

# Structures de contrôles en assembleur ARM et premiers exemples

# Hello world

label

str:

```
.data
.asciz    " Hello World\n "

.text
.globl    main

main:     stmfd    sp!,{lr}
          ldr      r0,=str
          bl       printf
          mov      r0,#0
          ldmfd    sp!,{lr}
          mov      pc,lr
```

Section .data

Section .text

Instruction et opérandes

# Hello world

Exécutez

```
> arm-linux-gnueabi-gcc helloworld.s  
> qemu-arm -L /usr/arm-linux-gnueabi a.out  
> Hello World  
>
```

Si ça ne marche pas

```
> sudo apt(-get) install gcc-arm-linux-gnueabi  
> sudo apt(-get) install qemu-user
```

Si vous ne trouvez pas la librairie (printf)

```
> readelf -l a.out
```

Déterminez où doit se trouver la librairie (/lib/ld-linux.so.3 par exemple)

Et cherchez le chemin de la librairie: locate ld-linux.so.3 (par exemple)

Finalement ajustez l'option `-L` de `qemu-arm`

# Syntaxe assembleur GNU

Un programme en assembleur est constitué de **3 colonnes**.

La 1<sup>ère</sup> colonne est utilisé pour les **labels**

La 2<sup>ème</sup> colonne pour les **instructions** (avec les opérandes)

La 3<sup>ème</sup> pour les **commentaires**

A priori une ligne = une instruction assembleur

Les séparateurs sont les espaces ou les caractères de tabulation

Pour les commentaires on peut utiliser /\* ... \*/, //, ou @ (essayez)

# Syntaxe assembleur GNU

Une ligne correspond à un **statement**.

Un **symbole** est composé d'au moins un caractère (maj. ou min.), de chiffre(s), et des trois caractères (.) , (\_) , (\$).

Un statement commence par un ou plusieurs labels suivis par un symbole clé (key symbol) qui détermine le type de statement. La syntaxe qui suit le symbole clé dépend de ce symbole.

Un **label** est un symbole terminé par (:) )

Un symbole (clé) qui commence par un (.) est une **directive d'assemblage**. S'il commence par une lettre c'est une **instruction du langage assembleur** qui sera traduit en une instruction machine (opcode, code machine c'est une suite de bits interprétée par le processeur)

# Syntaxe assembleur GNU

Pour les entiers (integers) on utilise:

- 0b pour les nombres binaires, 0b001101
- 0 pour les nombres en octal 001234567
- 0x pour les nombres en hexadécimal 0x0123456789ABCDEF
- Un nombre décimal commence par un digit différent de 0 et suivi de un ou plus digit(s)  
0123456789
- L'opérateur – désigne un nombre négatif

Les entiers sont stockés dans un *int* du langage C, typiquement sur 32 bits. Si ce n'est pas possible GNU parle de bignums.

Pour les nombres en virgule flottante (flonum)

- 0e(+/-)(integer part)(.)(fractional part)(E/e)(+/-)(exponent)

# Les sections

Les directives assembleur pour définir les sections sont:

- `.text`
- `.data`
- `.bss`

Les sections `.text` et `.data` contiennent le programme. En général `.text` contient ce qui est inaltérable – code et constante – et est partagé par les différents processus s’il y a lieu. La section `.data` contient ce qui change, typiquement les variables.

La section `.bss` est mise à zéro lorsque le programme commence.

Dans le fichier objet, la section `.text` commence à l’adresse 0, suit la section `.data` et `.bss`.

L’intérêt des sections est que le linker les considère comme un bloc. C’est-à-dire qu’il ne peut pas les fractionner. Si un statement dans un bloc est à l’adresse `x` lors de l’exécution du programme le statement suivant se trouve à l’adresse suivante (dépend du nombre de byte du statement à l’adresse `x`).

Les adresses de base des segments sont définies à l’exécution.

Les adresses dans une section sont définies par (base section) + (offset dans la section)

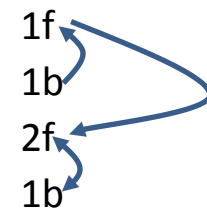
# Labels

Les labels sont utilisés dans le programme comme références à un statement particulier. Par exemple, dans le programme **str** fait référence au statement qui définit une chaîne de caractère.

Certains labels ont une visibilité locale, ceux qui s'écrivent N: où N est un nombre entier. Pour les accéder on utilise Nf pour désigner le prochain dans la section ou Nb celui qui précède dans la section.

## Exemple

1:	branch	1f
2:	branch	1b
1:	branch	2f
2:	branch	1b



Le symbol (.) fait référence à l'adresse du statement courant. Sont équivalents

bcl:	branch	bcl
	branch	.



# Directives d'assemblage

## **.align** expression

demande à l'assembleur d'ajouter des bytes de remplissage (padding) pour assurer que l'adresse suivante soit alignée selon la valeur de l'expression. L'expression représente le nombre de zéros que doit avoir l'adresse (.align 2 désigne une adresse divisible par 4)

## **.ascii** " string ", ...

## **.asciiz** " string ", ...

Idem à .ascii mais les chaînes de caractères se terminent par un caractère 0x0

## **.balign** expression

L'alignement en octets, .balign 4 la prochaine adresse est un multiple de 4

## **.bss**

## **.byte** expression, ...

Les expressions sont converties sur 8 bits.

# Directives d'assemblage

## **.data** subsection

Les données qui suivent doivent être assemblées dans la section data avec le numéro subsection (une expression)

## **.double** flonum, ... (**.float** flonum, ...)

Des nombres flottants.

## **.equ** symbol, expression

## **.global** symbol (**.globl** symbol)

Le symbol est globale c'est-à-dire connu par l'éditeur des liens

## **.hword** expression, ...

Les expressions sont converties sur 16 bits.

## **.int** expressions (idem **.long**)

Génère une valeur à l'exécution.

## **.if** expression ... **.endif** (voir le manuel pour les options, **ifeq**, **ifge**, ...)

Si la valeur de l'expression est différente de 0, le contenu du bloc est inséré dans le code source.

# Directives d'assemblage

**.macro** macroname arg,... **.endm**

Permet de définir une macro macroname(arg,...)- La valeur des paramètres est accessibles par \arg (\@ fait référence au nombre de macro que l'assembleur a exécuté). Les appels récursifs sont autorisés.

**.rept** count .... **endr**

Répète la séquence ... count fois.

.rept 3	.long 0
.long 0	.long 0
.endr	.long 0

**.set** symbol, expression

symbole = expression.

# Directives d'assemblage

**.skip** expression (, fill) (**.space** expression, fill)

L'expression indique le nombre de bytes à réserver (fill indique la valeur d'initialisation)

**.string** " string ", ...

Permet de définir une ou plusieurs chaîne de caractères.

**.text**

**.word** expression, ...

Les expressions sont converties sur 32 bits.

**.2byte** expression, ...

Ecrit sur 2 bytes la valeur de l'expression . On a aussi **4byte**, **8byte**.

# Examples

```
i:      .data
        .word    0           // static int i=0;
fmt:    .asciz    " Hello World\n " // static char fmt[] = " Hello World\n ";
ch:     .byte     'A', 'B', 0    // static char ch[] = {'A', 'B', 0}

        .align    2
ary:    .word     0, 1, 2       // static int ary[] = {0, 1, 2}

        .balign   4
a:      .skip     400          // static int a[100]

        .set      counter,0
        .rept     10
        str       r0,[r1,#counter]
        .set      counter, counter +8
        .endr
```

# Les macros

Les macros permettent de générer du code assembleur. Par exemple:

	macname	macargs
.long 0		.macro sum from=0, to=5
.long 1		.long \from ← référence à l'argument
.long 2		.if \to-\from
.long 3		sum "(\from+1)",\to ← appel récursif
.long 4		.endif
.long 5		.endm

La syntaxe est: *.macro macname macargs...* (les arguments sont optionnels et séparés par une virgule ou un espace). Ici, *sum 0,5* = *sum 0* = *sum*.

# .if ... .endif

**.if** une expression

Si l'expression est non nulle alors le code est inséré dans le programme à assembler. On peut utiliser **.else** pour une alternative et aussi **elseif**.

**.ifdef** symbol test si le symbol est défini

**.ifeq** si l'expression est nulle

**.ifge** si plus grand ou égal à zéro

**.ifgt** si (strictement) plus grand

**.ifle** si plus petit ou égal

**.iflt** si plus petit

# If ... then ... else

```
static int a,b,x;
```

```
    if (a<b)
```

```
        x=1;
```

```
    else
```

```
        x=0;
```

```
ldr    r0,=a
```

```
ldr    r1,=b
```

```
ldr    r0,[r0]
```

```
ldr    r1,[r1]
```

```
cmp    r0,r1
```

```
movlt  r0,#1
```

```
movge  r0,#0
```

```
ldr    r1,=x
```

```
str    r0,[r1]
```

instruction conditionnelle



# avec Branch

```
ldr    r0,=a
ldr    r1,=b
ldr    r0,[r0]
ldr    r1,[r1]
cmp    r0,r1
bge    else
mov    r0,#1
b      after
else:  mov    r0,#0
after: ldr    r1,=x
      str    r0,[r1]
```

Les ruptures de séquences forcent le processeur à vider le pipeline. En général s'il y a au moins trois instructions dans chaque blocs on préfère la version avec branch sinon on utilise les instructions conditionnelles.

# while ... end

```
loop:    cmp     r0,r1
         blt     done    while r0>=r1
         ...
         b       loop
done:    ...
```

# do ... while

```
loop:    ...  
        cmp     r0,r1  
        bge     loop
```

Avec un compteur

```
        ldr     r2, length          macro, ldr r2,[pc, #offset]  
  
loop:    ...  
        subs    r2, r2, #0x1  
        bne     Loop  
        . . . . .
```

```
length:  .word 0x12345678
```

# Appel de procédures

Avant l'appel le registre lr (Link Register) est initialisé avec l'adresse de retour de la routine

<pre>bl    foo ... ... foo:  ... ... mov   pc, lr</pre>	<pre>ldr    r12, =foo mov     lr, pc mov     pc, r12 ... <b>adresse de retour</b> ...  foo:    ... ... mov     pc, lr</pre>	(adr r12,foo)
---	---	---------------

# Appel de procédure imbriqués

Si on appelle une procédure depuis une procédure alors il faut sauvegarder la valeur de **lr**. On utilise en général **la pile**.

```
#include <stdio.h>
static char str1[] = "%d";
static char str2[] = "you entered %d\n";
static int n=0;
int main(){
    scanf(str1,&n);      str1 est un pointeur
    printf(str2,n);
    return 0;
}
```

# en assembleur

```

        .data
1tr1:   .asciz    "%d"
str2:   .asciz    "you entered %d\n"
n:      .word     0
        .text
        .globl   main
main:   stmfd     sp!,{lr}           sauvergarde lr
        ldr       r0,=str1
        ldr       r1,=n
        bl        scanf            ici on modifie lr
        ldr       r0,=str2
        ldr       r1,=n
        ldr       r1,[r1]
        bl        printf           ici on modifie lr
        mov       r0,#0
        ldmfd     sp!,{lr}
        mov       pc,lr
```

# Appel de fonctions avec plus de 4 arguments

Si la fonction appelée à plus de quatre argument il faut utiliser la pile.

```
printf("les resultst sont: %d %d %d %d %d %d",i,j,k,l,m)
```

```
ldr    r3,=m
ldr    r3,[r3]          r0 contient m
ldr    r0,=l
ldr    r0,[r0]
stmfd  sp!, {r0,r3}     r0 va à l'adresse la plus petite equ. str sp!,r3 puis str sp!, r0
ldr    r0,=fmtstring
ldr    r1,=i
ldr    r1,[r1]
...
bl     printf
add    sp,sp,#8         pour la mémoire sur la pile
```

# Variables automatiques

Les variables automatiques sont déclarées dans un bloc et locales à ce bloc. Leur durée de vie correspond au temps d'exécution du bloc. Elles sont stockées:

- Dans un registre (mot clé *register* en C)
- Sur la pile

int doit()	doit:	sub	sp,sp,#80	
{ int x[20];		mov	r2,#0	r2=i
register int i;	loop:	cmp	r2,#20	
for(i=0;i<20;i++) x[i]=i;		bge	done	
return i;		str	r2,[sp, r2,asl#2]	
}		add	r2,r2,#1	
		b	loop	
	done:	mov	r2,r0	return i
		add	sp,sp,#80	libère la pile
		mov	pc,lr	



# Structures - C

```
struct student{
    char    first_name[30];
    char    last_name[30];
    unsigned char  class;
    int     grade;
};

struct student newstudent; // nouvelle variable
strcpy(newstudent.first_name, "Sam");
strcpy(newstudent.last_name, "Smith");
newstudent.class=2;
newstudent.grade=88;
```

# Structures - ARM

```
.data
.equ      s_first_name, 0
.equ      s_last_name, 30
.equ      s_class, 60
.equ      s_grade, 64
.equ      s_size, 68
sam        .asciz
smith      .asciz      "Smith"

sub        sp,sp,#s_size  allocation mémoire sur la pile
mov        r0,sp          r0 pointe sur le début de la structure
add        r0,r0, #s_first_name
ldr        r1,=sam
bl        strcpy
```

} 1<sup>er</sup> appel à strcpy

# Structure - ARM

```
mov    r0,sp
add    r0,r0,s_last_name
ldr    r1,=smith
bl     strcpy
mov    r0,sp
mov    r1,#2
strb   r1,[r0, #s_class]
mov    r1,#88
str    r1,[r0,#s_grade]
```

} 2<sup>ème</sup> appel à strcpy

newstudent.class=2;

newstudent.grade=88;

# Le préprocesseur

Le préprocesseur de l'assembleur as (gnu) prépare le fichier source pour l'assemblage. Les opérations qu'il réalise sont:

- Il élimine les caractères espace (séparateurs) inutile (un seul suffit)
- Elimine les commentaires
- Convertit les caractères constants en des valeurs numériques.

# Gestion de la mémoire

Un **processus** ou un **thread** est déterminé par son **état (program state)**.

L'état d'un processus est défini par les valeurs des registres r0-r15, cpsr et le contenu de la mémoire qu'il peut accéder, état = (r0-r15, cpsr, Mem).

La mémoire peut être classée en:

- Le code, exécutable.
- Les données read-only statiques.
- Les données statiques.
- Le tas (the heap)
- La pile (the stack)

# Gestion de la mémoire

## Code du programme

Les instructions exécutées, en général séquentiellement, par le processeur. Cette zone mémoire peut être read-only.

## Allocation de la mémoire (statique vs dynamique)

Pour stocker la valeur d'une variable d'un programme il faut allouer, réserver, de la mémoire à ce programme. Cette allocation peut être dynamique ou statique.

Si l'allocation est **dynamique** elle se fait au moment où le programme veut accéder la variable la première fois, l'allocation se fait **durant l'exécution** du programme. En général elle se fait soit sur la pile (stack) soit sur le tas (heap). La durée de vie des variables dynamiques est plus petite que la durée de vie du programme.

L'allocation **dynamique** d'une variable se fait avant le programme exécute une instruction qui peut la référencer – la variable est visible, cette portion du code source s'appelle la **portée**.

Lorsque le processeur exécute du code hors de la portée de la variable, celle-ci est désallouée.

En C, une variable est allouée avant d'exécuter la première instruction du bloc où elle est définie et désallouée en quittant le bloc.

# Gestion de la mémoire

Lors de l'exécution du programme les variables dynamiques allouées implicitement utilisent la pile. Typiquement, lorsque le code d'une fonction alloue les **variables locales sur la pile**.

La mémoire allouée à la demande du programmeur, en C en appelant malloc()-free(), utilise le **tas**.

Si l'allocation est **statique** elle se fait au moment où le code du programme est initialisé – chargé en mémoire par le système d'exploitation. L'espace est inclut dans le fichier exécutable du programme (code source). La taille est calculée au moment de la compilation.

La mémoire allouée statiquement compose les sections .text, .bss, .data du programme assembleur.

Les variables allouées sur la pile doivent être allouées dans un ordre compatible avec la structure de pile. Sinon, on utilise le tas.

# Exemple de programme

```
#include <stdio.h>

int f(){
    static int i;      // on compile une fois avec une fois sans
    for(i=1;i<=5;i++)
        printf("valeur %d\n",i);
}

int main(void){
    f();
}
```

```
> arm-linux-gnueabi -fomit-frame-pointer exemple.c
> /usr/arm-linux-gnueabi/bib/objdump -d -a.out
```



# Avec

Contents of section .rodata:

**10500** 01000200 76616c65 75722025 640a0000 ....valeur %d...

Contents of section .data:

**21020** 00000000 00000000

0001044c <f>:

1044c:	e3a03001	mov	r3, #1	
10450:	e92d4070	push	{r4, r5, r6, lr}	
10454:	e1a02003	mov	r2, r3	
10458:	e1a06003	mov	r6, r3	
1045c:	e59f4028	ldr	r4, [pc, #40]	; 1048c <f+0x40>
10460:	e59f5028	ldr	r5, [pc, #40]	; 10490 <f+0x44>
10464:	e5843000	str	r3, [r4]	i est déclarée static
10468:	e1a01005	mov	r1, r5	
1046c:	e1a00006	mov	r0, r6	
10470:	ebffffa1	bl	102fc <__printf_chk@plt>	
10474:	e5942000	ldr	r2, [r4]	
10478:	e2822001	add	r2, r2, #1	
1047c:	e3520005	cmp	r2, #5	
10480:	e5842000	str	r2, [r4]	
10484:	dafffff7	ble	10468 <f+0x1c>	
10488:	e8bd8070	pop	{r4, r5, r6, pc}	
1048c:	0002102c	.word	0x0002102c	
10490:	00010504	.word	0x00010504	adresse pour la chaîne de caractère

# Sans

Contents of section .rodata:

**10504** 01000200 76616c65 75722025 640a00 ....valeur %d..

**00010438** <f>:

10438:	e52de004	push	{lr}	; (str lr, [sp, #-4]!)
1043c:	e24dd00c	sub	sp, sp, #12	
10440:	e3a03001	mov	r3, #1	le compteur sur la pile, non-statique
10444:	e58d3004	str	r3, [sp, #4]	
10448:	ea000005	b	10464 <f+0x2c>	
1044c:	e59d1004	ldr	r1, [sp, #4]	
10450:	e59f0028	ldr	r0, [pc, #40] ; 10480 <f+0x48>	
10454:	ebffffa1	bl	102e0 <printf@plt>	
10458:	e59d3004	ldr	r3, [sp, #4]	
1045c:	e2833001	add	r3, r3, #1	
10460:	e58d3004	str	r3, [sp, #4]	
10464:	e59d3004	ldr	r3, [sp, #4]	
10468:	e3530005	cmp	r3, #5	
1046c:	dafffff6	ble	1044c <f+0x14>	
10470:	e1a00000	nop		; (mov r0, r0)
10474:	e1a00003	mov	r0, r3	
10478:	e28dd00c	add	sp, sp, #12	
1047c:	e49df004	pop	{pc}	; (ldr pc, [sp], #4)
10480:	00010508	.word	0x00010508	adresse pour la chaîne de caractères

# Exemple de programme C

```
#include <stdio.h>
int foo(int a, int b){
    return (a+b);
}
void main(){
    int c;
    c = foo(3,4);
    printf("resultat=%d\n",c);
}
```

On compile

```
>arm-linux-gnueabi-gcc -S exemple.c
```

```
>less exemple.s
```

```
>arm-linux-gnueabi-gcc -S -fomit-frame-pointer exemple.c
```

```
>arm-linux-gnueabi-gcc -S -O exemple.c
```

# Exemple de programme C

```
.arch      arm5t
.fpu       softvfp
.eabi_attribute ....
...
.file      "exemple.c"
.text
.align 2
.global    foo
.syntax    unified
.arm
.type      foo, %function
```

soft floating-point  
des options

arm et thumb

défini le type d'un symbole

# Exemple de programme C

foo:

```
str    fp, [sp, #-4]!
add    fp, sp, #0
sub    sp, sp, #12
str    r0, [fp, #-8]
str    r1, [fp, #-12]
ldr    r2, [fp, #-8]
ldr    r3, [fp, #-12]
add    r3, r2, r3
mov    r0, r3
sub    sp, fp, #0
ldr    fp, [sp], #4
bx     lr
.size   foo, .-foo
.section .rodata
.align  2
.LC0:
.ascii  "resultat = %d\012\000"
```

main:

```
push   {fp, lr}
add    fp, sp, #4
sub    sp, sp, #8
mov    r1, #4
mov    r0, #3
bl     foo
str    r0, [fp, #-8]
ldr    r1, [fp, #-8]
ldr    r0, .L4
bl     printf
nop
sub    sp, fp, #4
pop    {fp, pc}

.L5:
.align  2
.L4:
.word   .LC0
```

.LC0:

Symbole local (.L), non visible dans le programme object (utiliser l'option -L pour debugger)

# Remarque

La gestion de la pile est standardisée dans un document appelé AAPCS (Procedure Call Standard for the ARM Architecture) pour les appels de procédures, voir le document sur le site [infocenter.arm.com](http://infocenter.arm.com), c'est une partie de l'ABI (Application Binary Interface, dans EABI E=Embedded).

Une pile pour un processeur ARM est de type Full Descending (FD)

Le pointeur de pile s'appelle **sp** (stack pointer) c'est le registre **r13** par convention.

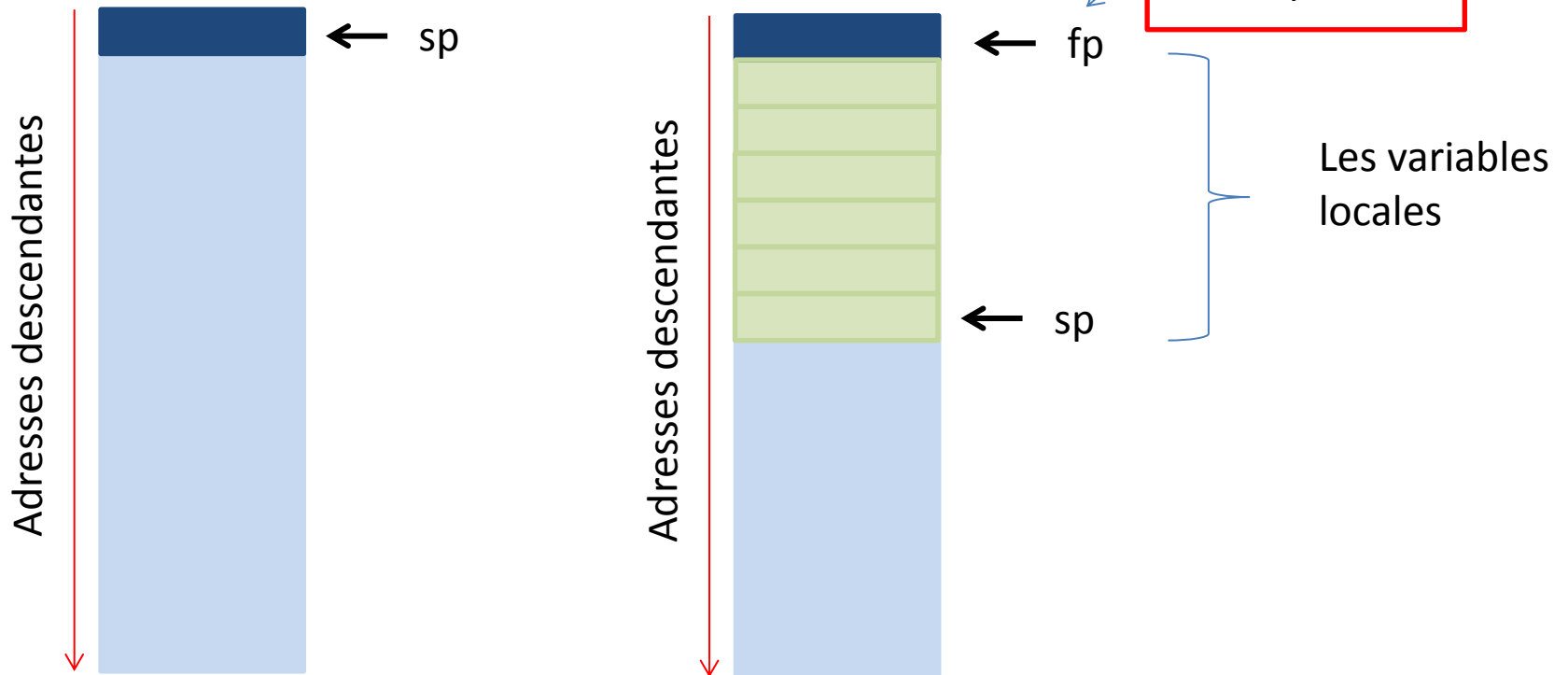
Le compilateur utilise un registre **fp**. C'est gcc pas ARM qui utilise fp (=r11), c'est une technique général des compilateurs pour gérer la pile.

fp = frame pointer

Le frame pointer n'est jamais nécessaire.

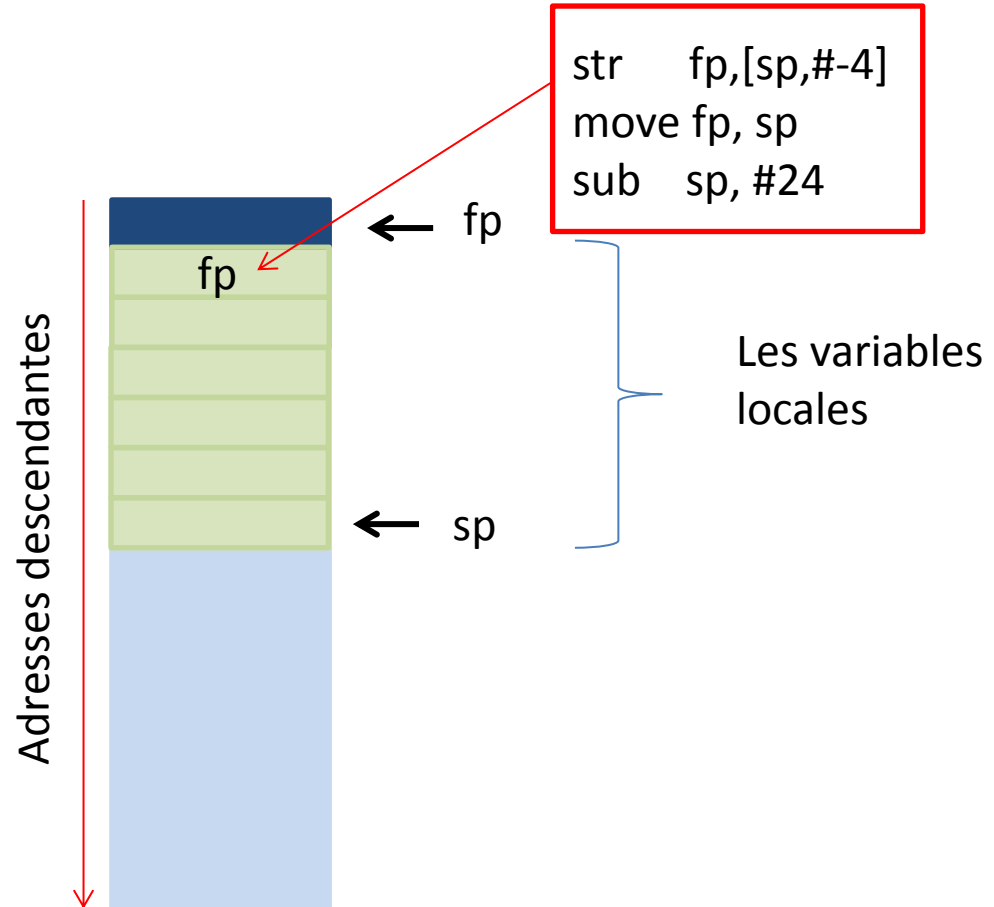
# Variables locales

Les variables locales sont déclarées sur la pile. Le pointeur `sp` peut varier mais pas `fp`, les références aux variables locales qui utilisent `fp` sont toujours les mêmes dans la procédure, par exemple, `ldr r0, [fp, #4]`.



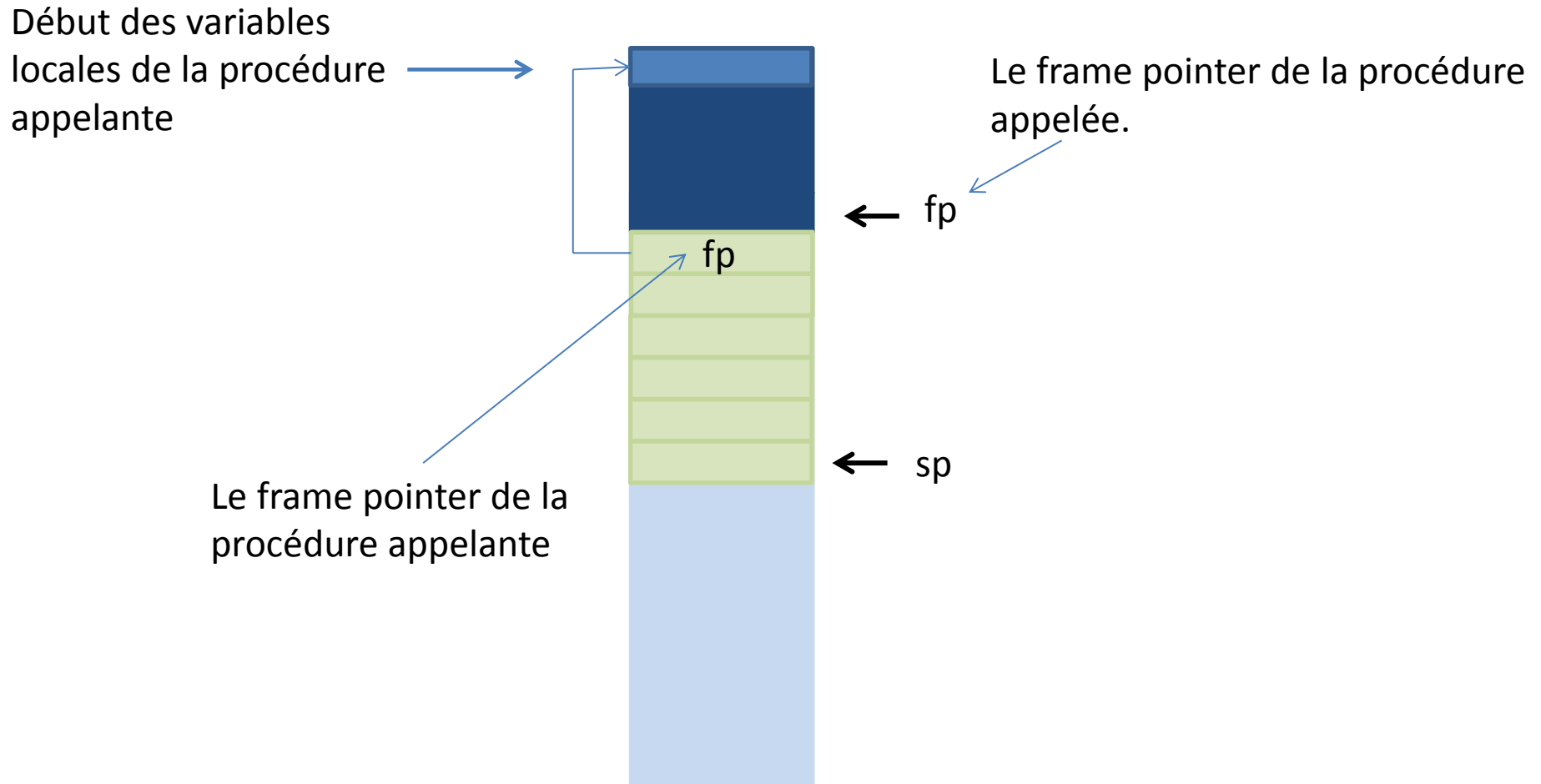
# Variables locales

Il faut préserver le contenu du fp avant de le modifier





# Variables locales



# Appel de procédure avec fp

```
bl    foo
```

```
...
```

```
...
```

```
foo:  str fp, [sp, #-4]!    // sauvegarde du fp de la procédure appelante
      add fp, sp, #0       // initialisation fp procédure appelée
      sub sp, sp, #xxx     // allocation mémoire pour les variables locales
      ....
      ....
      sub sp, fp, #0
      ldr fp, [sp], #4
      mov pc, lr
      bx  lr
```

# AAPCS

(Procedure Call Standard for the ARM Architecture)

Pour les appels aux procédures en assembleur, par exemple printf ou autres fonctions de la glib, les 4 premiers paramètres sont passés dans r0, ..., r3, les autres paramètres sont passés sur la pile.

La valeur de retour est dans r0.

L'adresse de retour est dans lr.

Si la pile est utilisée, c'est la routine appelante qui doit manipuler la pile (push et pop).

Les registres r0-r3 sont *volatiles* = leur valeur peut changer.

Les registres r4-r11 et r13 sont non-volatiles ils doivent être restaurés à la fin de la routine

Si r14=lr est utilisé (bl) il faut sauver sa valeur sur la pile.

# Ordre des paramètres

Sur la pile, il faut empiler les arguments de la fonctions en finissant par le 5<sup>ème</sup>.

```
printf(".... ",a,b,c,d,e,f);
```

**Pour la routine printf**, les paramètres se trouvent:

le pointeur sur la chaîne de caractère = r0

a=r1

b=r2

c=r3

d=[sp,#8]

e=[sp,#4]

f=[sp]

Pour empiler les paramètres dans l'ordre avec **stm**:

r0=d, r1=e, r2=f

```
stmfd sp!, {r0,r1,r2}
```

# Programme en C

```
#include <stdio.h>

static int x=5;
static int y=4;

int main(){
    int sum;
    sum = x+y;
    printf("sum = %d\n", sum);
    return 0;
}
```

```
>arm-linux-gnueabi-gcc -S -fomit-frame-pointer exemple.c
```

# Programme en C

```
.data
.align 2
x: .word 5
y: .word 4
.section .rodata
.align 2
.LC0: .ascii "sum = "d\012\000"
.text
main: str lr, [sp, #-4]!
      sub sp, sp, #12
      ldr r3, .L2
      ldr r2, [r3]
      ldr r3, .L2+4
      ldr r3, [r3]
      add r3, r2, r3
      str r3, [sp, #4]
      ldr r1, [sp, #4]
      ldr r0, .L2+8
      bl printf
      nop
      add sp, sp, #12
      ldr pc, [sp], #4
```

.L2: .align 2  
.word x  
.word y  
.word .LC0

} ldr r3, =x  
} ldr r3, =y

# Trouver le maximum

```
ldr    r1, =Value1 ; macro l'assembleur génère LDR R1,[pc, #<offset>]
ldr    r2, =Value2
cmp    r1, r2
bhi    Done ; comparaison unsigned, Higher R1>R2 }
mov    r1, r2                                     } MOVLS R1,R2
```

Done:

```
str r1, Result
. . . . .
```

```
Value1: .word    0x12345678
Value2: .word    0x87654321
Result: .word    0
```

# Addition sur 64 bits

```
adr    r0, Value1                                macro add r0, pc, #offset. même section

ldr    r1, [r0]
ldr    r2, [r0, #4]
adr    r0, Value2
ldr    r3, [r0]
ldr    r4, [r0, #4]
adds   r6, r2, r4    // ADD + S
adc     r5, r1, r3    // ADD + Carry
adr    r0, Result
str     r5, [r0]
str     r6, [r0, #4]
. . . . .
```

```
Value1: .word    0x12A2E640, 0xF2100123
Value2: .word    0x001019BF, 0x40023F51
Result: .word    0
```



# Le program counter (pc)

Le contenu du program counter (pc) dépend de l'état du processeur:

- Dans l'état ARM, le processeur exécute une instruction ARM (32 bits) et le pc contient l'adresse de l'instruction courante + 8 (2 instructions).
- Dans l'état Thumb, le processeur exécute une instruction Thumb (16 bits) et le pc contient l'adresse de l'instruction courante + 4 (2 instructions).

Ecrire une adresse dans le pc revient à effectuer un branchement à cette adresse – une rupture de séquence.

# Switches

CMP	R0, #8
ADDLT	pc,pc,R0, LSL#2
B	methode_d
B	methode_0
B	methode_1
B	methode_2
B	methode_3
B	methode_4
B	methode_5
B	methode_6
B	methode_7

CMP	R0, #8
DRLT	pc,[pc,R0, LSL#2]
B	methode_d
.word	methode_0
.word	methode_1
.word	methode_2
.word	methode_3
.word	methode_4
.word	methode_5
.word	methode_6
.word	methode_7

# Addition d'une liste de nombres 16 bits

```
.text
ldr    r0, =Table
eor    r1, r1, r1
ldr    r2, Length

Loop:  ldr    r3, [r0]
      add    r1, r1, r3
      add    r0, r0, #+4
      subs   r2, r2, #0x1
      bne    Loop
      str    r1, Result

      ldr r3,[r0],#4

.data
Table: .word    2040
      .align   2           alignement adresse divisible par 4
      .word    0x1C22
      align    2
      .word    0x0242
      .align   2
TablEnd: .word    0
Length:  .word    (TablEnd - Table) / 4   on peut aussi utiliser (. - Table - 4)/4
      .align   2
Result:  .word    0
```

