



Assembleur ARM





Plan

Généralités

Spécificités de l'ARM

Jeu d'instructions

Manipulation des données

Transfert des données

Branchements

Exécution conditionnelle

Directives d'assemblage

Application



Assembleur

- L'assembleur est un langage de programmation bas niveau :
 - Équivalent au code machine,
 - Lisible par l'Homme,
- Il est spécifique à chaque processeur et lié à son architecture.



- Pour faciliter la programmation, il existe en général des *directives d'assemblage*
 - Définir des symboles, des étiquettes, tables
 - Des macros–fonctions
- Ces directives dépendent des outils utilisés (nous utiliserons celles de gcc)



Plan

Généralités

Spécificités de l'ARM

Jeu d'instructions

Manipulation des données

Transfert des données

Branchements

Exécution conditionnelle

Directives d'assemblage

Application



Caractéristiques de l'ARM

- Processeur RISC 32 bits (Reduced Instruction Set Computer) :
 - Registres internes 32 bits
 - Les instructions font 32 bits
 - Instructions load/store pour interagir la mémoire et les périphériques
 - Les données traitées font 32 bits (word), 16 bits (half-word) ou 8 bits (byte).
- Il existe un mode de programmations compact (THUMB/2) où les instructions sont sur 16/32 bits

Le mode standard 32 bit est dit mode ARM



Modèle du programmeur

- 16 registres visibles utilisables par toutes les instructions
- 3 registres avec des utilisations spécifiques :
 - r_{15} (pc) : Compteur programme
 - r_{14} (lr) : Link register (adresse de retour)
 - r_{13} (sp) : Stack pointer (pointeur de pile)

r_0
r_1
r_2
r_3
r_4
r_5
r_6
r_7
r_8
r_9
r_{10}
r_{11}
r_{12}
r_{13} (sp)
r_{14} (lr)
r_{15} (pc)



Modèle du programmeur

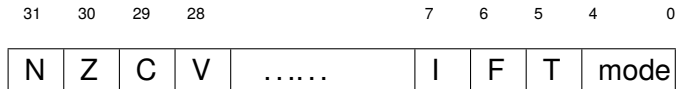
Les registres en fonction du mode

User/Syst	Supervi- sor	FIQ	IRQ	Abort	Undef
r_0	r_0	r_0	r_0	r_0	r_0
r_1	r_1	r_1	r_1	r_1	r_1
r_2	r_2	r_2	r_2	r_2	r_2
r_3	r_3	r_3	r_3	r_3	r_3
r_4	r_4	r_4	r_4	r_4	r_4
r_5	r_5	r_5	r_5	r_5	r_5
r_6	r_6	r_6	r_6	r_6	r_6
r_7	r_7	r_7	r_7	r_7	r_7
r_8	r_8	r_8fiq	r_8	r_8	r_8
r_9	r_9	r_9fiq	r_9	r_9	r_9
r_{10}	r_{10}	r_{10fiq}	r_{10}	r_{10}	r_{10}
r_{11}	r_{11}	r_{11fiq}	r_{11}	r_{11}	r_{11}
r_{12}	r_{12}	r_{12fiq}	r_{12}	r_{12}	r_{12}
$r_{13} \text{ (sp)}$	r_{13svc}	r_{13fiq}	r_{13irq}	r_{13abt}	r_{13und}
$r_{14} \text{ (lr)}$	r_{14svc}	r_{14fiq}	r_{14irq}	r_{14abt}	r_{14und}
$r_{15} \text{ (pc)}$	$r_{15} \text{ (pc)}$	$r_{15} \text{ (pc)}$	$r_{15} \text{ (pc)}$	$r_{15} \text{ (pc)}$	$r_{15} \text{ (pc)}$



Modèle du programmeur

Le registre d'état CSPR



■ Conditions :

- N : négatif
- Z : zéro
- C : retenue
- V : dépassement

■ Interruptions :

- F : désactiver les FIQ
- I : désactiver les IRQ
- T : Thumb
- Mode de fonctionnement



Plan

Généralités

Spécificités de l'ARM

Jeu d'instructions

- Manipulation des données

- Transfert des données

- Branchements

- Exécution conditionnelle

Directives d'assemblage

Application



Jeu d'instructions

On peut classer les instructions en trois grandes catégories :

- Traitement et manipulation des données :
 - Arithmétiques et logiques
 - Tests et comparaisons
- Transfert de données de et vers la mémoire
- Contrôle de flot
 - Branchements



Spécificités de l'ARM

■ Spécificités :

- Exécution conditionnelle de la majorité des instructions
- On peut combiner une instruction arithmétique et logique avec une instruction de décalage (ALU, Barrel Shifter)
- Possède des instructions pour des lectures/écritures multiples
- Possibilité d'auto incrémenter/décrémenter



Opérations arithmétiques et logiques

Opération sur 3 registres

OPE r_dest, r_s1, r_s2

Exemples

AND r_0, r_1, r_2 pour $(r_0 = r_1 \& r_2)$

ADD pc, pc, r_5 pour $(pc = pc + r_5)$



Opérations arithmétiques et logiques

Les instructions

ADD r0,r1,r2	→	$r0=r1+r2$	Addition
ADC r0,r1,r2	→	$r0=r1+r2+C$	Addition avec retenue
SUB r0,r1,r2	→	$r0=r1-r2$	Soustraction
SBC r0,r1,r2	→	$r0=r1-r2-C+1$	Soustraction avec retenue
RSB r0,r1,r2	→	$r0=r2-r1$	Soustraction inversée
RSC r0,r1,r2	→	$r0=r2-r1-C+1$	Soustraction inversée avec retenue
AND r0,r1,r2	→	$r0=r1\&r2$	Et binaire
ORR r0,r1,r2	→	$r0=r1 r2$	Ou binaire
EOR r0,r1,r2	→	$r0=r1\wedge r2$	Ou exclusif binaire
BIC r0,r1,r2	→	$r0=r1\&\sim r2$	Met à '0' les bits de r1 indiqués par r2



Opérations de mouvement de données

Opération sur 2 registres

OPE r_dest , r_s1

Exemples

MOV $r0, r1$ pour ($r_0 = r_1$)

MOV pc, lr pour ($pc = lr$)

MVN $r0, r1$ pour ($r_0 = \sim r_1$)



Opérations de mouvement de données

Les instructions

MOV r0,r1 → r0=r1 Déplacement

MVN r0,r1 → r0=~r1 Déplacement et négation



Remarque !

Modification des indicateur du CPSR

Les opérations arithmétiques et logiques ne modifient pas les indicateurs (N,Z,C,V) du CPSR.

Pour agir sur les indicateurs il faut ajouter le suffixe "S" au code de l'instruction.

Exemples

```
ADDS r0,r1,r2
```

```
ANDS r0,r1,r2
```

```
MOVS r0,r1
```



Opérations de comparaison

Opération sur 2 registres

OPE r_s1, r_s2

Exemples

CMP r0,r1 pour ($cpsr \leftarrow r_0 - r_1$)

TEQ r0,r1 pour ($cpsr \leftarrow r_0 \oplus r_1$)



Opérations de comparaison

Les instructions

CMP	r0,r1	→	cpsr $\leftarrow$ r0-r1	Comparer
CMN	r0,r1	→	cpsr $\leftarrow$ r0+r1	Comparer à l'inverse
TST	r0,r1	→	cpsr $\leftarrow$ r0&r1	Tester les bits indiqués par r1
TEQ	r0,r1	→	cpsr $\leftarrow$ r0^r1	Tester l'égalité bit à bit



Opérandes immédiats

- Un des opérandes source peut être un immédiat (une constante).
- Cette valeur se retrouve codée directement dans l'instruction.

Exemples

```
CMP r0,#0x20
```

```
ADD r0,r1,#0x1
```



Opérandes immédiats

Les instructions sont codées sur 32 bits, il ne reste que 12 bits pour coder l'immédiat.

Précautions à prendre

- Les valeurs immédiates ne sont codées que sur 8 bits ($0 \rightarrow 0xFF$)
- Un décalage pair de 0 à 30 est possible.

Exemples

ADD r0, r1, #0xFF00	($0xFF \ll 8$)
ADD r0, r1, #0x3FC	($0xFF \ll 4$)
ADD r0, r1, #0xF000000F	($0xFF \ll 28$)
ADD r0, r1, #0x102	Interdit !!



Combiner une opération avec un décalage

- Le barrel shifter peut être utilisé en même temps que l'ALU
- Toute opération peut être accompagnée du décalage du second opérande.

Exemples

ADD r0,r1,r2,LSL #4 $(r_0 = r_1 + r_2 \times 16)$

ADD r0,r1,r2,LSL r3 $(r_0 = r_1 + r_2 \times 2^{r_3})$



Combiner une opération avec un décalage

Les décalages possibles sont :

LSL	décalage logique à gauche ($\times 2$)
ASL	décalage arithmétique à gauche ($\times 2$ identique à LSL)
LSR	décalage logique à droite
ASR	décalage arithmétique à droite ($/2$ avec extension du bit de signe)
ROR	rotation à droite
RRX	rotation à droite d'un bit avec concaténation de la retenue (C)



Exercice

Écrire un programme en assembleur qui fasse le calcul suivant en utilisant le minimum d'instructions :

$$r1 = 17 * 3.5$$

Les résultat doit être l'entier le plus proche du résultat exact (59.5).

Profitez de cet exercice pour vous familiariser avec le déboguer/simulateur.

Notez l'évolution des registres et plus particulièrement du compteur programme (pc)



Les multiplications

- Les opérandes ne peuvent être des registres.
- En fonction des versions de l'architecture :
 - La retenue "C" et le dépassement "V" n'ont pas toujours le même comportement.
 - Toutes les instructions ne sont pas disponibles.



La multiplication

Les instructions

MUL r0,r1,r2	→	$r0=r*r2$	multiplication
MLA r0,r1,r2,r3	→	$r0=r1+r2*r3$	mult. et accumulation
MLS r0,r1,r2,r3	→	$r0=r1-r2*r3$	mult. soustraction
UMULL r0,r1,r2,r3	→	$\{r1,r0\}=r2*r3$	mult. 64bits non signée
SMULL r0,r1,r2,r3	→	$\{r1,r0\}=r2*r3$	mult. 64bits signée
UMLAL r0,r1,r2,r3	→	$\{r1,r0\}+=r2*r3$	MAC 64bits non signée
SMLAL r0,r1,r2,r3	→	$\{r1,r0\}+=r2*r3$	MAC 64bits signée



Instructions pour transférer les données

- Il existe deux instructions pour déplacer des données entre la RAM et le processeur
 - LDR pour charger un registre avec une donnée de 32 bits en mémoire
 - STR pour enregistrer la valeur sur 32 bits du registre en mémoire

Exemples

LDR $r_0, [r_1]$ ($r_0 = RAM[r_1]$)

STR $r_0, [r_1]$ ($RAM[r_1] = r_0$)

Les adresses doivent être alignées (multiples de 4 octets)

Il existe aussi des instructions pour transférer des mots de 8 bits (ldrb/strb) ou de 16 bits (ldrh/strh) avec l'alignement correspondant



Modes d'adressage

■ Adressage indirecte

LDR r0,[r1] ($r_0 = RAM[r_1]$)

■ Adressage indirecte avec déplacement (offset)

LDR r0,[r1,#8] ($r_0 = RAM[r_1 + 8]$)

LDR r0,[r1,r2] ($r_0 = RAM[r_1 + r_2]$)

■ Adressage indirecte avec déplacement et pré-incrémentation

LDR r0,[r1,#8]! ($r_1 = r_1 + 8$ puis $r_0 = RAM[r_1]$)

■ Adressage indirecte avec déplacement et post-incrémentation

LDR r0,[r1],#8 ($r_0 = RAM[r_1]$ puis $r_1 = r_1 + 8$)



Transferts multiples

En plus des instructions LDR et STR le jeu d'instruction ARM propose les instructions LDM et STM pour transférer plusieurs valeurs en même temps.

Exemples

LDMIA r0,{r1,r2,r3}

$(r_1 = RAM[r_0])$

$(r_2 = RAM[r_0 + 4])$

$(r_3 = RAM[r_0 + 8])$

STMIA r0,{r1-r3}

$(RAM[r_0] = r_1)$

$(RAM[r_0 + 4] = r_2)$

$(RAM[r_0 + 8] = r_3)$



Transferts multiples

Variantes

Il existe 4 suffixes possibles pour les instructions de transferts multiples :

- IA pour la post-incrémentation
- IB pour la pré-incrémentation
- DA pour la post-décrémentation
- DB pour la pré-décrémentation

Pour modifier la valeur du registre d'adresse il suffit de rajouter un !

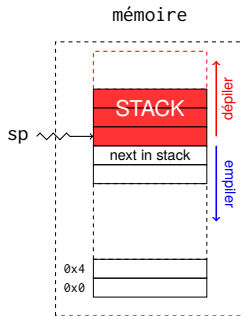
LDMIA r0!, {r1-r3}



Transferts multiples : la pile

Conventions

- Le registre **r13 (sp)** est le pointeur de pile (stack pointer)
- Le pointeur de pile contient l'adresse de la dernière donnée empilée
- A chaque empilement le pointeur de pile est décrémenté



La convention standard est donc "Full Descending"



Transferts multiples : la pile

Pour gérer la pile et éviter les confusions, il existe des équivalents des instructions LDM et STM avec des suffixes spécifiques en fonction des stratégies utilisées pour la pile.

- FD : Full Descending
- FA : Full Ascending
- ED : Empty Descending
- EA : Empty Ascending



Transferts multiples : la pile

Ou plus simplement :

Empiler

PUSH {r1-r5} ou STMFD sp!,{r1-r5}
 ou STMDB sp!,{r1-r5}

Dépiler

POP {r1-r5} ou LDMFD sp!,{r1-r5}
 ou LDMIA sp!,{r1-r5}



Synchronisation entre tâches

Il existe des instructions qui permettent la mise en œuvre de mécanismes de synchronisation de tâches (mutex, sémaphore ...) et de mémoire partagée ^{doc ARM}.

LDREX, STREX : EX pour exclusif.

SWP : Pour échanger la valeur d'un registre et d'un emplacement mémoire.

L'instruction `swp` est dépréciée pour les dernières architectures ARM surtout multiprocesseur et ne doit plus être utilisée.



Branchements

Il existe deux instructions de branchement :

B	adresse	Aller à l'adresse
BX	registre	Aller à l'adresse et éventuellement changer de mode (ARM/THUMB)

Ces instructions modifient le compteur programme "pc" (r15).

BL(X)	Pour garder l'adresse de retour dans "lr" (r14) L'adresse de retour est celle de l'instruction suivant BL.
-------	---

Il est aussi possible de modifier le registre pc en utilisant une instruction arithmétique, un mov ou un ldr mais attention aux alignements



Branchements

- pour les instructions B et BL l'adresse est stockée comme un immédiat codé sur 24 bits qui représente un offset par rapport à la position actuelle
 - $pc = pc + 8 + \text{adresse}$
 - le décalage appartient à l'intervalle $\pm 32Mo$
- pour l'instruction BX le mode est déterminé en fonction du bit de poids faible de l'adresse contenue dans le registre.
- Pour revenir d'un branchement BL il suffit de remettre 1r dans pc
`MOV pc, 1r` ou d'utiliser `BX 1r`



Exécution conditionnelle des instructions

L'exécution des instructions peut être rendue conditionnelle en rajoutant les suffixes suivant :

EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set and V set, or N clear and V clear ($N == V$)
LT	Signed less than	N set and V clear, or N clear and V set ($N != V$)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear ($Z == 0$, $N == V$)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set ($Z == 1$ or $N != V$)



Exécution conditionnelle des instructions

Exemples

CMP r_0, r_1	comparer r_0 à r_1
SUBGE r_0, r_0, r_1	si $r_0 \geq r_1$ alors $r_0 = r_0 - r_1$
SUBLT r_0, r_1, r_0	si $r_0 < r_1$ alors $r_0 = r_1 - r_0$
SUBS r_0, r_1, r_2	$r_0 = r_1 - r_2$
BEQ address	aller à adresse si le résultat est nul



Exercice

Écrivez un programme qui permet de remplir une zone mémoire de taille 0x100 octets (256) commençant à l'adresse 0x100, avec le motif 0xdeadbeef.

Déplacez ensuite toutes ces données vers une zone mémoire commençant à l'adresse 0x300.



Plan

Généralités

Spécificités de l'ARM

Jeu d'instructions

Manipulation des données

Transfert des données

Branchements

Exécution conditionnelle

Directives d'assemblage

Application



Syntaxe de l'assembleur GNU pour ARM

- Nous utiliserons gnu arm as (arm-none-eabi-as est installé dans les salles de TP)
- En plus du code assembleur, nous pouvons utiliser des directives pour obtenir un code plus lisible/facile à écrire.
- La doc <http://sourceware.org/binutils/docs/as>



Syntaxe de l'assembleur GNU pour ARM

- La forme générale des instructions est alors :

`[<Étiquette>:] [<instruction ou directive>] [@ <commentaire>]`



Syntaxe de l'assembleur GNU pour ARM

- Les étiquettes (labels) seront remplacées par l'adresse de l'instruction qui suit.
- Les lignes ne contenant que des commentaires ou étiquettes ne sont pas comptées.



Syntaxe de l'assembleur GNU pour ARM

Exemple

Start:

```
MOV r0,#0 @ mise zero de r0
```

```
MOV r2,#10 @ charger la valeur 10 dans r2
```

Loop:

```
ADD r0,r0,r2,LSL #1 @ r0=r0+2*r2
```

```
SUBS r2,r2,#1 @ r2--
```

```
BNE Loop
```

```
B Start
```



Quelques directives d'assemblage utiles

LDR r0,=VALEUR

- Cette directive permet de mettre une valeur quelconque dans un registre. Cette directive est remplacée en fonction de la valeur par :

- MOV r0,#VALEUR
- LDR r0, [pc,#offset]

...

.word VALEUR

Où offset est le décalage entre l'adresse de l'instruction et l'adresse où est positionnée la valeur (à la fin du code).



Quelques directives d'assemblage utiles

.EQU SYMBOLE, VALEUR ou

.SET SYMBOLE, VALEUR

- La macro est remplacée par la valeur.

Exemple :

```
.EQU COMPTEUR,10
```

```
...
```

```
MOV r0, #COMPTEUR
```



Exercice

Reprendre l'exercice précédent en utilisant des directives d'assemblage



Quelques directives d'assemblage utiles

Directives de remplissage :

- mettre une valeur sur 32 bits/16 bits/8 bits à la position actuelle :
`.word/.half/.byte VALEUR`
- mettre une chaîne de caractères :
`.ascii "La chaine de caracteres"`
`.asciz "Une autre chaine" se finit par '\0'`
- remplir une zone :
`.fill nombre, taille_en_octets, valeur`
Exemple : `.fill 100,4,0xdeadbeaf`



Quelques directives d'assemblage utiles

Récupérer l'adresse d'une étiquette :

```
ADR r0,ETIQUETTE
```

- Cette directive est remplacée par :

```
ADD r0,pc,#offset
```

Où offset est le décalage entre la position de l'instruction et la position de l'étiquette.

Exemple :

```
ADR r0, str
```

```
str:
```

```
.asciz "hello world"
```



Quelques directives d'assemblage utiles

Directives d'alignement :

- Si les données insérées ne sont pas multiple de la taille d'une instruction il faut réaligner en remplissant éventuellement les vides par un motif.

```
.balign nbr_octets, motif
```

```
.align log2(nbr_octets), motif
```



Quelques directives d'assemblage utiles

Définition de symboles

- Pour utiliser un symbole défini ailleurs, ou pour déclarer un symbole qui sera utilisé ailleurs :

```
.global NOM_DU_SYMBOLE
```

Préciser l'architecture du processeur utilisé (directive spécifique ARM)

- On peut préciser l'architecture du processeur ARM utilisé

```
.arch NOM
```

par exemple : `.arch arm7tdmi`



Plan

Généralités

Spécificités de l'ARM

Jeu d'instructions

Manipulation des données

Transfert des données

Branchements

Exécution conditionnelle

Directives d'assemblage

Application



Faire clignoter les leds de la maquette