, -2, 0xC`

x	x+4	x+8	x+12	x+13	x+14	x+15	x+16	x+17
0	0	0	0	0	0	0	3	-2

Lecture en mémoire

ldr **rd**, [adresse] @ lecture d'un mot (4 octets) adresse%4 == 0
ldr**h** rd, [adresse] @ lecture d'un demi-mot (2 octets) adresse%2 == 0

ldr**b** rd, [adresse] @ lecture d'un octet

	31	15	0
rd	00000000	00000000	00000000
			octet lu

rd	00000000	00000000	demi-mot lu
----	----------	----------	-------------

Écriture en mémoire

str **rs**, [adresse] @ écriture d'un mot (4 octets) adresse%4 == 0
str**h** **rs**, [adresse] @ écriture d'un demi-mot (2 octets) adresse%2 == 0

str**b** **rs**, [adresse] @ écriture d'un octet

	31	15	0	
rs	00000000	00000000	00000000 octet écrit	
rs	00000000	00000000	demi-mot écrit	

Modes d'adressage

Le calcul de l'adresse à accéder (entre les crochets) se fait toujours à partir d'un **registre de base**. Par exemple :

```
ldrb r0, [r1]
```

charge dans **r0** l'octet présent à l'adresse couramment dans **r1**. Dans ce cas, on appelle **r1** le registre de base.

Modes d'adressage

Afin de faciliter les accès dans les tableaux, il est possible d'ajouter une valeur au registre de base. Par exemple :

```
ldrb r0, [r1, #1]
```

charge dans **r0** l'octet présent à l'adresse suivant celle qui est couramment dans **r1**. La valeur de **r1** n'est **pas modifiée** suite à cette instruction.

Modes d'adressage

Il est également possible de modifier l'adresse contenue dans le registre de base **après** que l'accès ait eu lieu. Par exemple :

```
ldrb r0, [r1, #1]!
```

charge dans **r0** l'octet présent à l'adresse suivant celle qui est couramment dans **r1** puis **incrémente r1**.

Modes d'adressage

Il est également possible de modifier l'adresse contenue dans le registre de base **après** que l'accès ait eu lieu. Par exemple :

```
ldrb r0, [r1], #1
```

charge dans **r0** l'octet présent à l'adresse couramment dans **r1** puis **incrémente r1**.

Modes d'adressage

calcul de l'adresse (dans les crochets)

[r0]	adresse = r0
[r0, r1]	adresse = r0 + r1
[r0, r1, lsl #2]	adresse = r0 + r1*4
[r0, #4]	adresse = r0 + 4

mise à jour du registre de base (après les crochets)

!	registre de base \leftarrow adresse
, #2	registre de base \leftarrow registre de base + 2

Modes d'adressage

	r0 avant	adresse mémoire	r0 après accès
ldr r1, [r0]	x	x	x
ldr r1, [r0, #4]	x	x+4	x
ldr r1, [r0], #4	x	x	x+4
ldr r1, [r0, #4]!	x	x+4	x+4

La pseudo-instruction adr

Pour faire le lien entre les adresses représentées par des **étiquettes** et les instructions mémoire qui prennent un **registre** pour calculer l'adresse à accéder, on utilise une pseudo-instruction : **adr**.

```
var : .word 4      @ int var = 4;  
...  
adr r1, var      @ r1 reçoit l'adresse mémoire où se trouve la valeur 4  
                      @ r1 = &var;  
ldr r2, [r1]       @ charge la valeur 4 dans r2 depuis la mémoire  
add r2, r2, #1     @ écrit 5 en mémoire à l'adresse pointée par r1  
str r2, [r1]
```

var = var + 1

Exemple : ajouter 1 à chaque élément d'un tableau d'octets

Version adresse de base + index

adr	r5, tab
mov	r1, #0
tq :	cmp r1, #tailletab
bhs	fin
ldrb	r2, [r5, r1]
add	r2, r2, #1
strb	r2, [r5, r1]
add	r1, r1, #1
b	tq
fin :	...

r5	tab
r1	0, 1, ..., 9

tab
tab+1
tab+2

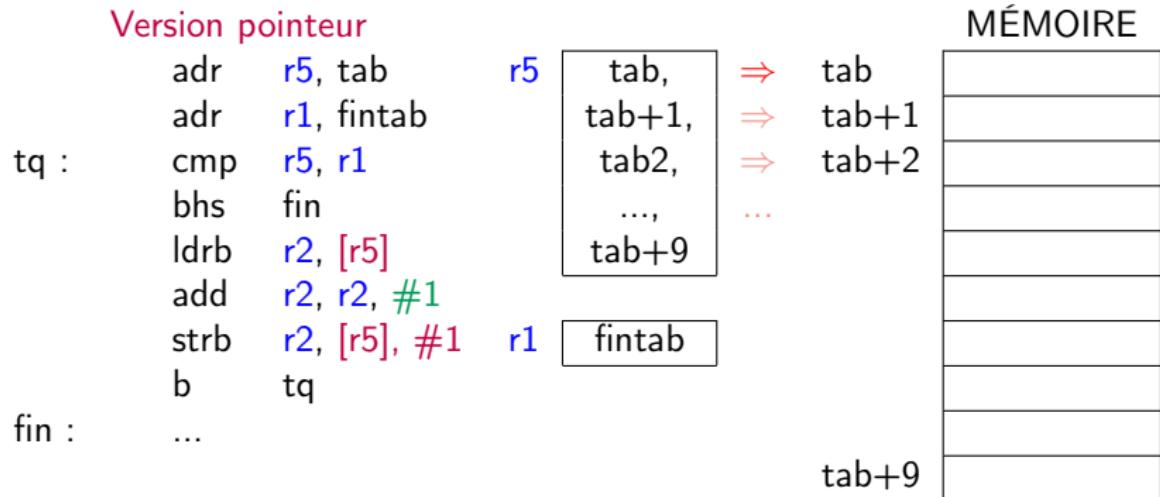
MÉMOIRE

tab+9

.equ tailletab, 10

tab : .fill tailletab, 1

Exemple : ajouter 1 à chaque élément d'un tableau d'octets



tab : .fill 10, 1

fintab :

Exercice 11 : Initialisation d'un tableau

- Écrire un programme qui initialise les éléments d'un tableau de 10 octets avec les 10 premiers nombres
- Proposer 2 versions :
 - une qui référence les éléments à partir d'un **pointeur** qui balaye le tableau
 - l'autre qui référence les éléments à partir de l'adresse de base du tableau et de leur **indice** dans le tableau

Exercice 12 : Recherche du maximum dans un tableau

Écrire un programme qui recherche l'indice du maximum dans un tableau de 10 entiers **signés**

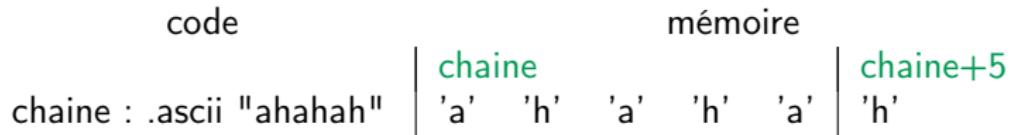
- Utiliser le mode d'adressage indexé (adresse de base + indice)
- Le résultat est le numéro du plus grand élément (le premier si plusieurs éléments ont la même valeur)

Exercice 12 : Recherche du maximum dans un tableau

```
int numero = 0;  
int maximum = tab[0];  
unsigned int i = 1;  
while (i < 10) {  
    if (tab[i] > maximum) {  
        maximum = tab[i];  
        numero = i;  
    }  
    i = i + 1;  
}
```

Réserver de la mémoire pour une chaîne de caractères

- La directive `.ascii` permet de réserver de la mémoire pour une chaîne de caractères



Réserver de la mémoire pour une chaîne de caractères

- La directive `.ascii` permet de réserver de la mémoire pour une chaîne de caractères
- La directive `.asciz` permet de réserver de la mémoire pour une chaîne de caractères **et un 0 marquant la fin de la chaîne**

code	mémoire				
chaine : .ascii "ahahah"	<code>chaine</code>				
	'a'	'h'	'a'	'h'	'a'
chaine : .asciz "ahahah"	<code>chaine+5</code>				
	'a'	'h'	'a'	'h'	'a'
					'h'
					0

Réserver de la mémoire pour une chaîne de caractères

- La directive **.ascii** permet de réserver de la mémoire pour une chaîne de caractères
- La directive **.asciz** permet de réserver de la mémoire pour une chaîne de caractères **et un 0 marquant la fin de la chaîne**
- La directive **.align** permet de **réaligner la prochaine adresse utilisée** sur un multiple de 4, pour y **stocker des entiers** (mots de 4 octets), ou y **placer des instructions** (32 bits/4 octets).

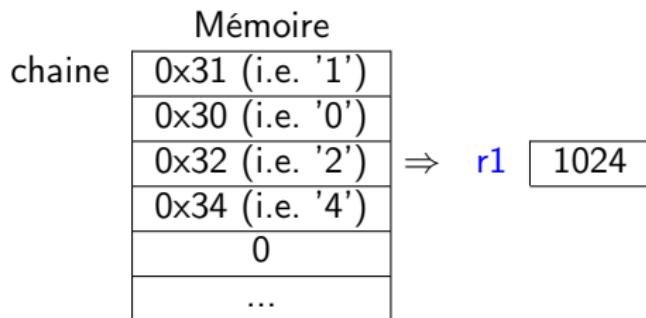
code	mémoire				
chaine : .ascii "ahahah"	chaine	'a'	'h'	'a'	'h'
chaine : .asciz "ahahah"	chaine	'a'	'h'	'a'	'h'
					0
chaine : .asciz "ahahah"	chaine+5	'h'			
.align	chaine+5	'h'	0	0	0

Exercice 13 : Recherche d'un caractère dans une chaîne

- Écrire un programme qui compte (dans **r1**) le nombre de 'a' dans une chaîne de caractères et place dans **r2** la position du premier 'a'
- La chaîne est déclarée comme suit :
chaine : .asciz "blablabla"
- Affectation des registres :
 - **r1** : nombre de 'a'
 - **r2** : position du premier 'a'
 - **r3** : indice de boucle i
 - **r4** : pointeur sur le caractère en cours de traitement

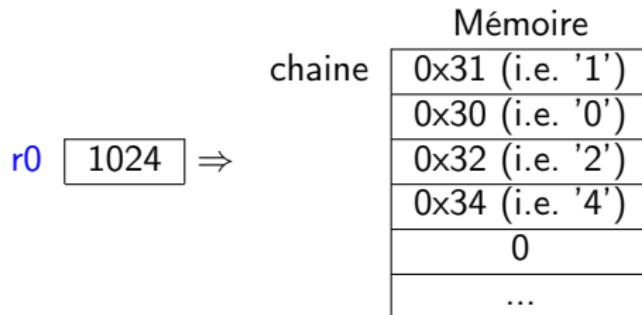
Exercice 14 : Fonction atoi()

Écrire un programme qui transforme une chaîne de caractères de chiffres en la valeur entière correspondante. On suppose que tous les caractères sont bien des chiffres et que la chaîne se termine par la valeur 0.



Exercice 15 : Fonction itoa()

Écrire un programme qui transforme un nombre présent dans le registre `r0` en une chaîne de caractères qui se termine par le code ASCII 0.



Exercice 15 : Fonction itoa()

Écrire un programme qui transforme un nombre présent dans le registre `r0` en une chaîne de caractères qui se termine par le code ASCII 0.

- On procèdera par **divisions entières** successives pour récupérer le chiffre des unités, puis des dizaines, puis des centaines, etc.
 - Quand on récupère un chiffre, comment le transformer facilement dans le caractère ASCII correspondant ?
- On ne sait pas à l'avance combien de caractères seront nécessaires pour coder le nombre présent dans `r0`, et donc où placer le caractère correspondant aux unités en mémoire.
 - Quel est le plus grand entier codable sur 32 bits ? De combien de chiffres est-il composé ?

Ordre de stockage en mémoire

Comment ranger le mot 0x03020100 en mémoire ?

Ordre de stockage en mémoire

Comment ranger le mot 0x03020100 en mémoire ?

Poids forts

Poids faibles

- Dans les architectures **Little Endian** (Petit boutiste), on stocke en mémoire d'abord les octets de poids **faible**

Ordre de stockage en mémoire

Comment ranger le mot 0x03020100 en mémoire ?

Poids forts

Poids faibles

- Dans les architectures **Little Endian** (Petit boutiste), on stocke en mémoire d'abord les octets de poids **faible**

Adresse	Contenu
100	0x00
101	0x01
102	0x02
103	0x03

Ordre de stockage en mémoire

Comment ranger le mot 0x03020100 en mémoire ?

Poids forts

Poids faibles

- Dans les architectures **LittleEndian** (Petit boutiste), on stocke en mémoire d'abord les octets de poids **faible**

Adresse	Contenu
100	0x00
101	0x01
102	0x02
103	0x03

- Dans les architectures **BigEndian** (Gros boutiste), on stocke en mémoire d'abord les octets de poids **fort**

Ordre de stockage en mémoire

Comment ranger le mot 0x03020100 en mémoire ?

Poids forts

Poids faibles

- Dans les architectures **LittleEndian** (Petit boutiste), on stocke en mémoire d'abord les octets de poids **faible**

Adresse	Contenu
100	0x00
101	0x01
102	0x02
103	0x03

- Dans les architectures **BigEndian** (Gros boutiste), on stocke en mémoire d'abord les octets de poids **fort**

Adresse	Contenu
100	0x03
101	0x02
102	0x01
103	0x00

Ordre de stockage en mémoire

Comment ranger le mot 0x03020100 en mémoire ?

Poids forts

Poids faibles

Adresse	Petit bout			
100	00	01	02	03
104				

Adresse	Gros bout			
100	03	02	01	00
104				

Exercice 16 : Little endian

- Le registre **r5** contient la valeur 0xFF223344
- Le registre **r1** contient la valeur 0x50000
- L'instruction suivante est exécutée :
`str r5, [r1]`
- Représenter les valeurs en mémoire en supposant que le processeur est configuré en mode little endian

Exercice 16 : Little endian

- Le registre **r5** contient la valeur 0xFF223344
- Le registre **r1** contient la valeur 0x50000
- L'instruction suivante est exécutée :
`str r5, [r1]`
- Représenter les valeurs en mémoire en supposant que le processeur est configuré en mode little endian

0x50000	
0x50001	
0x50002	
0x50003	
0x50004	

Exercice 16 : Little endian

- Le registre **r5** contient la valeur 0xFF223344
- Le registre **r1** contient la valeur 0x50000
- L'instruction suivante est exécutée :
`str r5, [r1]`
- Représenter les valeurs en mémoire en supposant que le processeur est configuré en mode little endian
- On exécute ensuite l'instruction
`ldrb r5, [r1]`
- Quelle valeur y a-t-il dans **r5** ?



Appel et retour de sous-programmes

- Les appels de sous-programmes en ARM sont gérés par le biais de **branchements** (instruction b)
- Contrairement aux boucles et sauts conditionnels, on doit pouvoir revenir à l'instruction de branchement une fois le sous-programme terminé

Appel et retour de sous-programmes

- Les appels de sous-programmes en ARM sont gérés par le biais de **branchements** (instruction b)
- Contrairement aux boucles et sauts conditionnels, on doit pouvoir revenir à l'instruction de branchement une fois le sous-programme terminé
- Cela est réalisé en ajoutant le suffixe **l** (link) à l'instruction de branchement
- En utilisant bl, le processeur **enregistre l'adresse de l'instruction suivante** dans le registre de lien (lr a.k.a. **r14**), puis **saute vers l'adresse de l'instruction cible**

Appel et retour de sous-programmes

```
x=5;  
sous_prog();  
y=y+1;  
z=y+1;
```



```
sous_prog(){  
...  
}  
  
sous_prog :  
...  
    mov pc, lr
```

Appel et retour de sous-programmes

```
x=5;  
sous_prog();  
y=y+1;  
z=y+1;
```

```
sous_prog(){  
...  
}
```

```
lr ← @retour
```

```
mov r2, #5  
bl sous_prog  
add r3, r2, #1  
add r4, r3, #1
```

```
sous_prog ...  
mov pc, lr
```

Appel et retour de sous-programmes

```
x=5;  
sous_prog();  
y=y+1;  
z=y+1;
```

```
sous_prog(){  
...  
}
```

```
mov r2, #5  
bl sous_prog  
add r3, r2, #1  
add r4, r3, #1
```

```
sous_prog  
...  
mov pc, lr
```

Appel et retour de sous-programmes

```
x=5;  
sous_prog();  
y=y+1;  
z=y+1;
```

```
sous_prog(){  
...  
}
```

```
mov r2, #5  
bl sous_prog  
add r3, r2, #1  
add r4, r3, #1
```

```
pc ← @retour
```

```
sous_prog ...  
mov pc, lr
```

Variables locales

L'exécution de sous-programmes utilise des registres pour stocker les opérandes et les résultats des instructions.

- Le nombre de registre est fixé pour le processeur : les différents programmes utilisent les **mêmes registres** matériels

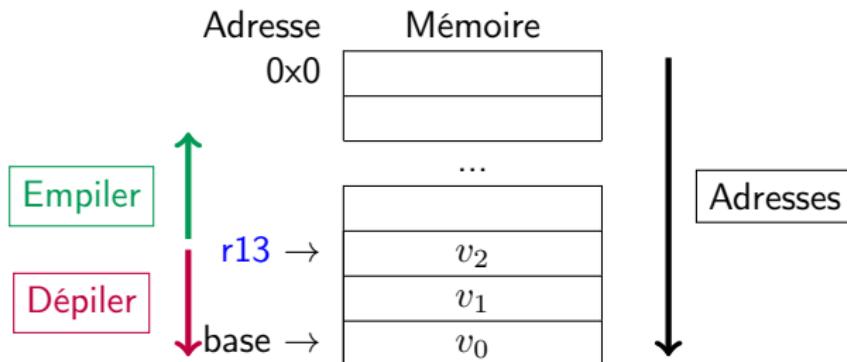
Variables locales

L'exécution de sous-programmes utilise des registres pour stocker les opérandes et les résultats des instructions.

- Le nombre de registre est fixé pour le processeur : les différents programmes utilisent les **mêmes registres** matériels
- Un sous-programme doit garantir que ses calculs ne modifient pas les valeurs des registres dont les autres programmes ont besoin
- On va **sauvegarder** les valeurs des registres **modifiés par le sous-programme** au début et les **restaurer** à la fin, à l'aide d'une **pile**

Mécanisme de pile

- On dédie une partie de la **mémoire** au stockage des valeurs dont on aura besoin au moment du retour d'un sous-programme. Cette mémoire est utilisée sous la forme d'une pile.
- Le registre **r13** aussi appelé **stack pointer (sp)** contient l'adresse du **sommet** (dernier élément empilé) de **la pile en mémoire**



Mécanisme de pile

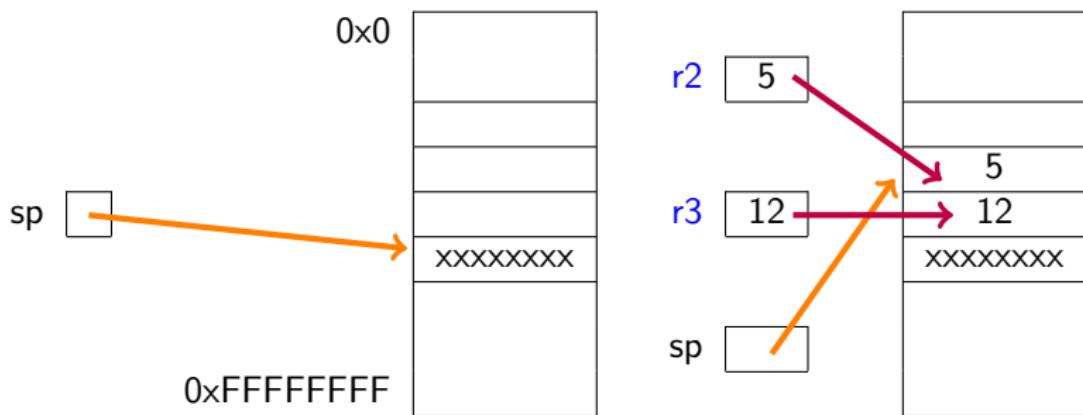
- On dédie une partie de la **mémoire** au stockage des valeurs dont on aura besoin au moment du retour d'un sous-programme. Cette mémoire est utilisée sous la forme d'une pile.
- Le registre **r13** aussi appelé **stack pointer (sp)** contient l'adresse du **sommet** (dernier élément empilé) de **la pile en mémoire**
- On peut empiler une valeur (par exemple présente dans **r0**) :
`str r0, [sp, #-4]` !
- On peut dépiler le sommet de la pile (par exemple dans **r0**) :
`ldr r0, [sp], #4`

Mécanisme de pile

- On dédie une partie de la **mémoire** au stockage des valeurs dont on aura besoin au moment du retour d'un sous-programme. Cette mémoire est utilisée sous la forme d'une pile.
- Le registre **r13** aussi appelé **stack pointer (sp)** contient l'adresse du **sommet** (dernier élément empilé) de **la pile en mémoire**
- On peut empiler une valeur (par exemple présente dans **r0**) :
`str r0, [sp, #-4]` !
- On peut dépiler le sommet de la pile (par exemple dans **r0**) :
`ldr r0, [sp], #4`
- Pour empiler plusieurs valeurs :
`stmfd sp!, {r0-r2, r4}`
- Pour dépiler plusieurs valeurs :
`ldmfd sp!, {r0-r2, r4}`

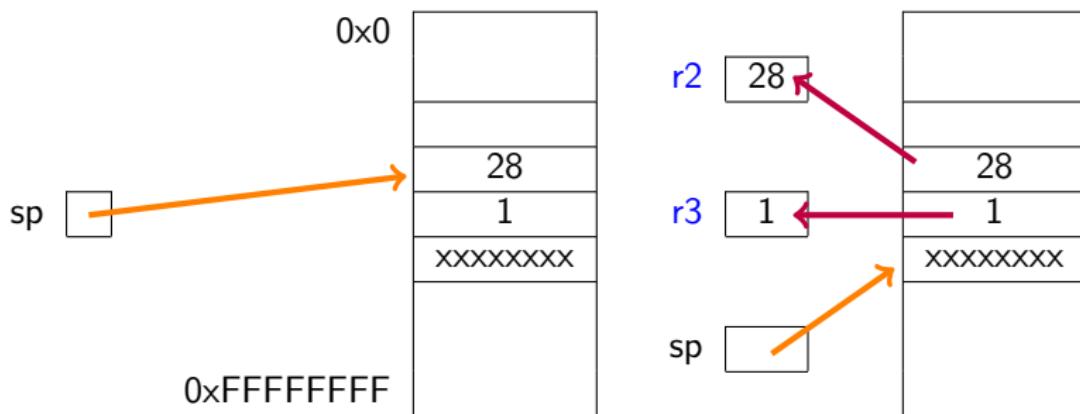
Exemple d'empilement

stmfd sp !, {r2, r3}



Exemple de dépilement

ldmfd sp !, {r2, r3}

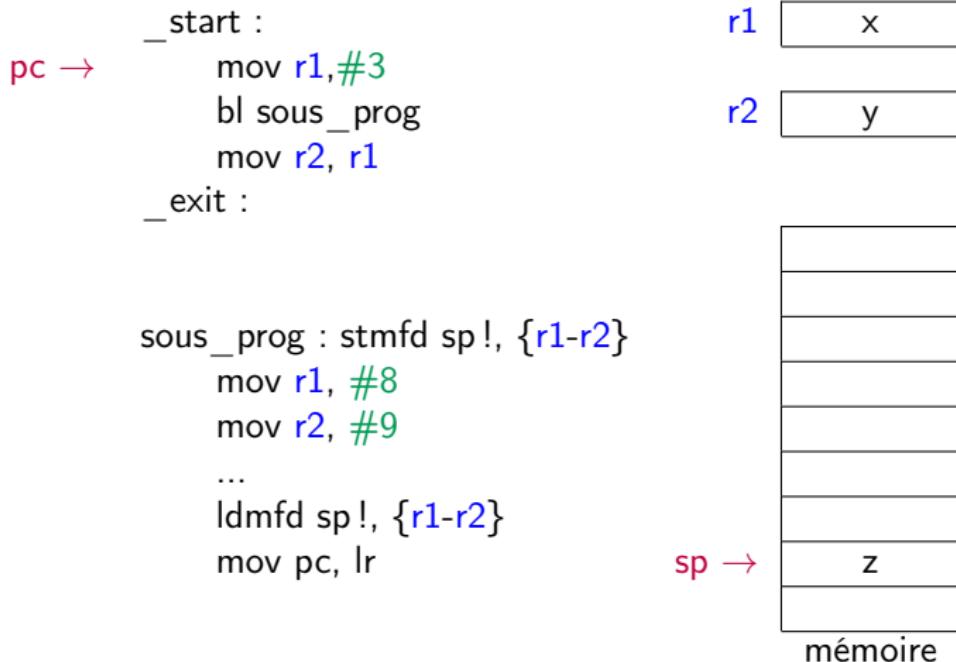


Variables locales : exemple

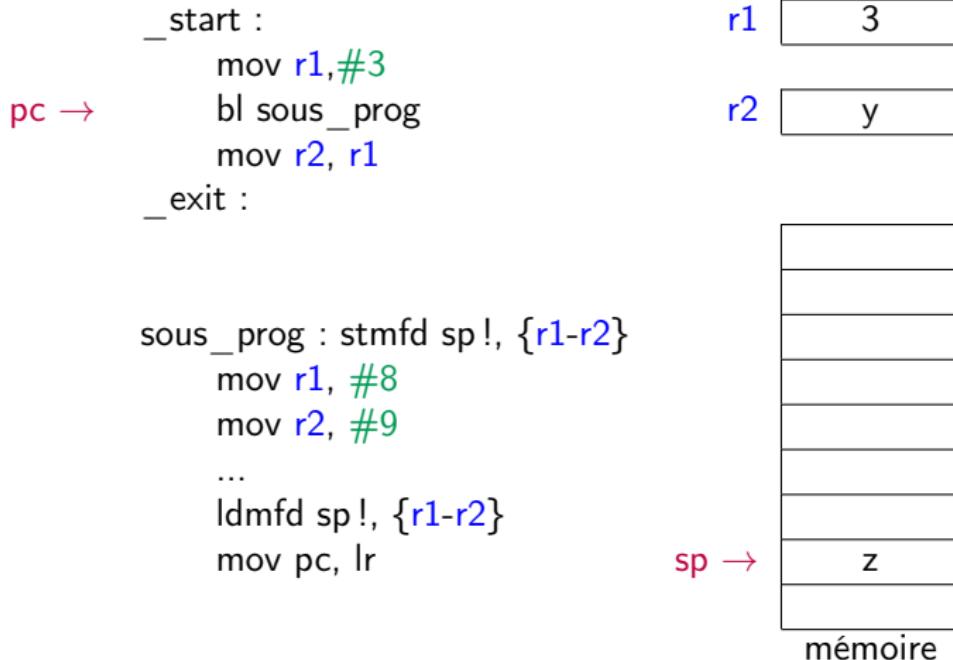
```
void sous_prog() {  
    int a, b;          Les variables a et b existent uniquement  
}  
                                dans le sous-programme
```

```
sous_prog : stmfd sp !, {r7, r8}      @ sauvegarde des registres  
...  
@ ici on peut utiliser  
@ r7 (a) et r8 (b)  
...  
ldmfd sp !, {r7, r8}      @ restauration des registres  
mov pc, lr
```

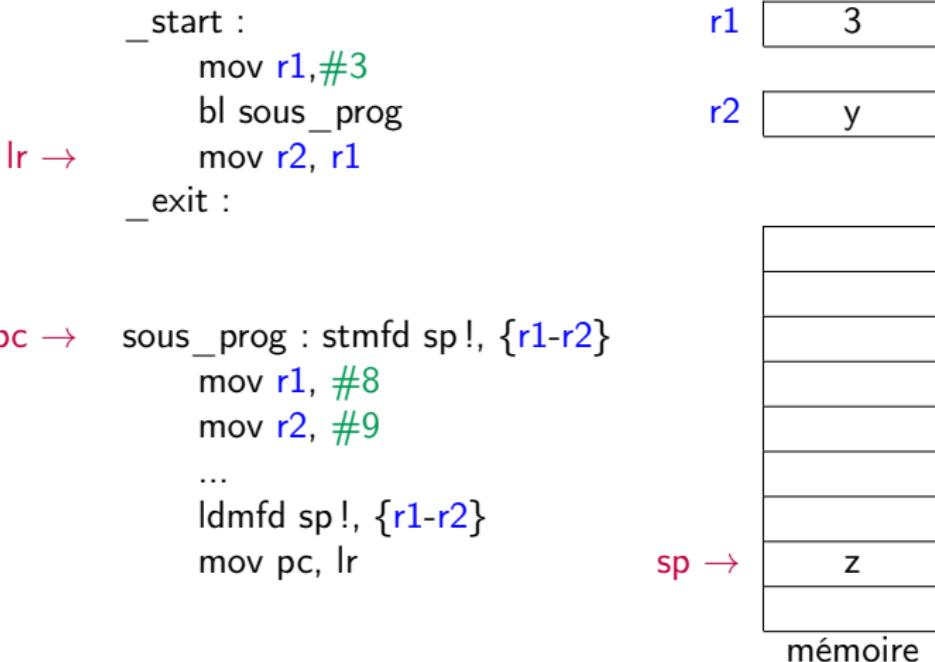
Variables locales : exemple



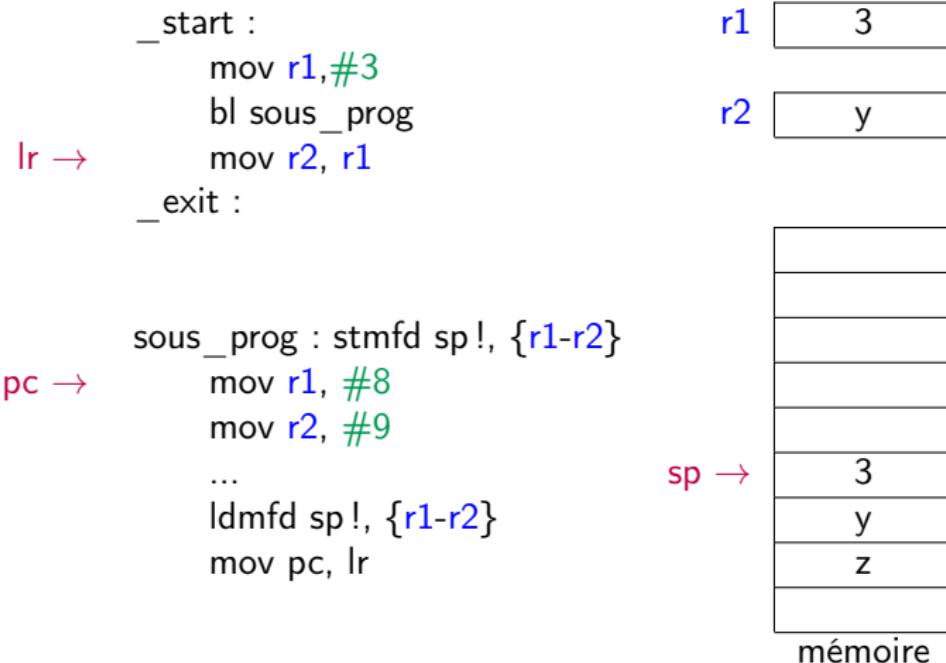
Variables locales : exemple



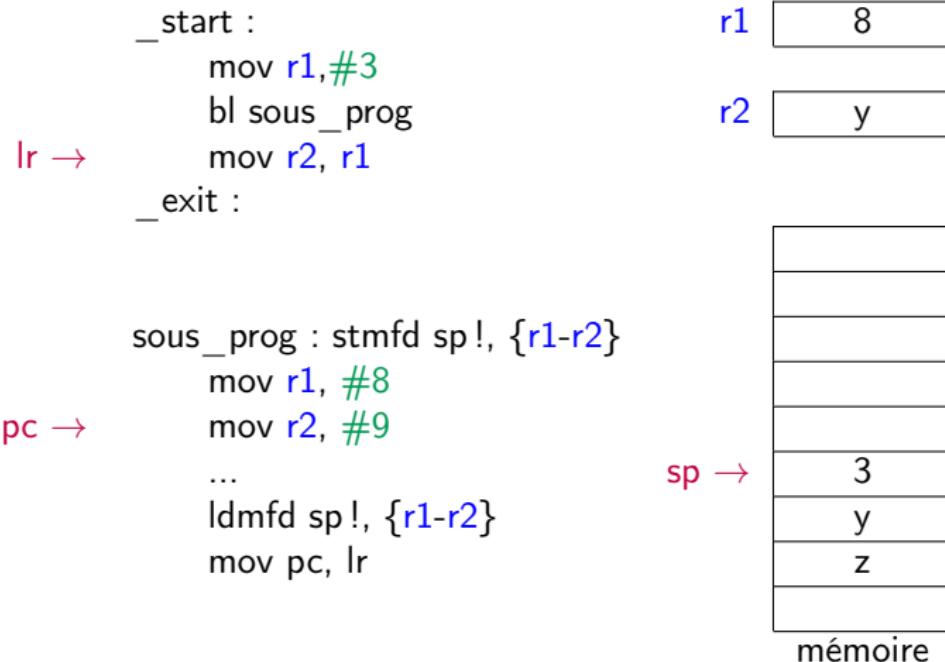
Variables locales : exemple



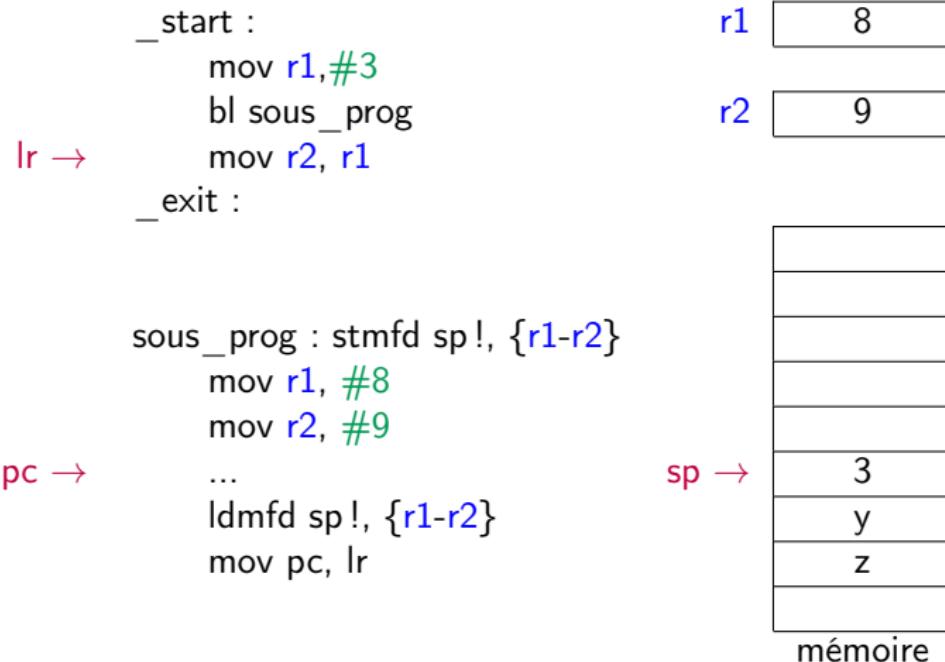
Variables locales : exemple



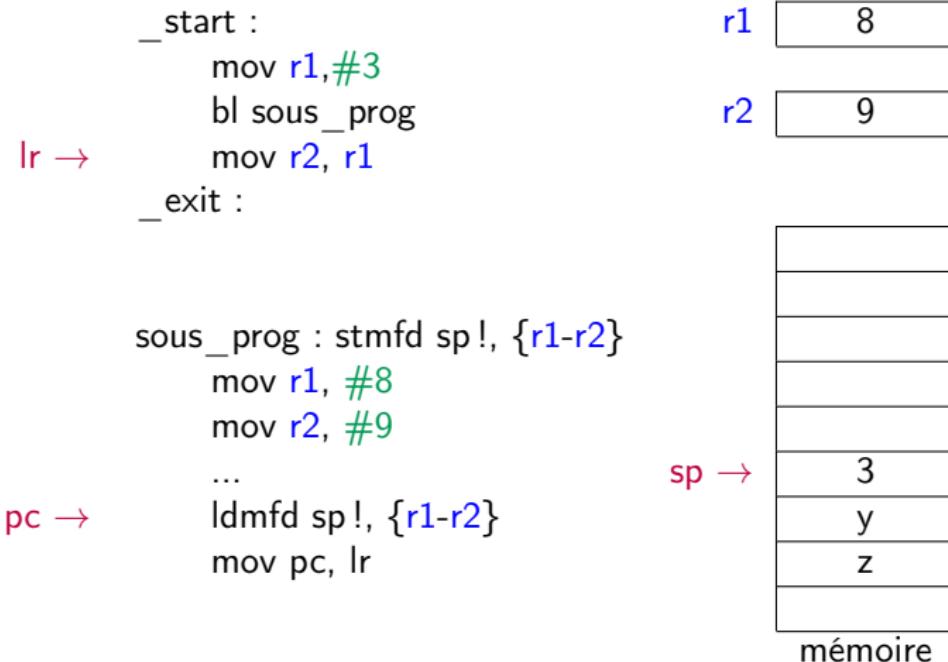
Variables locales : exemple



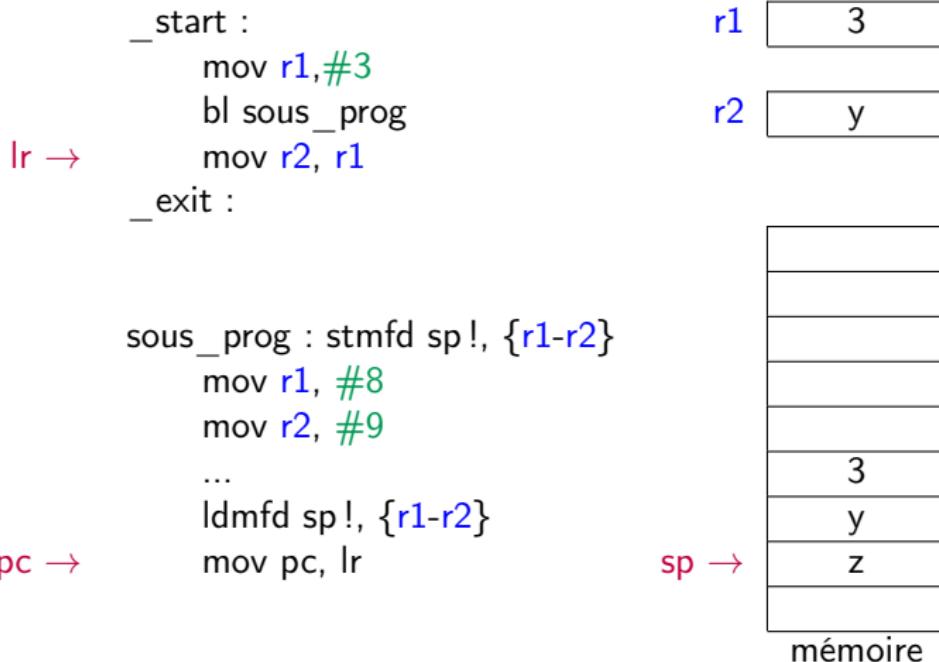
Variables locales : exemple



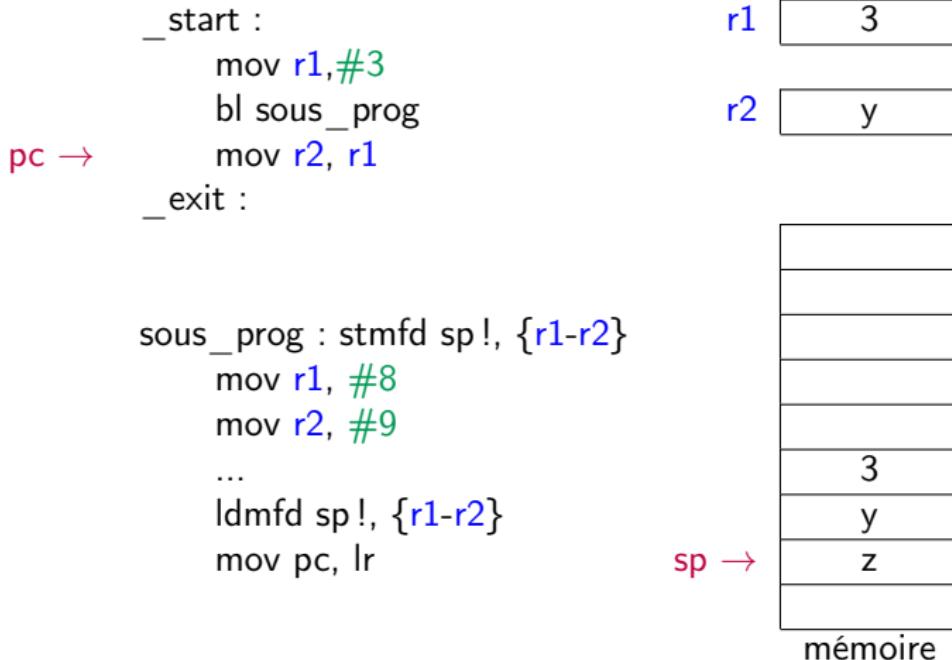
Variables locales : exemple



Variables locales : exemple



Variables locales : exemple



Passage de paramètres par registres

- Lors de l'appel d'un sous-programme, le programme appelant peut passer des **paramètres d'appel** par les registres
- Le sous-programme utilise directement ces registres
- NB : usuellement les registres **r0** à **r3** sont utilisés pour le passage de paramètres

Exercice 17 : Sous-programme d'initialisation d'un tableau

- ① Ecrire un sous-programme qui initialise tous les éléments (entiers) d'un vecteur dont l'adresse TAB et la taille N sont passées dans les registres **r0** et **r1** à une valeur passée dans **r2**.
- ② Ecrire un programme principal qui appelle ce sous-programme.

Exercice 18 : Sous-programme itoa()

- Ecrire un sous-programme qui transforme un nombre présent dans un registre en une chaîne de caractères qui se termine par le code ASCII 0.
- Le nombre est passé en paramètre dans le registre **r0**
- L'adresse de la chaîne est passée en paramètre dans le registre **r1**

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
pc → main : ...
    bl spg1
ret1 : ...

spg1 : stmfd sp !, {...}
...
    bl spg2
ret2 : ...
...
    ldmfd sp !, {...}
    mov pc, lr

spg2 : stmfd sp !, {...}
...
    ldmfd sp !, {...}
    mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
pc →      bl spg1
ret1 : ...

spg1 : stmfd sp !, {...}
...
bl spg2
ret2 : ...
...
ldmfd sp !, {...}
mov pc, lr

spg2 : stmfd sp !, {...}
...
ldmfd sp !, {...}
mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
lr →  ret1 : ...

pc →  spg1 : stmfd sp !, {...}
          ...
          bl spg2
ret2 : ...
        ...
        ldmfd sp !, {...}
        mov pc, lr

spg2 : stmfd sp !, {...}
        ...
        ldmfd sp !, {...}
        mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
|lr →  ret1 : ...

spg1 : stmfd sp!, {...}
pc → ...
      ...
      bl spg2
ret2 : ...
      ...
      ldmfd sp!, {...}
      mov pc, lr

spg2 : stmfd sp!, {...}
      ...
      ldmfd sp!, {...}
      mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
lr →  ret1 : ...

spg1 : stmfd sp !, {...}
...
pc →      bl spg2
ret2 : ...
...
ldmfd sp !, {...}
mov pc, lr

spg2 : stmfd sp !, {...}
...
ldmfd sp !, {...}
mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
ret1 : ...

spg1 : stmfd sp !, {...}
...
      bl spg2
```

lr → ret2 : ...
ldmfd sp !, {...}
mov pc, lr

pc → spg2 : stmfd sp !, {...}
...
ldmfd sp !, {...}
mov pc, lr

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
ret1 : ...

spg1 : stmfd sp !, {...}
...
      bl spg2
```

lr → ret2 :

```
...
      ldmfd sp !, {...}
      mov pc, lr
```

```
spg2 : stmfd sp !, {...}
```

pc →

```
...
      ldmfd sp !, {...}
      mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
ret1 : ...

spg1 : stmfd sp !, {...}
...
      bl spg2
```

lr → ret2 : ...
ldmfd sp !, {...}
mov pc, lr

```
spg2 : stmfd sp !, {...}
```

pc → ...
ldmfd sp !, {...}
mov pc, lr

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
ret1 : ...

spg1 : stmfd sp !, {...}
...
      bl spg2
```

lr → ret2 :

```
...
      ldmfd sp !, {...}
      mov pc, lr
```

```
spg2 : stmfd sp !, {...}
```

```
...
      ldmfd sp !, {...}
      mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main :    ...
          bl spg1
ret1 :    ...

spg1 :    stmfd sp!, {...}
...
          bl spg2
pc, lr → ret2 :
...
          ldmfd sp!, {...}
          mov pc, lr

spg2 :    stmfd sp!, {...}
...
          ldmfd sp!, {...}
          mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
ret1 : ...
```



```
spg1 : stmfd sp !, {...}
      ...
      bl spg2
```

lr → ret2 :

pc → ...
 ldmfd sp !, {...}
 mov pc, lr

```
spg2 : stmfd sp !, {...}
      ...
      ldmfd sp !, {...}
      mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
ret1 : ...
```

```
spg1 : stmfd sp !, {...}
      ...
      bl spg2
```

lr → ret2 :

pc →

```
ldmfd sp !, {...}
      mov pc, lr
```

```
spg2 : stmfd sp !, {...}
      ...
      ldmfd sp !, {...}
      mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
ret1 : ...

spg1 : stmfd sp !, {...}
...
      bl spg2
```

lr → ret2 :

pc → ...
 ldmfd sp !, {...}
 mov pc, lr

```
spg2 : stmfd sp !, {...}
...
      ldmfd sp !, {...}
      mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
ret1 : ...  
  
spg1 : stmfd sp!, {...}  
      ...  
      bl spg2  
pc, lr → ret2 : ...  
      ...  
      ldmfd sp!, {...}  
      mov pc, lr  
  
spg2 : stmfd sp!, {...}  
      ...  
      ldmfd sp!, {...}  
      mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
pc →           bl spg1
ret1 : ...
```

```
spg1 : stmfd sp!, {..., lr}
...
bl spg2
ret2 : ...
...
ldmfd sp!, {..., lr}
mov pc, lr
```

```
spg2 : stmfd sp!, {...}
...
ldmfd sp!, {...}
mov pc, lr
```

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...
      bl spg1
lr →  ret1 : ...

pc →  spg1 : stmfdf sp!, {..., lr}
          ...
          bl spg2
ret2 : ...
          ...
          ldmfd sp!, {..., lr}
          mov pc, lr

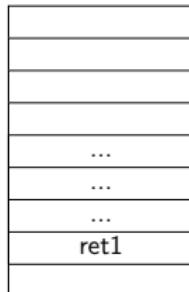
spg2 : stmfdf sp!, {...}
          ...
          ldmfd sp!, {...}
          mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
lr → ret1 : ...  
  
pc → spg1 : stmfdf sp!, {..., lr}  
          ...  
          bl spg2  
ret2 : ...  
          ldmfd sp!, {..., lr}  
          mov pc, lr  
  
spg2 : stmfdf sp!, {...}  
          ...  
          ldmfd sp!, {...}  
          mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
lr → ret1 : ...
```

```
spg1 : stmfdf sp!, {..., lr}  
      ...  
pc →     bl spg2  
ret2 : ...  
      ldmfd sp!, {..., lr}  
      mov pc, lr
```



```
spg2 : stmfdf sp!, {...}  
      ...  
      ldmfd sp!, {...}  
      mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
ret1 : ...
```

spg1 : stmfdf sp!, {..., lr}

...

bl spg2

lr → ret2 :

...

ldmfdf sp!, {..., lr}

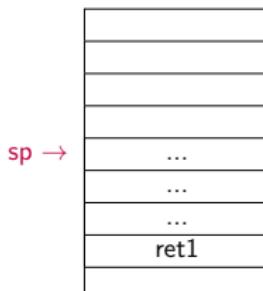
mov pc, lr

pc → spg2 : stmfdf sp!, {...}

...

ldmfdf sp!, {...}

mov pc, lr



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
ret1 : ...
```

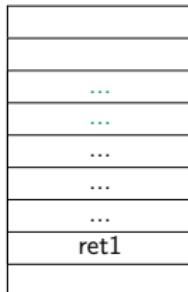
lr → spg1 : stmfd sp!, {..., lr}

...
bl spg2

ret2 : ...

ldmfld sp!, {..., lr}
mov pc, lr

sp →



spg2 : stmfd sp!, {...}

pc → ...

ldmfld sp!, {...}
mov pc, lr

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
ret1 : ...
```

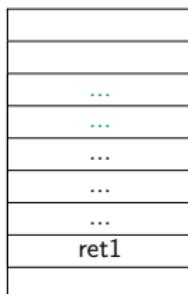
lr → spg1 : stmfd sp!, {..., lr}

sp →

...
bl spg2

ret2 : ...

ldmfld sp!, {..., lr}
mov pc, lr



spg2 : stmfd sp!, {...}

...

pc → ldmfd sp!, {...}
mov pc, lr

Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
ret1 : ...
```

spg1 : stmfdf sp!, {..., lr}

...

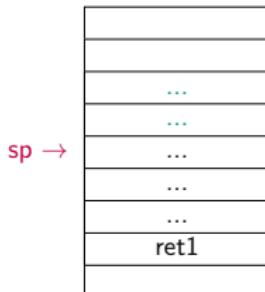
bl spg2

lr → ret2 :

...

ldmfdf sp!, {..., lr}

mov pc, lr



spg2 : stmfdf sp!, {...}

...

ldmfdf sp!, {...}

mov pc, lr

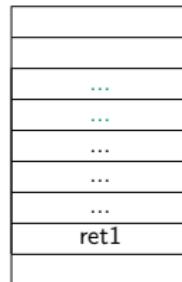
Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main :    ...
          bl spg1
ret1 :    ...

spg1 :  stmfld sp !, {..., lr}
...
bl spg2
pc, lr → ret2 : ...
...
ldmfld sp !, {..., lr}
mov pc, lr

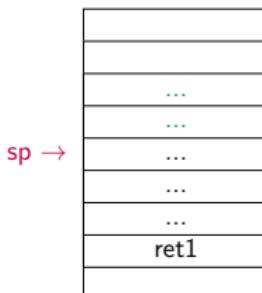
spg2 :  stmfld sp !, {...}
...
ldmfld sp !, {...}
mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

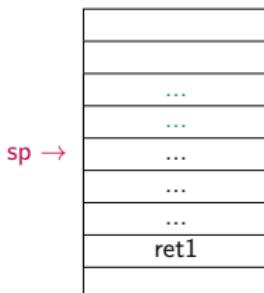
```
main : ...  
      bl spg1  
ret1 : ...  
  
spg1 : stmfdf sp!, {..., lr}  
      ...  
      bl spg2  
lr → ret2 : ...  
pc →      ldmfd sp!, {..., lr}  
      mov pc, lr  
  
spg2 : stmfdf sp!, {...}  
      ...  
      ldmfd sp!, {...}  
      mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

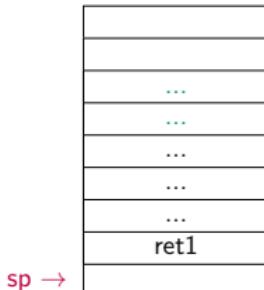
```
main : ...  
      bl spg1  
ret1 : ...  
  
spg1 : stmfdf sp!, {..., lr}  
      ...  
      bl spg2  
lr → ret2 : ...  
pc →     ldmfd sp!, {..., lr}  
          mov pc, lr  
  
spg2 : stmfdf sp!, {...}  
      ...  
      ldmfd sp!, {...}  
      mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
lr →  ret1 : ...  
  
spg1 : stmfdf sp!, {..., lr}  
      ...  
      bl spg2  
ret2 : ...  
      ldmfd sp!, {..., lr}  
pc →      mov pc, lr  
  
spg2 : stmfdf sp!, {...}  
      ...  
      ldmfd sp!, {...}  
      mov pc, lr
```



Sous-programmes imbriqués

Un programme appelle un sous-programme spg1, qui lui-même appelle un sous-programme spg2.

```
main : ...  
      bl spg1  
pc, lr → ret1 : ...
```

```
spg1 : stmfdf sp !, {..., lr}
```

```
...
```

```
bl spg2
```

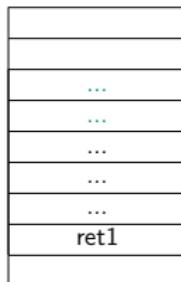
```
ret2 : ...
```

```
...
```

```
ldmfd sp !, {..., lr}
```

```
mov pc, lr
```

sp →



```
spg2 : stmfdf sp !, {...}
```

```
...
```

```
ldmfd sp !, {...}
```

```
mov pc, lr
```

Exercice 19 : Sous-programme itoa()

- ① Ecrire un sous-programme div qui reçoit deux valeurs en entrée, dans `r2` et `r3`, et calcule le résultatat de la division entière de `r2` par `r3`. Le quotient sera rangé dans `r0` et le reste dans `r1`.
- ② Reprendre le sous-programme itoa précédent pour qu'il utilise le sous-programme div.

Exercice 20 : Sous-programme récursif

On considère le programme suivant :

```
_start:    mov r0, #5
            bl fnct
ret1:
_exit:    nop
fnct:      stmfd sp!, {r0, lr}
            cmp r0, #1
            beq finfnct
            sub r0, r0, #1
            bl fnct
ret2:      mul r0, r1, r0
            ldmfd sp!, {r1, pc}
finfnct:
```

Faire la trace de l'exécution de ce programme et déterminer ce qu'il calcule.

Entrées/Sorties

- Le processeur est connecté à un ensemble de **périphériques** par le biais d'un système d'entrées/sorties.

Entrées/Sorties

- Le processeur est connecté à un ensemble de **périphériques** par le biais d'un système d'entrées/sorties.
- Le processeur est relié à une carte (carte mère, système-sur-puce) qui elle même contient un ou plusieurs **bus** sur lequel **la mémoire** (RAM, Flash) et les périphériques sont branchés.

Entrées/Sorties

- Le processeur est connecté à un ensemble de **périphériques** par le biais d'un système d'entrées/sorties.
- Le processeur est relié à une carte (carte mère, système-sur-puce) qui elle même contient un ou plusieurs **bus** sur lequel **la mémoire** (RAM, Flash) et les périphériques sont branchés.
- Parmi les périphériques les plus connus se trouvent **les interfaces homme-machine** (clavier, souris, écran, carte son), **les accélérateurs** (cartes graphiques – GPUs), **les interfaces réseau** (port Ethernet, carte wifi)

Entrées/Sorties

- Le processeur est connecté à un ensemble de **périphériques** par le biais d'un système d'entrées/sorties.
- Le processeur est relié à une carte (carte mère, système-sur-puce) qui elle même contient un ou plusieurs **bus** sur lequel **la mémoire** (RAM, Flash) et les périphériques sont branchés.
- Parmi les périphériques les plus connus se trouvent **les interfaces homme-machine** (clavier, souris, écran, carte son), **les accélérateurs** (cartes graphiques – GPUs), **les interfaces réseau** (port Ethernet, carte wifi)
- Pour les systèmes embarqués, d'autres I/O peuvent être connectées (bus CAN, i2C, etc., GPIO, écrans LCD, LEDs, boutons poussoirs, interrupteurs, interface moteur, etc.)

Entrées/Sorties

- Chaque périphérique branché sur le bus peut être accédé via une plage d'adresses qui lui est dédiée.

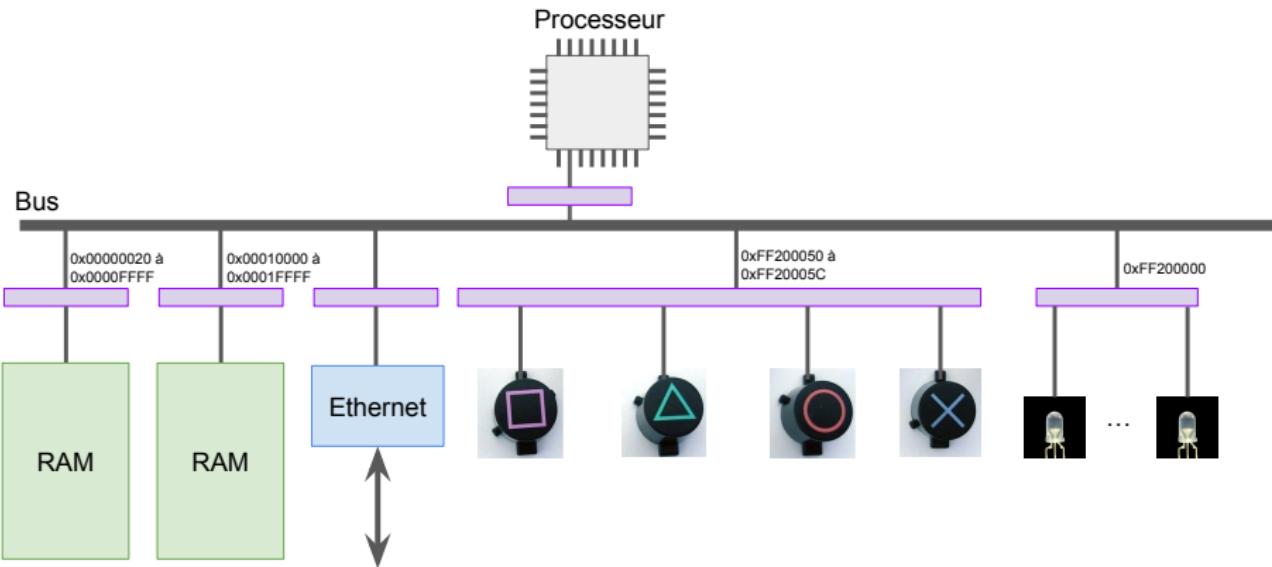
Entrées/Sorties

- Chaque périphérique branché sur le bus peut être accédé via une plage d'adresses qui lui est dédiée.
- En dehors de la mémoire, ces adresses correspondent à ce que l'on appelle les **registres d'entrée/sortie**, c'est à dire l'équivalent des registres du processeur pour le périphérique en question.

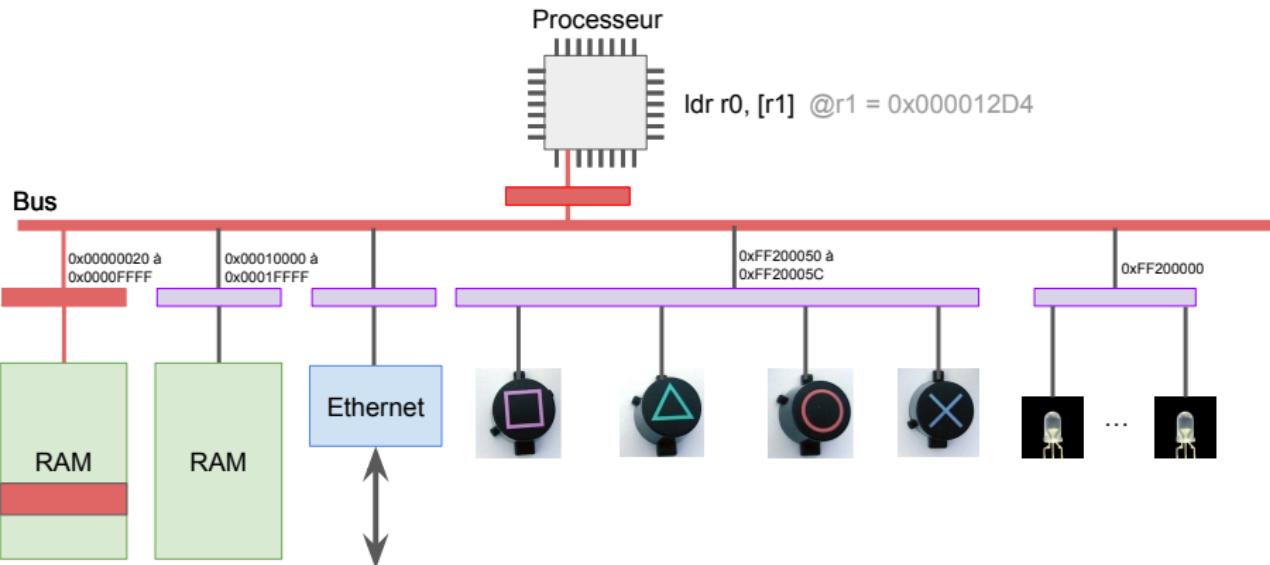
Entrées/Sorties

- Chaque périphérique branché sur le bus peut être accédé via une plage d'adresses qui lui est dédiée.
- En dehors de la mémoire, ces adresses correspondent à ce que l'on appelle les **registres d'entrée/sortie**, c'est à dire l'équivalent des registres du processeur pour le périphérique en question.
- Le processeur peut lire ou écrire dans les registres d'entrée/sortie pour **contrôler** les périphériques, en utilisant les instructions **ldr** et **str**.

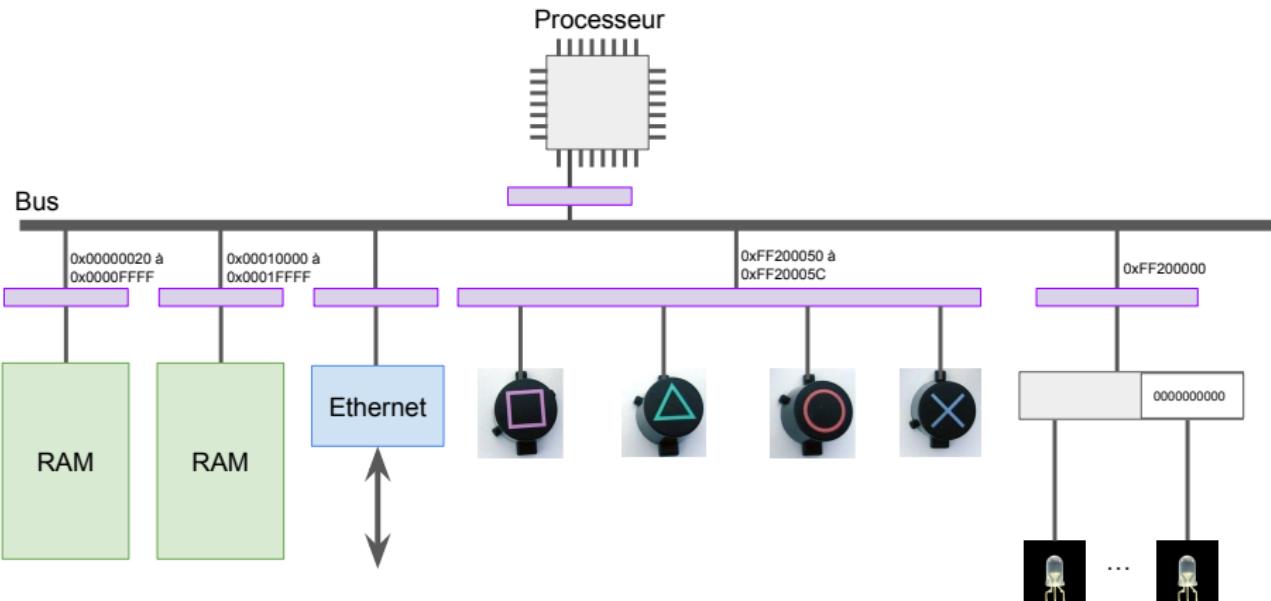
Entrées/Sorties



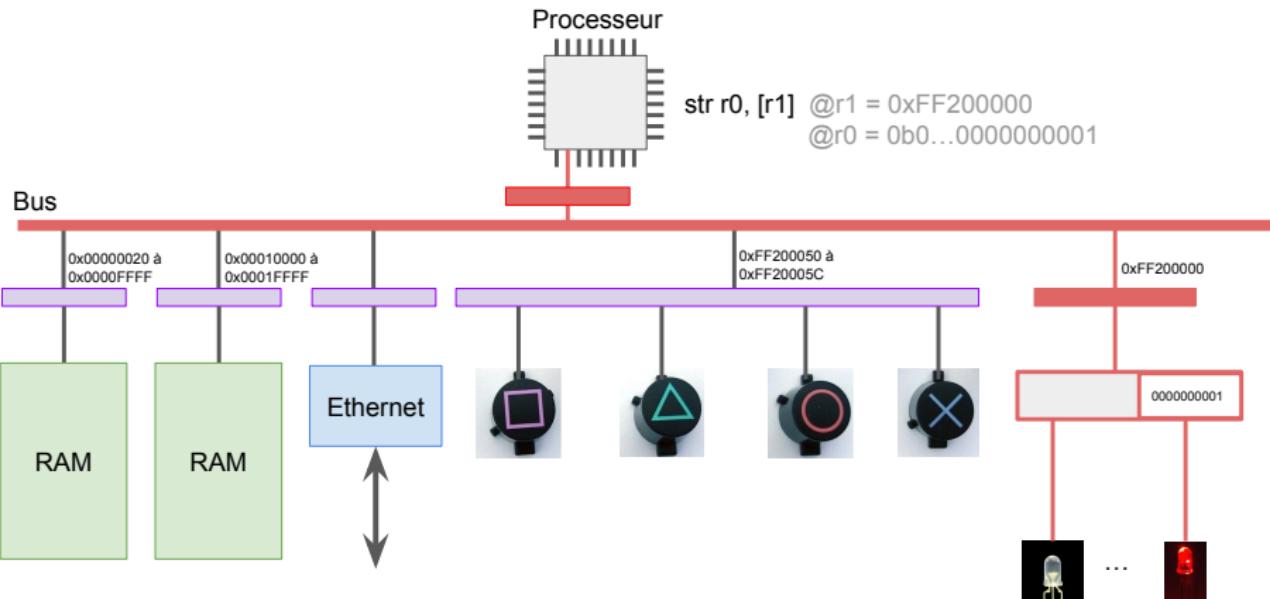
Entrées/Sorties



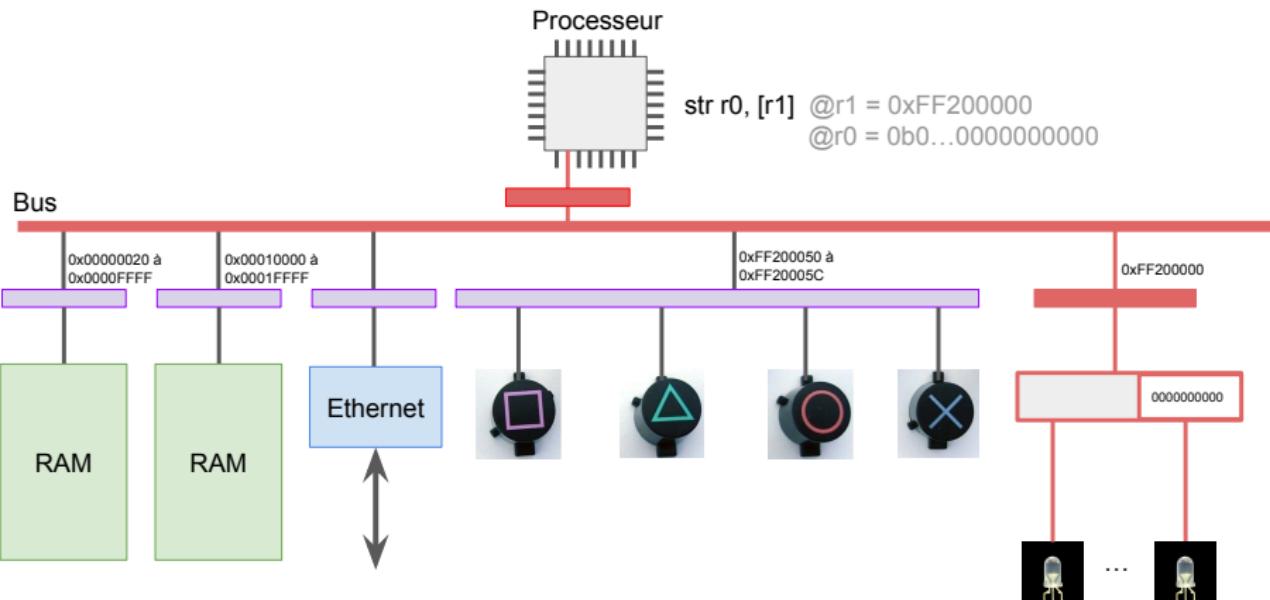
Entrées/Sorties



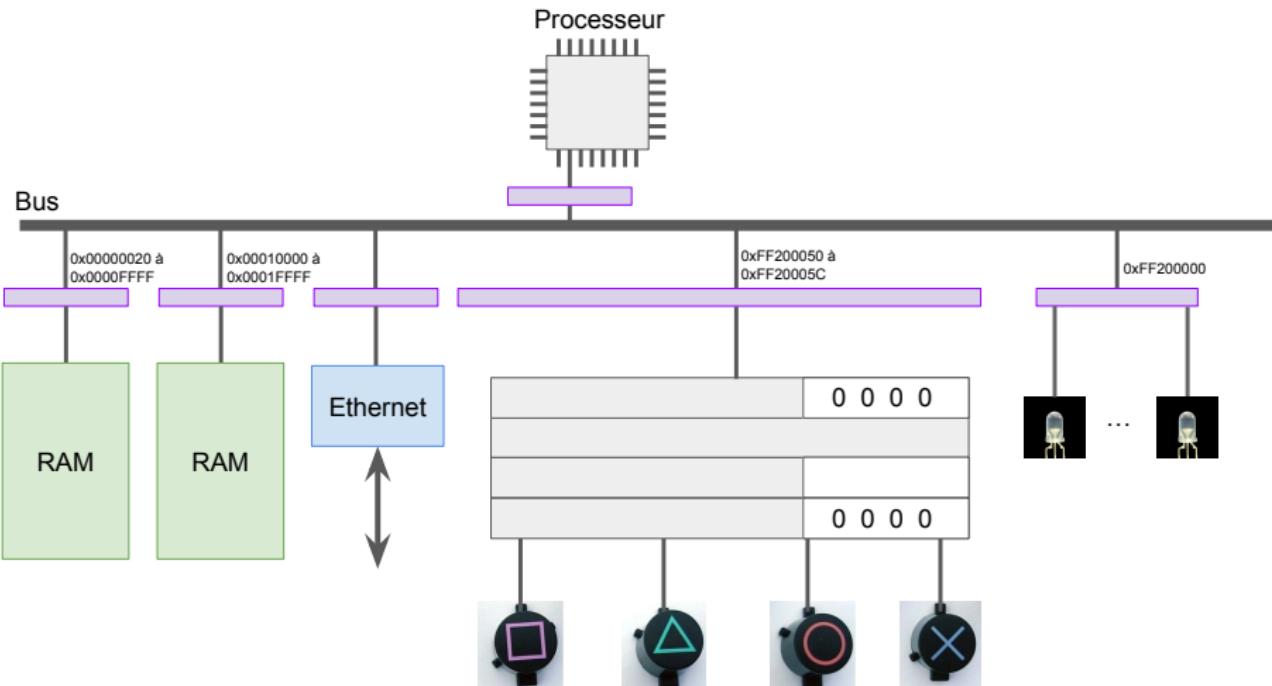
Entrées/Sorties



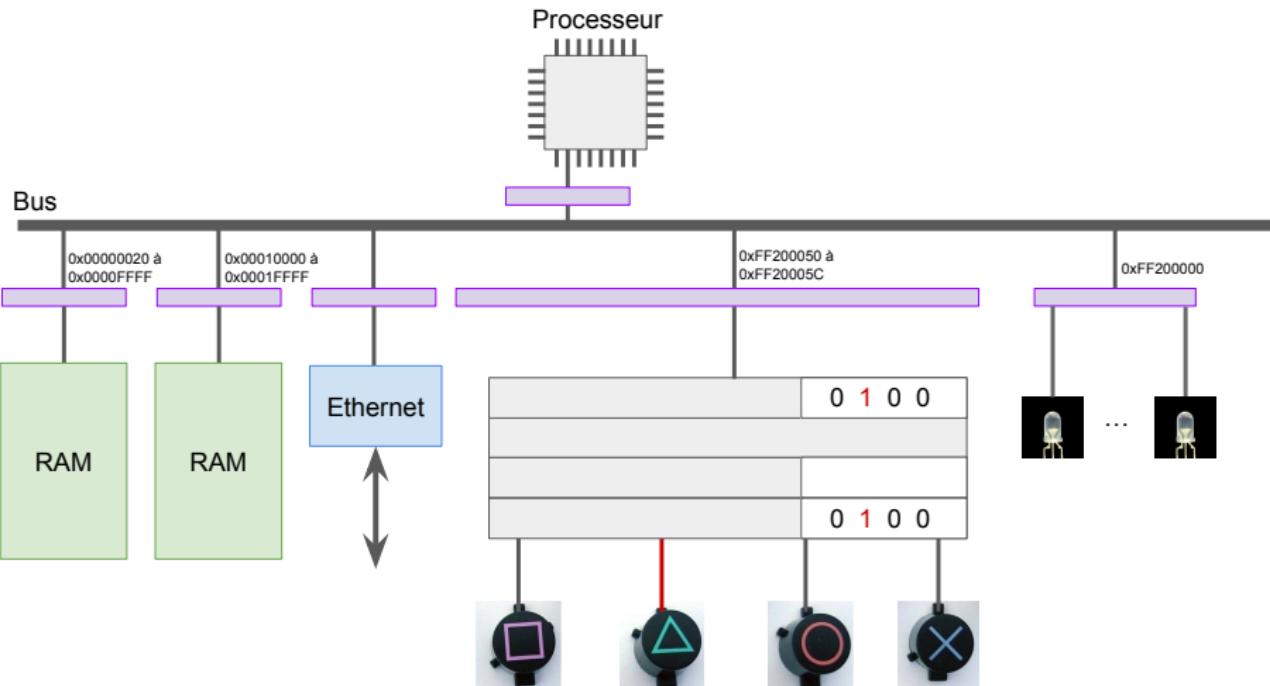
Entrées/Sorties



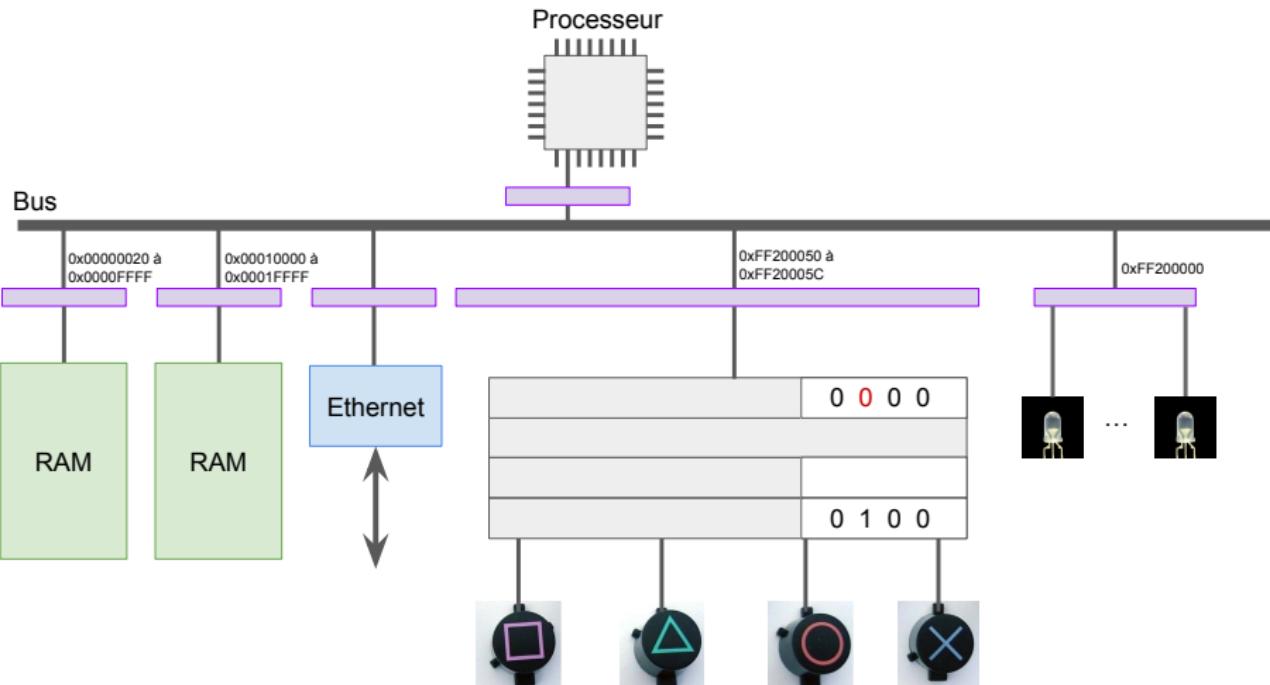
Entrées/Sorties



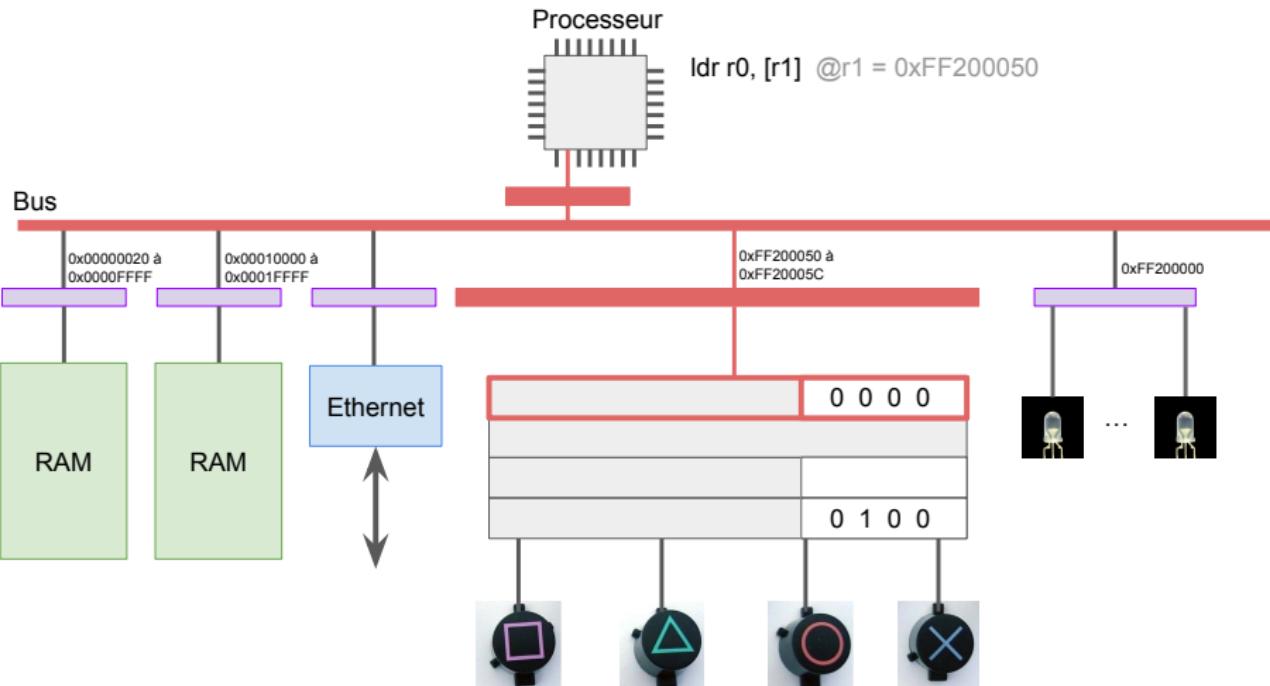
Entrées/Sorties



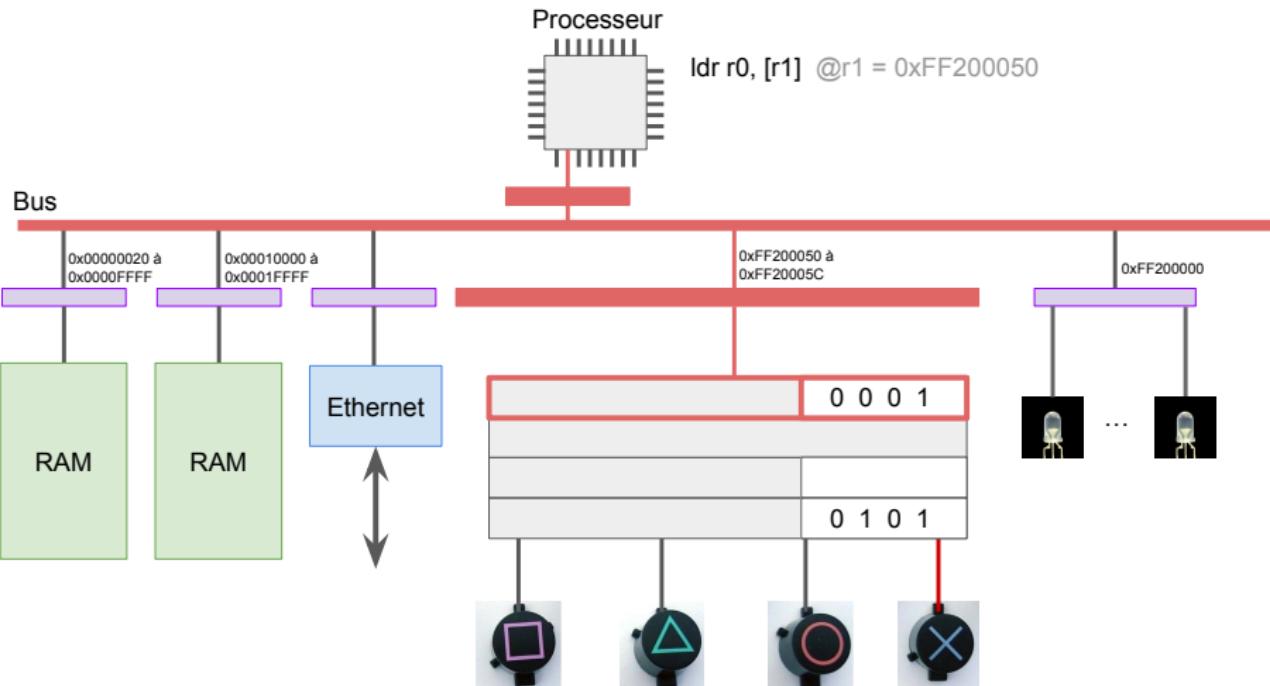
Entrées/Sorties



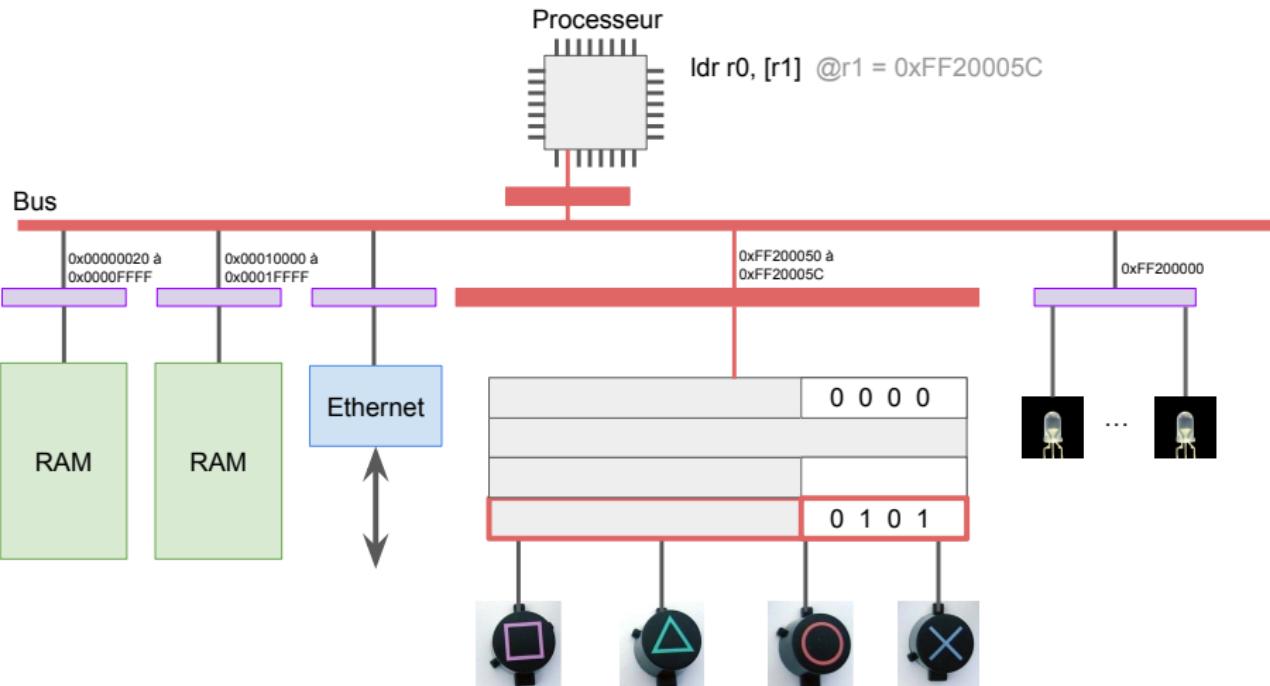
Entrées/Sorties



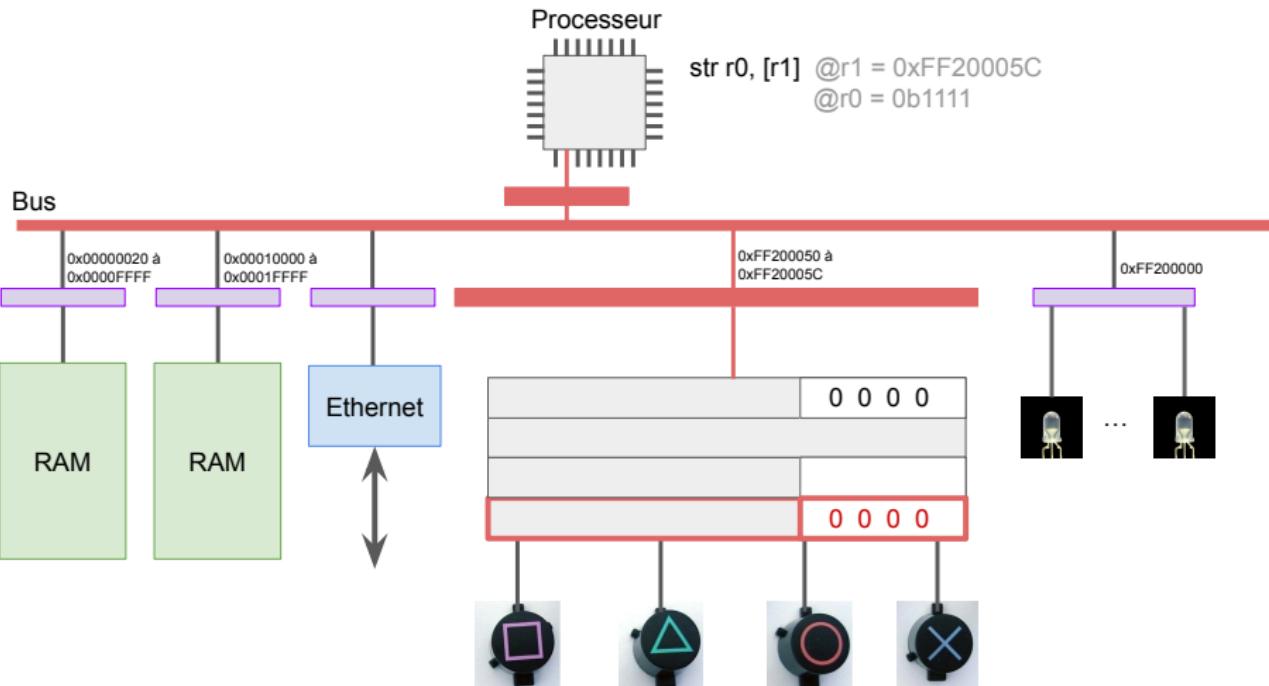
Entrées/Sorties



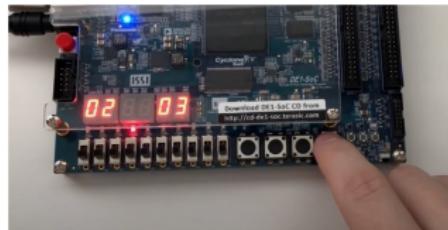
Entrées/Sorties



Entrées/Sorties



Le simulateur



Stopped Step Into F2 Step Over Ctrl-F2 Step Out Shift-F2 Continue F3 Stop F4 Restart Ctrl-Shift-L Reload Ctrl-Shift-L File Help

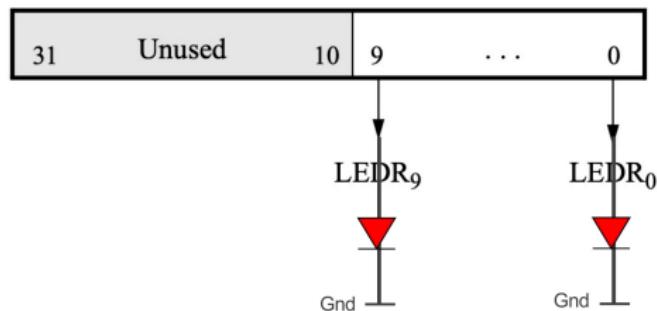
Registers			</> Disassembly (Ctrl-D)			Devices		
Refresh			Go to address, label, or register: 00000020			Refresh		
r0	00000053	Address	Opcode	Disassembly	LEDs ff200000			
r1	0000000f	00000010c	e3a01001	mov r1, #1 ; 0x1	Switches ff200040			
r2	0000000f	000000cc	e1a01501	lsl r1, r1, #10	Push buttons IRQ 73 ff200050			
r3	00000114	000000d0	e590100c	str r1, [r0, #3088]	Seven-segment displays ff200020			
r4	ffffc100	000000d4	e5801c10	pop {r0, r1, pc}	JTAG UART IRQ 80 ff201000			
r5	ff200000	000000d8	e8bd8003					
r6	00000000							
r7	00000000							
r8	00000000							
r9	00000000							
r10	00000000							
r11	00000000							
r12	00000000							
sp	3fffffc							
lr	0000003c							
dc	00000050							

buttons_ISR:

```
push {r0, r1, r2, r3, r4, r5}
ldr r0, [pc, #68] ; 0x12c
ldr r1, [r0, #12]
mov r2, #15 ; 0xf
str r2, [r0, #12]
tst r1, #1 ; 0x1
adr r3, 0x11c (0x11c: leds)
ldr r4, [r3]
```

Contrôle des LEDs

xx xx 0000011001



On écrit dans ce registre pour éteindre/allumer des LEDs :
bit i à 1 : LED i allumée
bit i à 0 : LED i éteinte

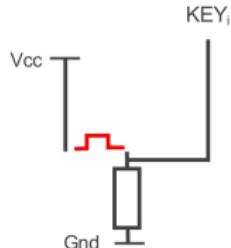
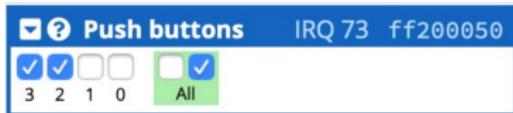


Lecture des boutons

31	30	...	4	3	2	1	0
Unused							KEY ₃₋₀
Unused							
Unused							Mask bits
Unused							Edge bits

Data register = état courant des boutons

On lit ce registre pour connaître l'état courant des boutons :
bit i à 1 : bouton i enfoncé
bit i à 0 : bouton i relâché



Lecture des boutons

31	30	...	4	3	2	1	0
Unused		KEY ₃₋₀					
Unused							
Unused		Mask bits					
Unused		Edge bits					

Edgecapture register = fronts montants observés

On lit ce registre pour savoir sur quels signaux il y a eu un front montant depuis la dernière remise à 0

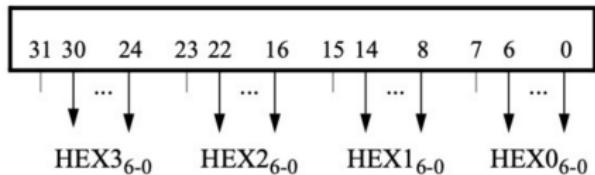
bit i à 1 : il y a eu un front montant sur le signal i (le bouton i a été enfoncé)

bit i à 0 : il n'y a pas eu de front montant sur le signal i

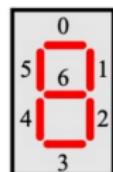
On écrit 0b1111 dans ce registre pour le remettre à 0.

Contrôle des afficheurs 7 segments

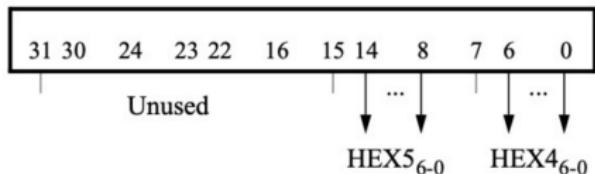
00111000 00000100 01010100 01110001



Data register



Segments



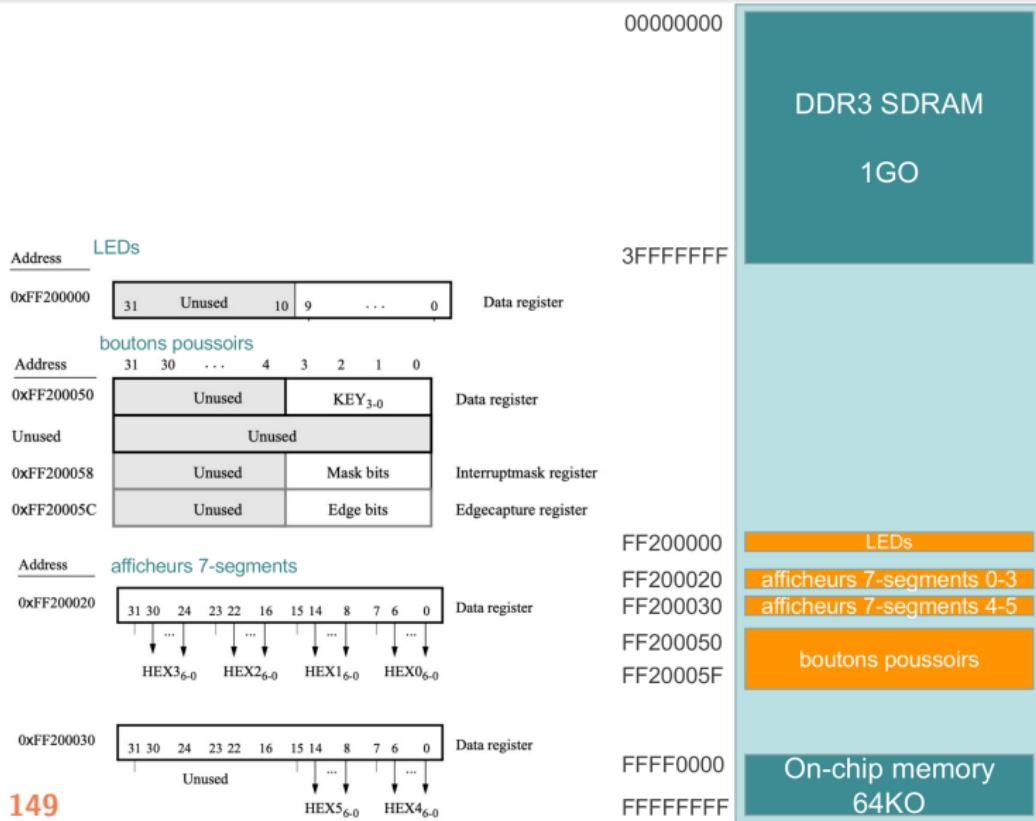
Data register

Chaque afficheur est commandé par 7 bits dans un des deux registres :

le bit i à 1 allume le segment i
le bit i à 0 éteint le segment i



Carte des adresses du simulateur



Charger des grandes valeurs

- On a besoin de charger les adresses des registres d'entrée/sortie directement dans les registres du processeur
 - On ne peut pas utiliser **adr** car il n'y a pas d'étiquette désignant ces adresses
 - On ne peut pas utiliser **mov** car les adresses à charger sont généralement trop grosses pour être encodées sur un immédiat
- On va utiliser la pseudo instruction :
`ldr rd, =adresse`
- Exemple : `ldr r0, =0xFF200000`
met dans `r0` l'adresse du registre contrôlant les LEDs

Charger des grandes valeurs

- On a besoin de charger les adresses des registres d'entrée/sortie directement dans les registres du processeur
 - On ne peut pas utiliser **adr** car il n'y a pas d'étiquette désignant ces adresses
 - On ne peut pas utiliser **mov** car les adresses à charger sont généralement trop grosses pour être encodées sur un immédiat
- On va utiliser la pseudo instruction :
`ldr rd, =adresse`
- Exemple : `ldr r0, =0xFF200000`
met dans **r0** l'adresse du registre contrôlant les LEDs
- De même, quand on veut spécifier le contenu à écrire dans un registre, on utilise cette pseudo instruction
- Exemple : `ldr r1, =0b1111111111`
met dans **r1** la valeur 0b1111111111 qui n'est pas encodable par un immédiat

Exercice 21 : Allumer des LEDs

- Écrire un programme qui allume une LED sur 2 dans le simulateur (les leds 9, 7, 5, 3, 1)

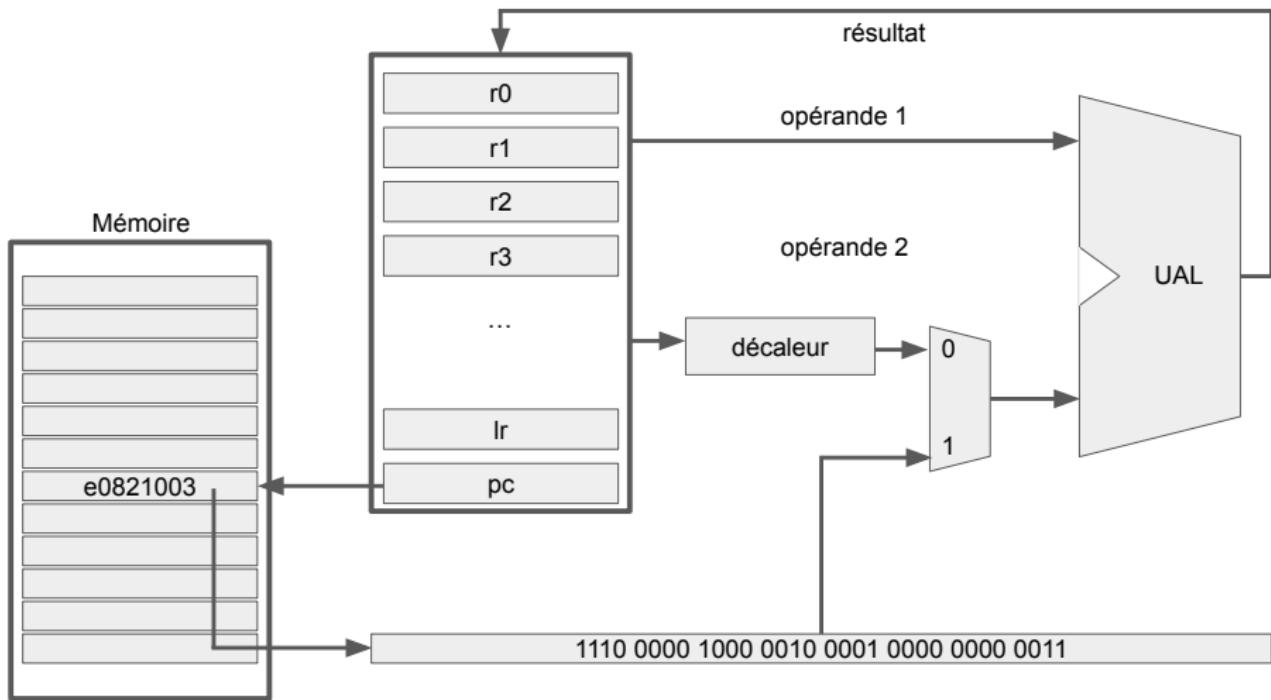
Exercice 22 : Faire clignoter une LED

- Ecrire un programme qui fait clignoter une LED
 - Pour cela on fait une boucle infinie dans laquelle on allume la LED, puis on attend un moment, puis on éteint la LED, et on attend à nouveau
 - L'attente se fait en initialisant un registre à la valeur 0, puis en faisant une boucle dans laquelle on incrémente ce registre, et dont on sort quand il atteint une valeur suffisamment grande

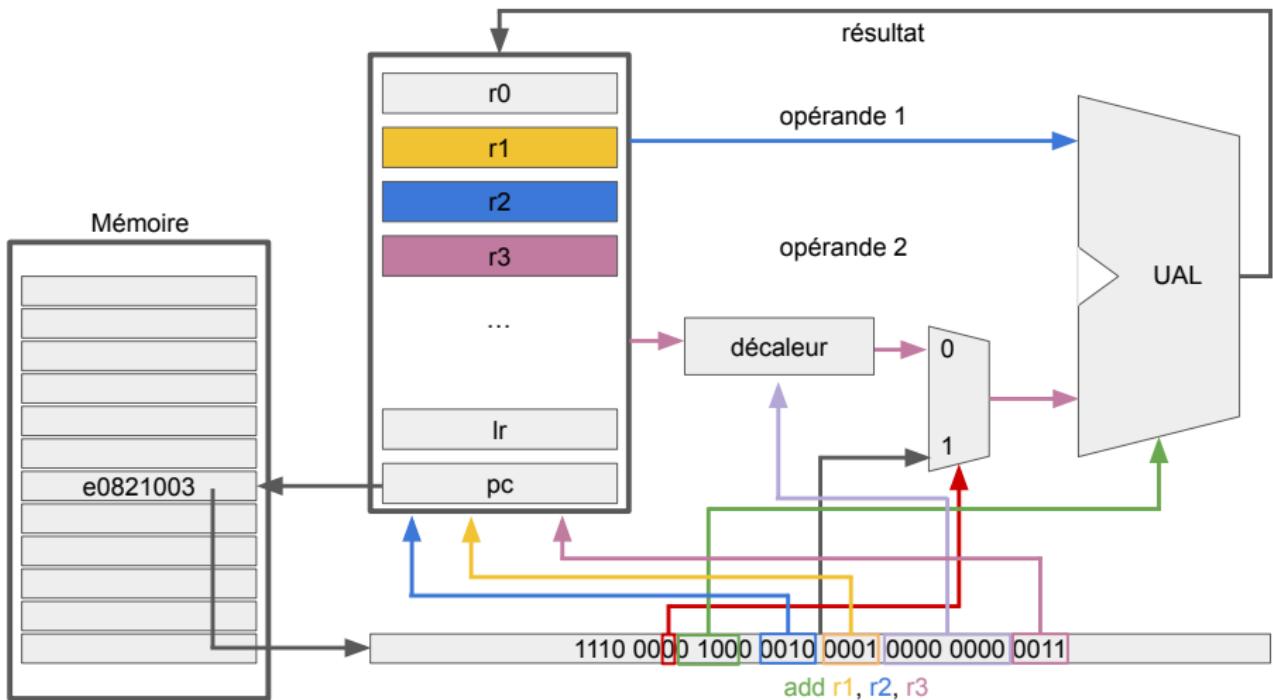
Exercice 23 : Afficher 32 sur les afficheurs 7 segments

- Écrire un programme permettant d'afficher 32 à l'aide des afficheurs 7 segments

Exemple d'exécution d'une instruction

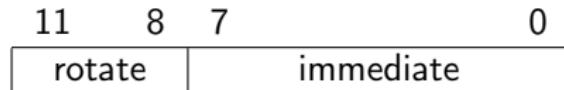


Exemple d'exécution d'une instruction



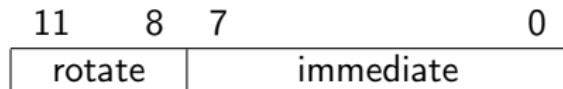
Codage des valeurs immédiates

- Les valeurs immédiates sont codées sur **12 bits** dans les instructions ARM
- Le codage s'appuie sur 2 champs :
 - Un champ de rotation sur 4 bits appelé **rotate**
 - Un champ de 8 bits appelé **immediate**



Codage des valeurs immédiates

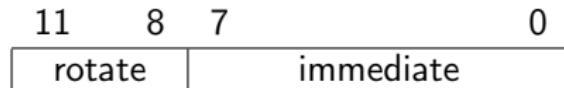
- Les valeurs immédiates sont codées sur **12 bits** dans les instructions ARM
- Le codage s'appuie sur 2 champs :
 - Un champ de rotation sur 4 bits appelé **rotate**
 - Un champ de 8 bits appelé **immediate**



- A partir de ces 2 champs, le processeur calcule la valeur comme suit :
$$\text{constante} = (\text{immediate}) \ggg (2 * \text{rotate})$$
- Ex : 64 se code 000001000000

Codage des valeurs immédiates

- Les valeurs immédiates sont codées sur **12 bits** dans les instructions ARM
- Le codage s'appuie sur 2 champs :
 - Un champ de rotation sur 4 bits appelé **rotate**
 - Un champ de 8 bits appelé **immediate**



- A partir de ces 2 champs, le processeur calcule la valeur comme suit :
$$\text{constante} = (\text{immediate}) >>> (2 * \text{rotate})$$
- Ex : 2048 se code 101100000010

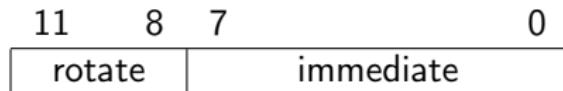
00000000 00000000 00000000 0000010

Après 22 rotations sur la droite :

00000000 00000000 00001000 00000000

Codage des valeurs immédiates

- Les valeurs immédiates sont codées sur **12 bits** dans les instructions ARM
- Le codage s'appuie sur 2 champs :
 - Un champ de rotation sur 4 bits appelé **rotate**
 - Un champ de 8 bits appelé **immediate**



- A partir de ces 2 champs, le processeur calcule la valeur comme suit :
$$\text{constante} = (\text{immediate}) >>> (2 * \text{rotate})$$
- Ex : 2048 se code 101100000010
- Lorsque plusieurs combinaisons sont possibles pour coder une valeur, le compilateur choisit celle qui produit le plus petit champ **rotate**

Exercice 24 : Codage des immédiats

- La valeur 0x30D00
 - Est-elle codable sous la forme d'un immédiat ?
 - Si oui, comment ?
- La valeur 0x3D00
 - Est-elle codable sous la forme d'un immédiat ?
 - Si oui, comment ?

Exercice 24 : Codage des immédiats

- La valeur 0x30D00
 - Est-elle codable sous la forme d'un immédiat ?
 - Non : le champ immédiate est trop petit
 - Si oui, comment ?
- La valeur 0x3D00
 - Est-elle codable sous la forme d'un immédiat ?
 - Si oui, comment ?

Exercice 24 : Codage des immédiats

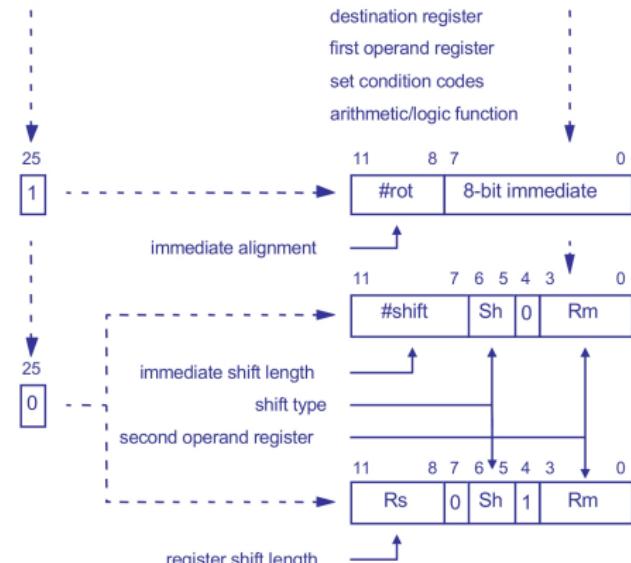
- La valeur 0x30D00
 - Est-elle codable sous la forme d'un immédiat ?
 - Non : le champ immédiate est trop petit
 - Si oui, comment ?
- La valeur 0x3D00
 - Est-elle codable sous la forme d'un immédiat ?
 - Oui (3D rentre sur un octet)
 - Si oui, comment ?
 - Immediate = 0x3D (0b00111101)
 - Rotate = 0b1100 (décalage de 8 bits vers la gauche = 32-8 vers la droite, soit 24, soit encore 2*12)

Traitement des données

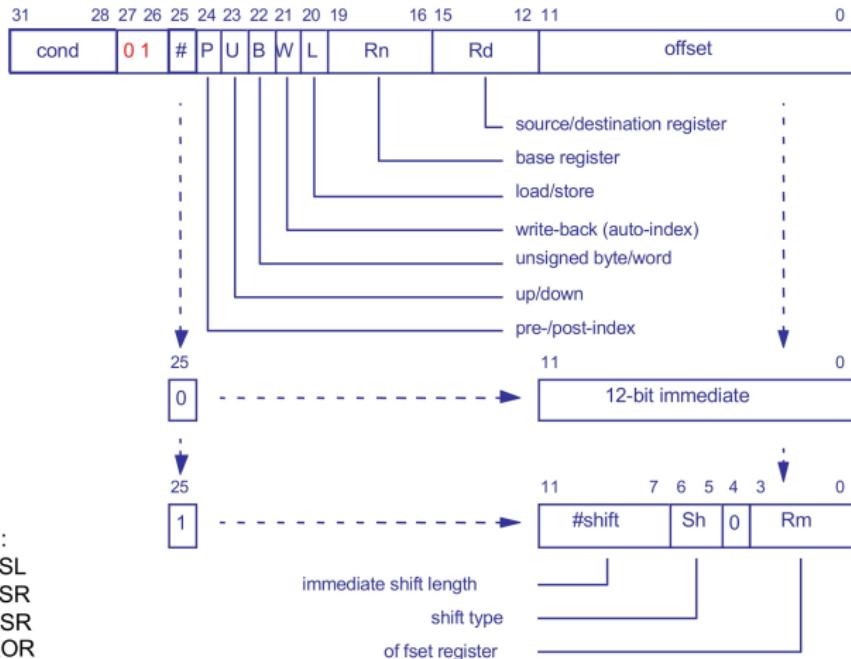


Opcode

0000	AND	$Rd := Rn \text{ AND } Op2$
0001	EOR	$Rd := Rn \text{ EOR } Op2$
0010	SUB	$Rd := Rn - Op2$
0011	RSB	$Rd := Op2 - Rn$
0100	ADD	$Rd := Rn + Op2$
0101	ADC	$Rd := Rn + Op2 + C$
0110	SBC	$Rd := Rn - Op2 + C - 1$
0111	RSC	$Rd := Op2 - Rn + C - 1$
1000	TST	$Scc \text{ on } Rn \text{ AND } Op2$
1001	TEQ	$Scc \text{ on } Rn \text{ EOR } Op2$
1010	CMP	$Scc \text{ on } Rn - Op2$
1011	CMN	$Scc \text{ on } Rn + Op2$
1100	ORR	$Rd := Rn \text{ OR } Op2$
1101	MOV	$Rd := Op2$
1110	BIC	$Rd := Rn \text{ AND NOT } Op2$
1111	MVN	$Rd := \text{NOT } Op2$



Accès mémoire



Shift type :

- 00 : LSL
- 01 : LSR
- 10 : ASR
- 11 : ROR

Branchements



Condition code

0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Décoder une instruction

- ① On commence par regarder les bits 27 et 26 pour déterminer la **nature** de l'instruction :
 - **00** : traitement de données
 - **01** : accès mémoire
 - **10** suivi d'un 1 au bit 25 : branchement

Décoder une instruction

- ① On commence par regarder les bits 27 et 26 pour déterminer la **nature** de l'instruction :
 - **00** : traitement de données
 - **01** : accès mémoire
 - **10** suivi d'un 1 au bit 25 : branchement
- ② Selon la nature de l'instruction on se réfère au schéma d'encodage

Décoder une instruction

- ① On commence par regarder les bits 27 et 26 pour déterminer la **nature** de l'instruction :
 - **00** : traitement de données
 - **01** : accès mémoire
 - **10** suivi d'un 1 au bit 25 : branchemetn
- ② Selon la nature de l'instruction on se réfère au schéma d'encodage
- ③ Selon la valeur du bit 25, le décodage des bits 0 à 11 est différent :
 - pour une instruction de **traitement des données**, bit 25 = 0 => opérande2 = registre et bit 25 = 1 => opérande2 = immédiat
 - pour une instruction d'**accès mémoire**, c'est l'inverse

Décoder une instruction

- ① On commence par regarder les bits 27 et 26 pour déterminer la **nature** de l'instruction :
 - **00** : traitement de données
 - **01** : accès mémoire
 - **10** suivi d'un 1 au bit 25 : branchement
- ② Selon la nature de l'instruction on se réfère au schéma d'encodage
- ③ Selon la valeur du bit 25, le décodage des bits 0 à 11 est différent :
 - pour une instruction de **traitement des données**, bit 25 = 0 => opérande2 = registre et bit 25 = 1 => opérande2 = immédiat
 - pour une instruction d'**accès mémoire**, c'est l'inverse
- ④ Lorsque le second opérande est un registre, on doit calculer son décalage (qui peut valoir 0). Le bit 4 permet de spécifier si le décalage est spécifié par un immediat court (bit 4 = 0) ou par un autre registre (bit 4 = 1)

Exercice 25 : Traduire en assembleur l'instruction

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	
1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0

Exercice 26 : Écrire le code de l'instruction add r0, r0, r1

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

Exercice 27 : Encoder l'instruction adds r8, r5, #65536

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

Exercice 28 : Encoder l'instruction

ldr r0, [r1], #1

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

Exercice 29 : Encoder les instructions de branchement dans le programme

```
main :  mov r0, #0
          mov r1, #1
tq :   cmp r1, #N
        bhi ftq
        add r0, r0, r1
        add r1, r1, #1
        b tq
ftq :  nop
```

Exercice 30 : Décoder le programme

e28f0020

e28f1040

e3a02000

e1500001

2a000003

e4903004

eafffffd

e1a00000