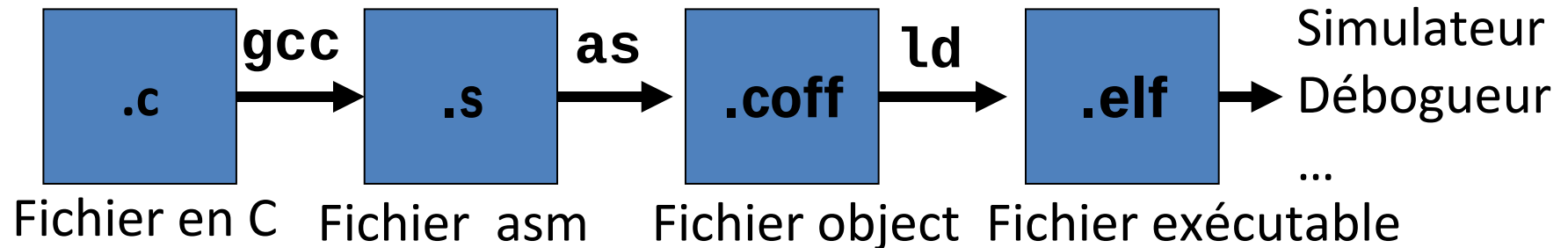


Ch. 4

Programmation en langage assembleur

Mounir Boukadoum

Flux de développement typique (Linux)



- COFF (common object file format)
- ELF (extended linker format)
- Segments dans le fichier object
 - Text : code
 - Data : variables globales initialisées
 - BSS : variables globales sans valeurs initiales

La chaîne GNU et le format GAS

- GNU (« GNU's Not Unix ») : système d'exploitation ressemblant à Unix et basé sur du logiciel libre.
 - Développé au MIT en 1983
 - Comprend une chaîne d'outils de développement de programmes
- GAS (« GNU Assembler ») : Assembleur de la chaîne GNU



Le format GAS

Déclare du code à venir → **.text**

.file "test.s"

Déclare un symbole public → **.global main**

.type main, %function

↑
Spécifie le type d'un symbole (objet ou fonction)

main:

MOV R0, #100

ADD R0, R0, R0

SWI #11

signale la fin du programme → **.end**

Code pour retourner au système d'exploitation

Format général des instructions

label	opération	arguments	commentaires
main:	LDR	R1, valeur	@ Charge valeur
	STR	R1, resultat	
	SWI	#11	
valeur:	.word	0x0000C123	
resultat:	.word	0	

Structures de contrôle

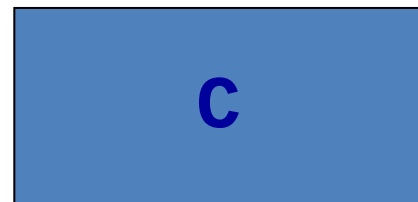
- Séquence
- Décision: if-then-else, switch
- Itération: repeat-until, do-while, for
- Décomposition: Le problème à résoudre est décomposé en problèmes plus petits qui sont plus simples à résoudre (sous-programmes/fonctions/procédures)

Décision

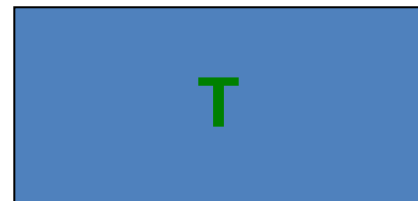
- If-then-else
- switch

If

if **C** then **T** else **E**



BNE else



B endif

else:



endif:

// Fonction maximum
if (R0>R1) then R2=R0;
else R2=R1;

CMP R0, R1

BLE else

MOV R2, R0

B endif

else: **MOV** R2, R1

endif:

If

```
// Fonction maximum  
if (R0>R1) then R2=R0; else R2=R1;
```

Deux alternatives :

```
CMP    R0, R1
```

```
MOVGT  R2, R0
```

```
MOVLE  R2, R1
```

```
MOV    R2, R0
```

```
CMP    R0, R1
```

```
MOVLE  R2, R1
```

```
CMP R0, R1
```

```
BLE else
```

```
MOV R2, R0
```

```
B endif
```

```
else: MOV R2, R1
```

```
endif:
```

If

```
if (R1==1 || R1==5 || R1==12) R0=1;
```

```
TEQ    R1, #1
```

```
TEQNE  R1, #5
```

```
TEQNE  R1, #12
```

```
MOVEQ  R0, #1
```

```
BNE faux
```

If

```
if (R1==0) zero  
else if (R1>0) plus  
else if (R1<0) neg
```

```
TEQ    R1, #0
```

```
BMI    neg
```

```
BEQ    zero
```

```
BPL    plus
```

```
neg:    . . .
```

```
B exit
```

```
Zero:    . . .
```

If

R0=abs (R0)

TEQ R0, #0

RSBMI R0, R0, #0

Saut conditionnels multiples

CMP R0, #`0'

BCC autre @ < '0'

CMP R0, #`9'

BLS chiffre @ entre '0' et '9'

CMP R0, #`A'

BCC autre

CMP R0, #`Z'

BLS lettre @ entre 'A' et 'Z'

CMP R0, #`a'

BCC autre

CMP R0, #`z'

BHI autre @ pas entre 'a' et 'z'

letter: ...

Switch

switch (exp) {	e=exp;
case c1: S1; break;	if (e==c1) {S1}
case c2: S2; break;	else
...	if (e==c2) {S2}
case cN: SN; break;	else
default: SD;	...
}	

Switch

```
switch (R0) {  
    case 0: S0; break;  
    case 1: S1; break;  
    case 2: S2; break;  
    case 3: S3; break;  
    default: err;  
}
```

Lent si beaucoup
de cas !

```
CMP R0, #0  
BEQ S0  
CMP R0, #1  
BEQ S1  
CMP R0, #2  
BEQ S2  
CMP R0, #3  
BEQ S3
```

```
err: ...  
    B exit  
  
S0: ...  
    B exit
```

Switch

ADR R1, JMPTBL Et si le choix est entre M and N?

CMP R0, #3

LDRLS PC, [R1, R0, LSL #2]

err:...

B exit

S0: ...

JMPTBL:

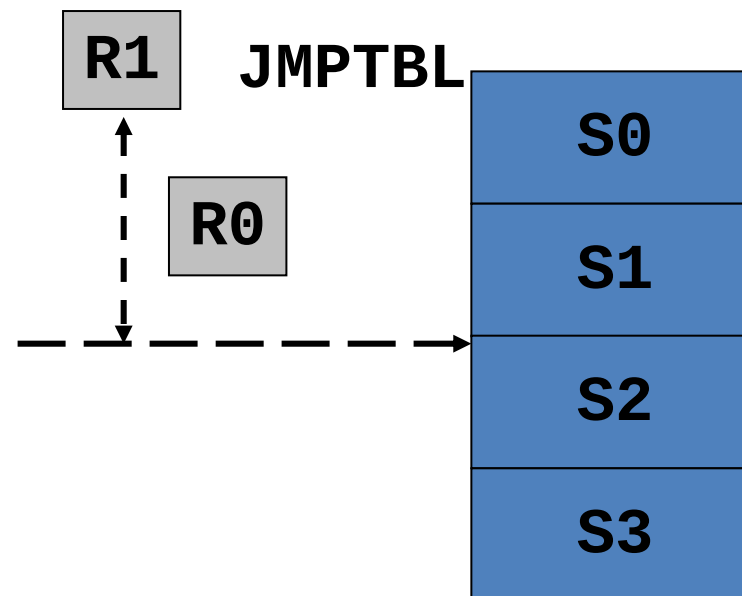
.word S0

.word S1

.word S2

.word S3

Pour N grand ou des valeurs dispersées, un tableau de hashage peut être plus efficace



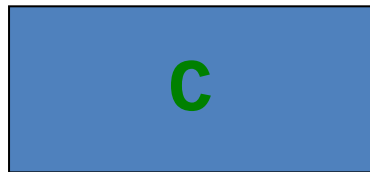
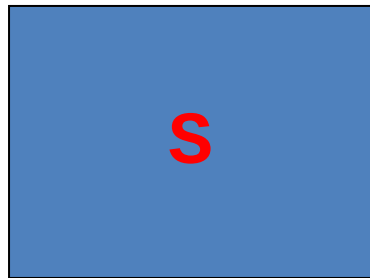
Itération

- repeat-until
- do-while
- for

Boucle *do*

do { **S** } while (**C**)

loop:



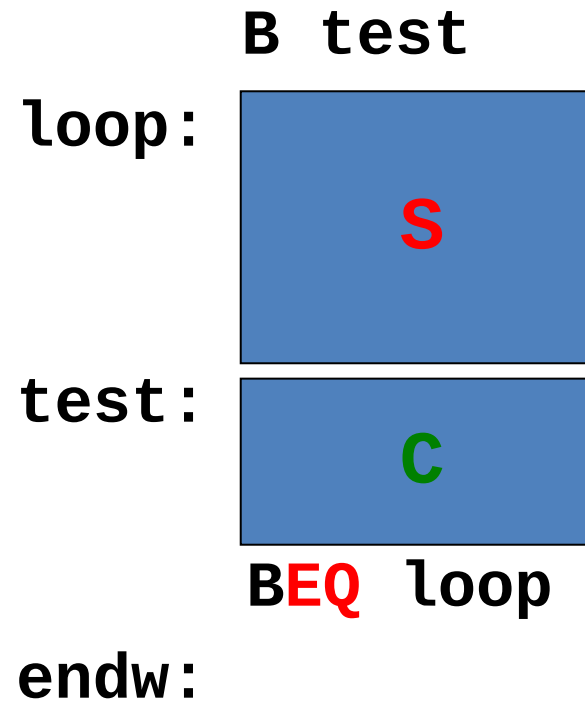
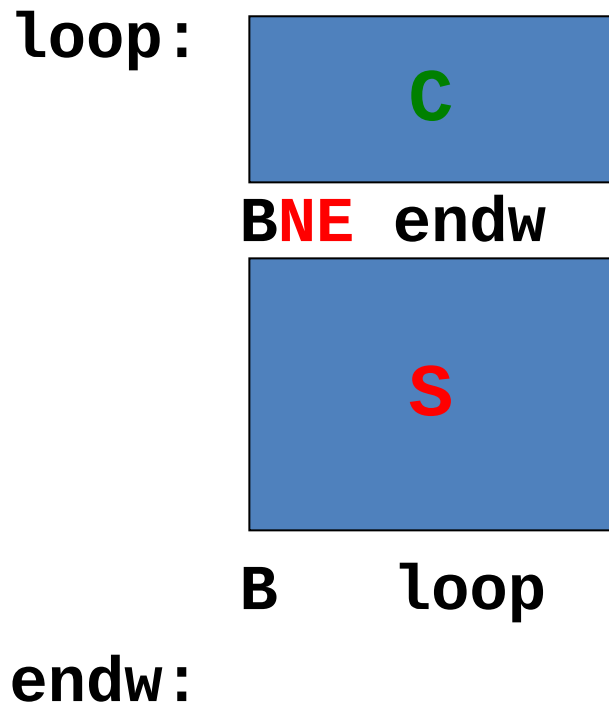
BEQ

loop

endw:

Boucle *while*

while (**C**) { **S** }

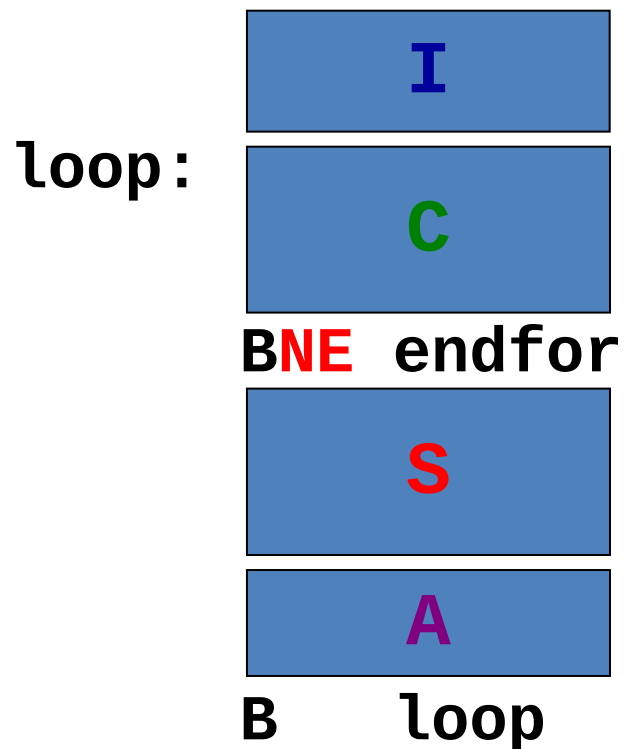


PGDC

```
int pgdc (int i, int j)    Loop:  CMP    R1, R2
{                               SUBGT R1, R1, R2
    while (i!=j)             SUBLT R2, R2, R1
    {                         BNE     loop
        if (i>j)
            i -= j;
        else
            j -= i;
    }
}
```

Boucle *for*

for (**I** ; **C** ; **A**) { **S** }



endfor:

for (i=0; i<10; i++)
{ a[i]:=0; }

MOV R0, #0
ADR R2, A
MOV R1, #0
loop: CMP R1, #10
BGE endfor
STR R0, [R2, R1, LSL #2]
ADD R1, R1, #1
B loop

endfor:

for loops

```
for (i=0; i<10; i++)  
{ ... }
```

Exécuter une boucle un
nombre constant de fois

```
MOV R1, #0  
loop: CMP R1, #10  
BGE endfor  
@ code ...  
ADD R1, R1, #1  
B loop
```

endfor:

```
MOV R1, #10  
loop:  
@ code ...  
SUBS R1, R1, #1  
BNE loop
```

endfor:

- Il est plus efficace de compter vers le bas

Procédures

- Arguments: variables passées à une fonction
- Paramètres: valeurs spécifiées dans un appel

```
void func(int a, int b) Appelé
{
    ...
}
int main(void) Appelant
{
    func(100, 200);
    ...
}
```

The diagram illustrates the relationship between function parameters and arguments. A green arrow points from the text "paramètres" to the parameters "int a, int b" in the function definition. A blue arrow points from the text "arguments" to the values "100, 200" in the function call.

Procédures

main:

```
...  
BL  fonc  
...  
  
.end
```

fonc:

```
...  
...
```

.end

- Comment passer les paramètres? Par registres ? Par pointeur ? Dans quel ordre ?

Procédures

main:

Appelant

@ R5 utilisé

BL fonc

@ R5 utilisé

...

...

.end

fonc:

...

@ R5 utilisé

...

...

.end

Appelé

- Qui doit sauvegarder R5?

Procédures (Sauvegarde par l'appelant)

main: Appelant

@ R5 utilisé

@ R5 sauvé

BL fonc

@ R5 remis en état

@ R5 utilisé

.end

fonc:

...

@ R5 utilisé

.end

Appelé

Procédures (sauvegarde par l'appelé)

main:	Appelant			Appelé
	@ R5 utilisé	→	fonc:	@ R5 sauvé
	BL fonc			...
	@ R5 utilisé			@ R5 utilisé
				@ R5 remis en état
	.end			.end

- Il faut un protocole

ARM Procedure Call Standard (APCS)

- ARM Ltd. A défini des règles pour les appels et les sorties de procédures afin que :
 - Le code objet généré par différents compilateur soit cohérent au niveau de l'éditeur de liens
 - Les appels de procédures peuvent avoir lieu entre un langage de haut niveau et du code assembleur
- APCS définit
 - L'usage des registres
 - L'utilisation de la pile (R13)
 - Le format d'une trame de données dans la pile
 - Le mécanisme de passation des arguments

Règle d'utilisation des registres dans APCS

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

Règle d'utilisation des registres dans APCS

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

- Utilisés pour passer les premiers 4 paramètres
- Sauvés par l'appelant si requis

Règle d'utilisation des registres dans APCS

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

- Le contenu doit pas être le même au retour
- Sauvés par l'appelé au besoin

Règle d'utilisation des registres dans APCS

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

- Registres dédiés
- Peuvent être utilisés si les contenus sont saués au préalable.

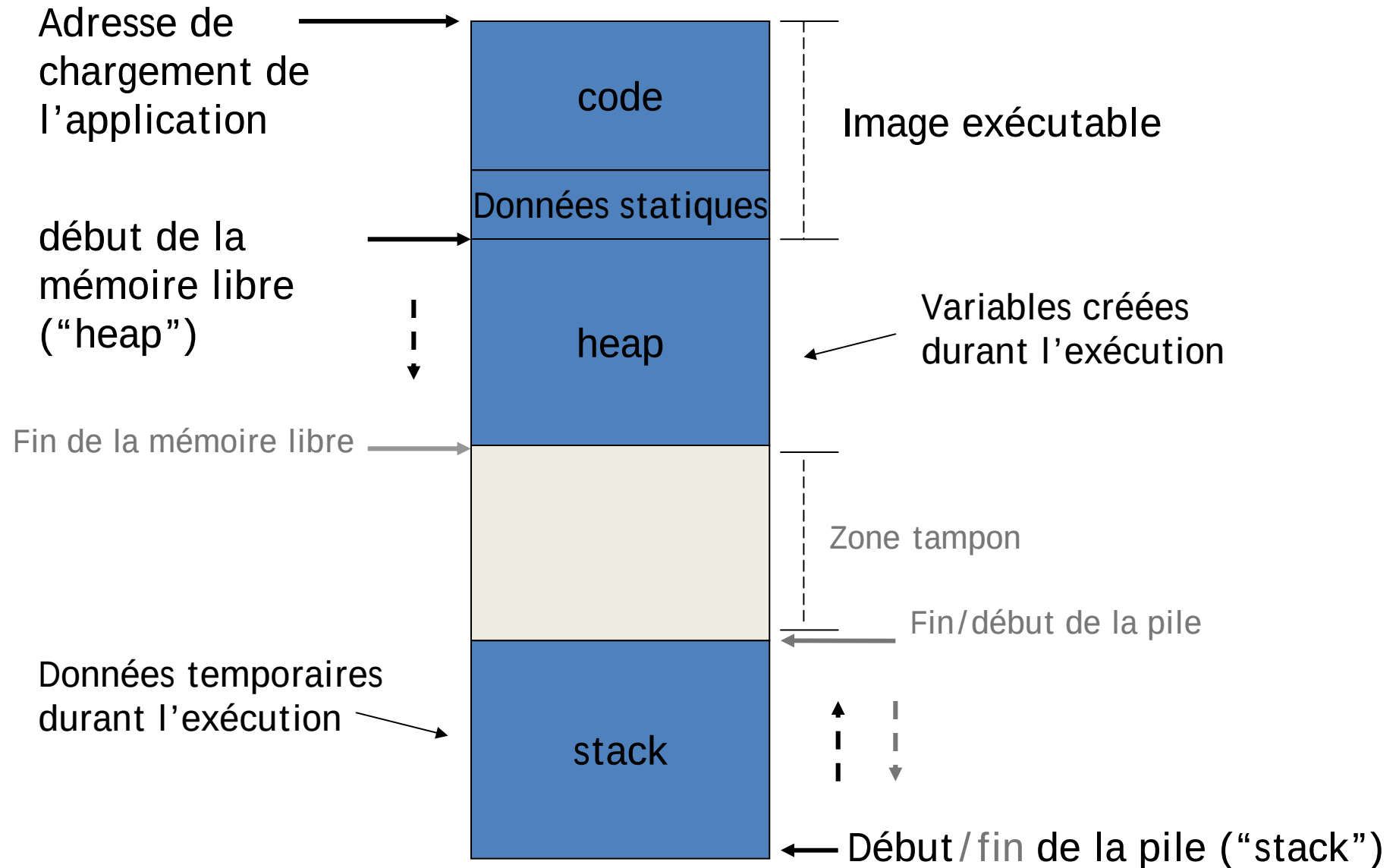
Passation des paramètres

- Les premiers quatre paramètres de 8-32 bits sont passés dans R0 - R3.
- Les paramètres restant sont sauvés dans la pile en ordre inverse.
- Les procédures avec quatre paramètres ou moins sont plus efficaces.

Valeur de retour

- R0 pour une valeur 8 – 32 bits
- R0-R1, R0-R2, ou R0-R3 pour une valeur qui demande 2~4 mots

Allocation typique de l'espace mémoire



Accès aux arguments

- Argument passé dans un registre: lire le registre
- Argument passé dans la pile : utiliser le pointer de pile (R13, alias lr, alias SP) en mode d'adressage relatif
- Constante : Utiliser le PC en mode d'adressage relatif
- Variable locale: allouer l'espace dans la pile et utiliser le pointer de pile en mode d'adressage relatif
- Variable globale : allouée dans la mémoire libre et accédée avec le registre R9 en mode d'adressage relatif

Procedure

main:

LDR R0, #0

. . .

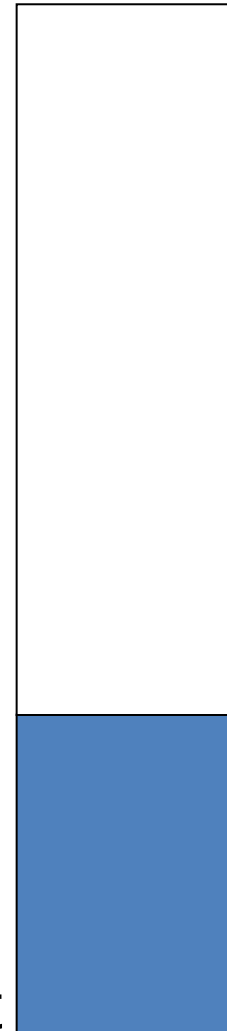
BL func

. . .

Bas

haut

Pile



Procedure

```
fonc:  STMFD SP!, {R4-R6, LR}
        SUB    SP, SP, #0xC
        . . .
        STR    R0, [SP, #0] @ v1=a1
        . . .
        ADD    SP, SP, #0xC
        LDMFD  SP!, {R4-R6, PC}
```

