



Programmation assembleur ARM

Dr. Eng. Chiheb Ameer ABID

[in /in/chiheb-ameur-abid](#)
chiheb.abid@gmail.com

Les instructions de test

Les instructions de test

- Les instructions de tests permettent de réaliser des tests arithmétiques et logiques entre deux valeurs en mettant à jour le registre d'état CPSR : Current Program Status

Structure du registre d'état CPSR

31	30	29	28	27	-	24	-	19-16	-	9	8	7	6	5	4-0
N	Z	C	V	Q		J		GE		E	A	I	F	T	M

- Negative : N prend 1 si le résultat est négatif, sinon 0
- Zero : Z prend 1 si le dernier résultat est nul, sinon 0
- Carry : bit de retenue (décalage, addition)
- OVERflow : bit de débordement (opération signée)
- Q : bit de saturation utilisé pour certaines instructions (\geq ARMv6)

Plan

- L'exécution conditionnelle et les branchements
- Accès à la mémoire, la gestion des piles
- Les sous-programmes
- Les appels système
- Mode Thumb

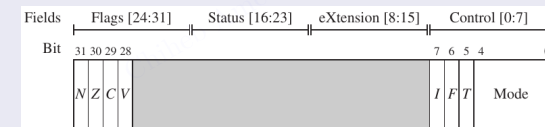
Lecture/Écriture du registre CPSR

- Deux instructions permettent d'accéder au registre CPSR

- mrs** Move to Register from Status
- msr** Move to Status from Register

```
1 mrs{<cond>} Rd, <CPSR|SPSR>{<fields>}
2 msr{<cond>} <CPSR|SPSR>{<fields>}, Rs
```

- cond suffixe de condition
- fields : spécifier les champs F, S, X et C



Name	Effect	Description
mrs	$Rd \leftarrow CPSR SPSR$	Move from Status Register
msr	$CPSR SPSR \leftarrow Rn$	Move to Status Register



En mode utilisateur, la modification n'est possible que pour les bits N, C, V et 0

Les instructions de test

Exemple : Lecture/Écriture du registre CPSR

```

1 MRS R0, CPSR @ Sauvegarde CPSR dans R0
2
3 ORR R1, R0, #0x80 @ Met le bit 7 (I bit - IRQ disable)
4 MSR CPSR_c, R1 @ Applique la modification
5
6 MSR CPSR_cxsf, R0 ; Restaure CPSR complet
    
```

Les instructions de test

Exécution conditionnelle des instructions

- ↳ L'architecture ARM 32 bits (ARMv4 à ARMv7) introduit le mécanisme d'exécution conditionnelle
 - ☛ Exécuter (ou non) une instruction en fonction des drapeaux du registre d'état du programme CPSR
- ✅ Optimiser le flux de contrôle en évitant les branches inutiles
 - ☛ Jusqu'à 30 % d'amélioration dans certains codes
- ↳ Par défaut, les instructions ARM n'affectent pas les drapeaux du CPSR
- ↳ Pour qu'une instruction modifie un registre d'état, on la postfixe avec **s**

```

1 SUB R0, R1, R2 @ Soustraction sans mise à jour des flags
2 SUBS R0, R1, R2 @ Soustraction avec mise à jour : Z=1 si R0=0, etc.
    
```

Les instructions de test

Les instructions de test

Exécution conditionnelle des instructions

- ↳ L'exécution d'une instruction peut être conditionnée selon les valeurs des flags du registre d'état

<cond>	English Meaning
a1	always (this is the default <cond>)
eq	Z set (=)
ne	Z clear (≠)
ge	N set and V set, or N clear and V clear (≥)
lt	N set and V clear, or N clear and V set (<)
gt	Z clear, and either N set and V set, or N clear and V set (>)
le	Z set, or N set and V clear, or N clear and V set (≤)
hi	C set and Z clear (unsigned >)
ls	C clear or Z (unsigned ≤)
hs	C set (unsigned ≥)
cs	Alternate name for HS
lo	C clear (unsigned <)
cc	Alternate name for LO
mi	N set (result < 0)
pl	N clear (result ≥ 0)
vs	V set (overflow)
vc	V clear (no overflow)

Les instructions de test

- ↳ Il est possible de mettre à jour le registre d'état avec les instructions des tests

Instruction	Description	Commentaire
CMP Rn, operand2	Compare	CPSR ← Rn - operand2
CMN Rn, operand2	Compare negative	CPSR ← Rn + operand2
TST Rn, operand2	Test	CPSR ← Rn and operand2
TEQ Rn, operand2	Test equal	CPSR ← Rn xor operand2

- ↳ Ces instructions ne rangent aucun résultat dans les autres registres

Instruction de branchement inconditionnel

Instruction de branchement inconditionnel

- Une instruction de branchement inconditionnel permet de fixer la prochaine instruction à exécuter

```

i0
i1
B etiquette PC=etiquette
i2
i3
i4
etiquette: i5
...
    
```

Instructions de branchement conditionnel

Instructions de branchement conditionnel

- Une instruction de branchement conditionnel effectue un branchement selon une condition relative au registre d'état

Mnémonique	Interprétation	commentaire
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out

Instructions de branchement conditionnel

Instructions de branchement conditionnel

- Une instruction de branchement conditionnel effectue un branchement selon une condition relative au registre d'état

Mnémonique	Interprétation	commentaire
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Arithmetic comparison gave lower or same

La structure de contrôle IF-THEN-ELSE

La structure de contrôle IF-THEN-ELSE

```

if C then T else E // find maximum
if (R0>R1) then R2:=R0
else R2:=R1

C
BNE else
T
B endif
else:
E
endif:

CMP R0, R1
BLE else
MOV R2, R0
B endif
else: MOV R2, R1
endif:
    
```

La structure de contrôle SWITCH

La structure de contrôle SWITCH

```

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

```

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
    
```



Les boucles FOR

Les boucles FOR

```

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

```

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

```

loop:
    I
    C
    BNE endfor
    S
    A
    B loop
endfor:
    
```

```

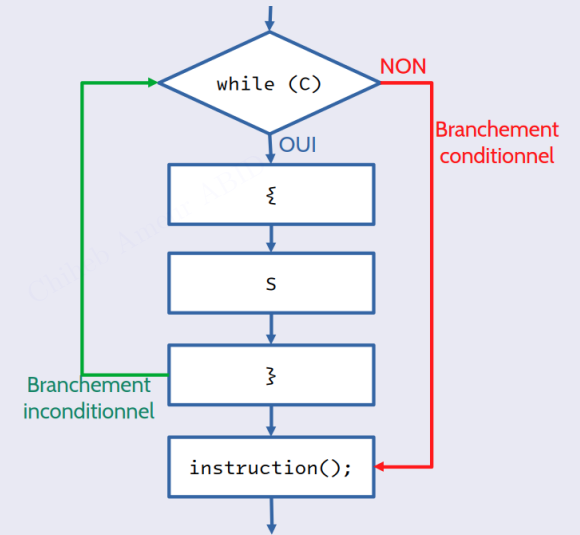
MOV R0, #0
ADR R2, A
MOV R1, #0
loop:
    CMP R1, #10
    BGE endfor
    STR R0, [R2, R1, LSL #2]
    ADD R1, R1, #1
    B loop
endfor:
    
```



La boucle WHILE

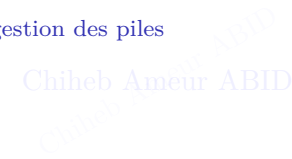
```

while (C) {
    S;
}
instruction();
    
```



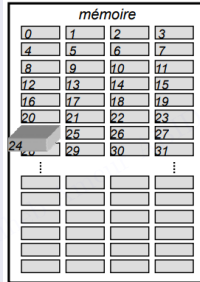
Plan

- 1 L'exécution conditionnelle et les branchements
- 2 Accès à la mémoire, la gestion des piles
- 3 Les sous-programmes
- 4 Les appels système
- 5 Mode Thumb



Adressage avec ARM

- Sur le processeur ARM, une cellule mémoire contient 1 octet (8 bits) et chaque cellule mémoire possède une adresse



- Les échanges entre la mémoire et les registres se font sur des données de 32 bits, les types de données sont en format 8, 16 ou 32
- Il faut certaines règles pour garder un accès cohérent aux données
 - L'accès à un mot mémoire (32 bits) doit être aligné sur des adresses divisibles par 4
 - L'accès à un demi-mot (16 bits) doit être aligné sur des adresses divisibles par 2
 - Tout accès non aligné peut produire un comportement non prévisible



Les instructions de chargement

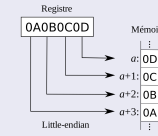
Les instructions de chargement

- Instructions sur les mots (32 bits)
 - LDR Rd, effective address Load Register
 - STR Rd, effective address Store Register
- Instructions sur les demi-mots (Half-word 16 bits)
 - LDRH Rd, effective address (Unsigned)
 - STRH Rd, effective address
 - LDRSH Rd, effective address (Signed)
 - STRSH Rd, effective address
- Instructions sur les octets (Byte 8 bits)
 - LDRB Rd, effective address (Unsigned)
 - STRB Rd, effective address
 - LDRSB Rd, effective address (Signed)
 - STRSB Rd, effective address



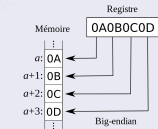
Arrangement des données dans la mémoire

- Little endian : la partie la moins significative se trouve à l'adresse la plus petite



- Par défaut, linux configure les processeurs ARM avec little-endian

 - Big-Endian : la partie la plus significative se trouve à l'adresse la plus petite



- À partir d'ARMv4 et v5, l'organisation des données en mémoire est par défaut en little-endian
- Le bit 9 (E) du registre CPSR contrôle l'endianess
 - 0 : little-endian ; 1 : big-endian



Les instructions de chargement

Exemple d'accès à la mémoire

Instruction	Rd[31...24]	Rd[23...16]	Rd[15...8]	Rd[7...0]
LDR	Mem[e.a] _{word}			
LDRH	0000...0000	Mem[e.a] _{half-word}		
LDRSH	Extension bit de signe		Mem[e.a] _{half-word}	
LDRB	0000...0000	Mem[e.a] _{byte}		
LDRSB	Extension bit de signe			Mem[e.a] _{byte}

	addr	addr+1	
ldr _{sb}	11001101		11111111 11111111 11111111 11001101
ldr _{sh}	11001101	10100101	11111111 11111111 10100101 11001101
ldr _{sb}	01001101		00000000 00000000 00000000 01001101
ldr _{sh}	11001101	00100101	00000000 00000000 00100101 11001101



Modes d'adressage

Modes d'adressage

ARM possède deux modes d'adressage

1 Mode d'adressage indirect par registre

- L'adresse d'une donnée est spécifiée à travers un registre
- LDR r0, [r1]

2 Mode d'adressage indexé

- Deux registres sont utilisés pour spécifier l'adresse d'une donnée : le registre de base et le registre d'index



Le mode d'adressage direct disponible sur d'autres architectures est interdit par ARM

Modes d'adressage

Types d'adressage indirect

Notation	Nom	Adresse effective
[Rn]	Adressage indirect	Valeur de Rn
[Rn, offset]	Adressage indirect pré-indexé	Valeur de Rn + offset
[Rn, offset!]	Adressage indirect pré-indexé automatique	Valeur de Rn + offset
[Rn], offset	Adressage indirect post-indexé (auto)	Valeur de Rn

- Rn = registre de base
- offset =
 - #literal
 - ± Rm
 - ± Rm, shift

- Les modes pré-indexé automatique et post-indexé modifient le registre Rn
 - On effectue deux opérations : accès mémoire ET modification du registre de base Rn
- Mode pré -indexé automatique
 - On fait d'abord $Rn \leftarrow Rn + offset$
 - Puis, on accède à mem[Rn]
- Mode post-indexé
 - On accède d'abord à mem[Rn]
 - Puis, on fait $Rn \leftarrow Rn + offset$

Chargement/Sauvegarde multiple

Chargement/Sauvegarde multiple

- ARM possède deux instructions pour le chargement et la sauvegardes de plusieurs registres

- ldm Load Multiple Registers
- stm Store Multiple Registers

Syntaxe

1 <op><variant> Rd{!}, <register_list>

- <op> : stm ou ldm
- <variant> : une variante choisie à partir des tableaux suivants

Block Copy Method		Stack Type	
Variant	Description	Variant	Description
ia	Increment After	ea	Empty Ascending
ib	Increment Before	fa	Full Ascending
da	Decrement After	ed	Empty Descending
db	Decrement Before	fd	Full Descending

- Le !, optionnel, spécifie si la valeur du registre Rd est modifiée avec chaque opération

Chargement/Sauvegarde multiple

Chargement/Sauvegarde multiple

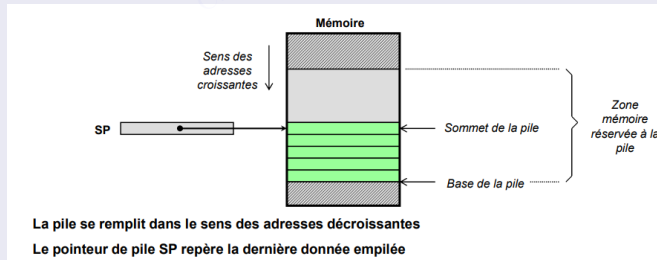
```

1 ldmia r8,{r0-r7} @ load eight words from source specified by r8
2 stmia r9,{r0-r7} @ store them in destination
3 stmb r9!,{r0-r7} @ Store 8 registers at the location pointed to by r9,
4 @ and increment r9 BEFOR each store. After executing,
5 @ r9 will point to the last item stored.
6 ldmia r4,{r0,r2,r3} @ Load r0, r2, and r3 at the location pointed to by r4
7 @ and increment the address AFTER each store. After
8 @ executing, r4 will contain its original value.
    
```

Gestion de la pile

Structure de la pile

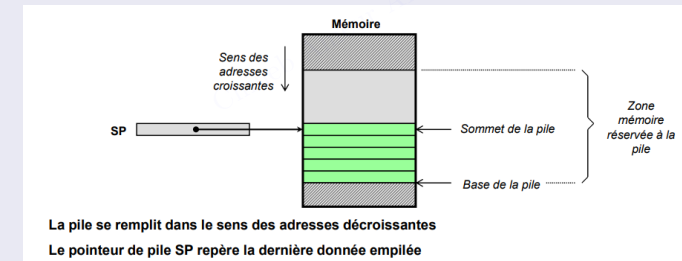
- La pile réside dans un espace mémoire qui est réservée spécifiquement à cet usage
- Les données sont rangées dans des cellules adjacentes au fur et à mesure qu'elles sont empilées
- Pour repérer le niveau de remplissage de la pile, on utilise un registre pointeur de pile (Stack Pointer SP). C'est le registre R13 du processeur ARM qui remplit cette fonction
- Une pile fonctionne toujours en mode FILO (First In Last Out)



Gestion de la pile

Empilement d'une donnée

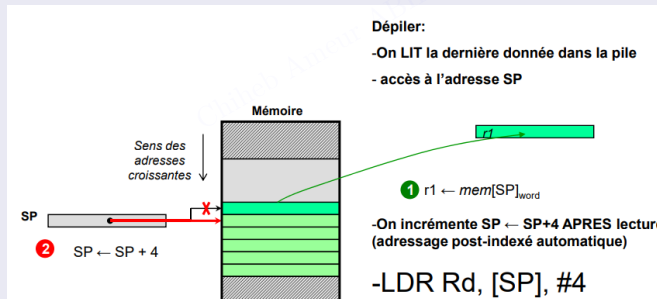
- L'empilement (PUSH) d'une donnée s'effectue en utilisant l'instruction STR
- STR empile la valeur d'un registre toute entière (32 bits)



Gestion de la pile

Dépilement d'une donnée

- L'empilement (PUSH) d'une donnée s'effectue en utilisant l'instruction STR
- LDR dépile la dernière valeur enregistrée dans la pile



Gestion de la pile

Modes de gestion d'une pile

- Full Ascending : le sommet de la pile indique le dernier élément empilé et la pile évolue avec des adresses croissantes
 - PUSH : Incrémenter SP avant de remplir
 - POP : Récupérer la valeur puis décrémenter
- Full Descending : le sommet de la pile indique le dernier élément empilé et la pile évolue avec des adresses décroissantes
 - PUSH : Décrémenter SP avant de remplir
 - POP : Récupérer la valeur puis incrémenter
- Empty Ascending : le sommet de la pile indique la première case libre et la pile évolue avec des adresses croissantes
 - PUSH : Remplir, puis incrémenter SP
 - POP : Décrémenter SP, puis récupérer la valeur
- Empty Descending : le sommet de la pile indique la première case libre et la pile évolue avec des adresses décroissantes
 - PUSH : Remplir, puis décrémenter SP
 - POP : Incrémenter SP, puis récupérer la valeur



Gestion de la pile

Modes de gestion d'une pile

- ↳ Sous linux, le mode de gestion des piles est Full Descending
- ↳ Deux mnémoniques pour empiler et dépiler plusieurs registres en mode Full
 - ➊ PUSH : STMFD SP!, {liste des registres}
STM : STore Multiple
 - ➋ POP : LDMFD SP!, {liste des registres}
LDM : LoaD Multiple

Plan

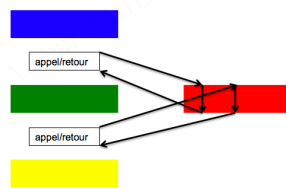
- 1 L'exécution conditionnelle et les branchements
- 2 Accès à la mémoire, la gestion des piles
- 3 Les sous-programmes
- 4 Les appels système
- 5 Mode Thumb

Les sous-programmes

Les sous-programmes



Ici le bloc de code en rouge est dupliqué



Ici on crée un sous-programme correspondant à ce bloc, et on l'appelle quand on en a besoin.

Les sous-programmes

Les sous-programmes

- ↳ L'appel à une procédure se fait à l'aide d'un branchement qui permet de retourner à l'adresse de appelant
- ↳ La pile permet la mise en place de sous-programmes ou procédure
 - À l'entrée d'un sous programme, on sauvegarde les registres utilisés comme variables locales
 - Avant de sortir, on restaure la valeur de ces registres

Les sous-programmes

L'instruction Branch and Link

- ↳ L'instruction `BL étiquette` permet de sauvegarder le registre PC dans le registre LR (Le registre R14), puis d'effectuer un branchement vers l'instruction spécifiée par `étiquette`



`BL étiquette` $\text{LR} \leftarrow \text{PC}$; $\text{PC} \leftarrow \text{étiquette}$;

```

1  @@ Code du sous-programme
2  @ R1 : le premier paramètre
3  @ R2 : le deuxième paramètre
4  PERMUTER:
5      STMFD SP!,R3,R14 @ Sauvegarde du contexte
6      MOV R3,R1
7      MOV R1,R2
8      MOV R2,R3
9      LDMFD SP!,R3,R15 @ Restauration du contexte
10
11  @@ Appel au sous-programme
12  MOV r1,5
13  MOV r2,10
14  BL PERMUTER

```



Appels des routines C standards

Exemple : appel de la routine printf

- ↳ Le prototype de la fonction `printf` : `printf(const char *format , ...)`
- ↳ Le prototype de la fonction `scanf` : `scanf(const char *format , ...)`
- ↳ Exemple d'appel à la fonction `printf`

```

.data
format: .asciz "Valeur %d affichée"
...
MOV R1,#20
LDR r0,=format
BL printf

```



Appels des routines C standards

Appels des routines C standards

- ↳ Il est possible de faire l'appel à une routine C depuis assembleur
- ↳ Le passage de paramètres à une routine écrite en C se fait selon la convention suivante :
 - ↳ Les registres : les 4 premiers paramètres sont passés à travers les registres R0-R3
 - ↳ A partir du 5ème paramètre, on utilise **la pile**
 - ↳ La valeur de retour à l'appelant est spécifiée à travers le registre R0



Appels des routines C standards

Exemple d'appel de la routine printf

- ↳ Soit à effectuer l'appel

```

printf("The results are: %d %d %d %d %d",i,j,k,l,m);

```

```

ldr r3,=m @ load last argument ('m')
ldr r3,[r3]
ldr r0,=1 @ load '1'
ldr r0,[r0]
stmfd sp!,{r0,r3} @ push them on the stack
ldr r0,=fmtstr @ load pointer to format string
ldr r1,=i @ load pointer to i in r1
ldr r1,[r1] @ load value of i in r1
mov r2,r6 @ copy value of j from r6 to r2
mov r3,r7 @ copy value of k from r7 to r3
bl printf @ call printf
add sp,sp,#8 @ pop 2 words from stack

```



Plan

- 1 L'exécution conditionnelle et les branchements
- 2 Accès à la mémoire, la gestion des piles
- 3 Les sous-programmes
- 4 Les appels système
- 5 Mode Thumb

Chiheb Ameer ABID

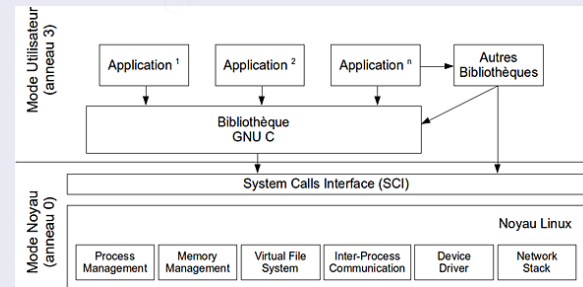


Appels système

Appels système

Appels système

- Un appel système est une fonction primitive fournie par le noyau d'un système d'exploitation
 - Suite à un appel système, le noyau prend le contrôle : l'exécution est interrompue pendant l'exécution de l'appel système
 - Le noyau s'exécute avec plus de privilèges par rapport à l'exécution d'un programme ordinaire (mode utilisateur)
- Généralement, les appels système sont accessibles à travers une API fournie par une bibliothèque telle que glibc



Appels système

Appels système

- Le système GNU/Linux offre un peu plus de 200 appels système qui sont regroupés par catégories :
 - Gestion des processus
 - Gestion des signaux
 - Gestion du système de fichiers
 - Mécanisme de protection
 - Fonctions de minuterie et statistiques



Convention d'un appel système

- La convention d'un appel système est la suivante :
 - Les registres de R0 à R6 : les données pour l'appel système
 - Le registre R7 spécifie le numéro de l'appel système
 - Exécuter l'instruction SVC 0 avec SVC : Service Call
 - Le registre R0 contient le code de retour de l'appel système
- Code de retour d'un appel système
 - ≥ 0 : exécution réussie
 - < 0 : échec de l'exécution



Plan

- 1 L'exécution conditionnelle et les branchements
- 2 Accès à la mémoire, la gestion des piles
- 3 Les sous-programmes
- 4 Les appels système
- 5 Mode Thumb

Jeu d'instructions Thumb

Jeu d'instructions Thumb

- ↳ Thumb est un jeu d'instructions codées sur 16 bits
 - ▣ Une densité de code améliorée
 - ▣ En pratique, on gagne 1/3 de l'espace mémoire
- ↳ Introduit avec l'architecture ARMv4-T
- ↳ L'accès aux registres est restreint
- ↳ Pas d'exécution conditionnelle, sauf pour les instructions de branchement
- ↳ Le bit 5 du registre d'état CPSR indique le mode

31	30	29	28	27	-	24	-	19-16	-	9	8	7	6	5	4-0
N	Z	C	V	Q		J		GE		E	A	I	F	T	M

Mode Thumb

Vue des registres en mode Thumb

- ↳ Thumb possède une vue limitée pour les registres
- ↳ Dans le mode ARM, le registre R13 est utilisé par convention comme pointeur de pile, alors que pour le mode Thumb est le registre SP est câblé matériellement comme étant un pointeur de pile

Registers	Access
r0-r7	fully accessible
r8-r12	only accessible by MOV, ADD, and CMP
r13 sp	limited accessibility
r14 lr	limited accessibility
r15 pc	limited accessibility
cpsr	only indirect access
spsr	no access

Mode Thumb

Utiliser le mode Thumb

- ↳ Initialement, un processeur ARM est au mode ARM
- ↳ Pour passer au mode Thumb, on utilise l'instruction de branchement BLX
 - ▣ BLX[cond] Rn : copie dans le registre R14 (LR) le contenu de Rn, effectue un branchement vers l'adresse contenue dans Rn, et passe en mode Thumb si le bit 0 du registre Rn est à 1
- ↳ Pour revenir au mode ARM, on utilise l'instruction de branchement BX LR
 - ▣ BX[cond] Rn : effectue un branchement vers l'adresse contenue dans Rn, et passe en mode Thumb si le bit 0 du registre Rn est à 1

```

1 blx thumb_sub @ call thumb subroutine
2 ...
3 thumb_sub:
4 ...
5 bx lr @ return from subroutine

```

- ↳ Indiquer au compilateur l'encodage des instructions sur 16 ou 32 bits, et effectuer l'alignement approprié des instructions
 - ▣ La directive .arm : code ARM
 - ▣ La directive .thumb : code Thumb

Mode Thumb

Exemple : utiliser le mode Thumb

```

1  .global main
2  .func main
3  .arm
4  main:
5      ADR R0, thumbcode+1
6      BLX R0
7  exit:
8      MOV R0, #0
9      MOV R7, #1
10     SVC 0
11
12     .thumb @ All Thumb code to be placed here
13 thumbcode:
14     MOV R3, #0
15     loop:
16         ADD R3, #1
17         SUB R0, R1
18         BGE loop
19         SUB R3, #1
20         ADD R2, R0, R1
21         BX LR @ Return to ARM

```



Mode Thumb

Exemple : utiliser le mode Thumb

↳ Désassemblage du code Thumb avec la commande `objdump -d -s`

```

00010468 <thumbcode>:
10468: 2300      movs    r3, #0
0001046a <loop>:
1046a: 3301      adds   r3, #1
1046c: 1a40      subs   r0, r0, r1
1046e: dafc     bge.n  1046a <loop>
10470: 3b01      subs   r3, #1
10472: 1842      adds   r2, r0, r1
10474: 4770     bx     lr
10476: 46c0     nop
                                ; (mov r8, r8)

```

Chaque instruction est codée sur 2 octets



Merci pour votre attention



Questions?

