



IP PARIS

# Langage assembleur

Exemple de l'assembleur ARM

Tarik Graba

Année scolaire 2023/2024



# Plan

## Généralités

## Spécificités de l'architecture ARM

## Jeu d'instructions

- Manipulation des données

- Transfert des données

- Branchements

- Exécution conditionnelle

## Encodage

## Directives d'assemblage

- Un processeur interprète des instructions “numériques” (généralement codées en binaire),
  - C’est ce qui est appelé *langage machine*.
- L’assembleur est un langage de programmation bas niveau,
  - lisible par l’Homme (représentation textuelle),
  - équivalent au langage machine (1 ↔ 1).
- Il est spécifique à chaque processeur et lié à son *architecture*.

- Pour faciliter la programmation, en plus des instructions, les outils permettent de :
  - Définir des symboles, des étiquettes, tables
  - Des macros–fonctions
- On appelle cela des ***directives d'assemblage***.
- Ces directives varient en fonction des outils utilisés (*chaîne de compilation*).
- *Nous utiliserons dans ce cours celles de Gnu AS.*



# Plan

Généralités

Spécificités de l'architecture ARM

Jeu d'instructions

- Manipulation des données

- Transfert des données

- Branchements

- Exécution conditionnelle

Encodage

Directives d'assemblage

# Caractéristiques de l'ARM

## Taille des données

### ■ Architecture LOAD/STORE

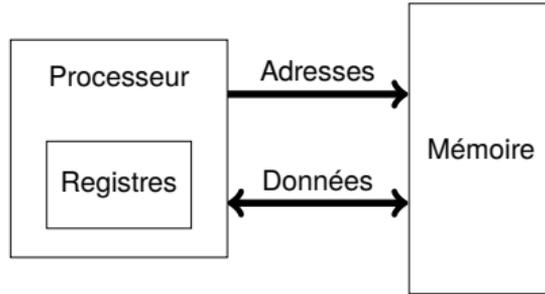
- Dans l'esprit RISC (Reduced Instruction Set Computer)

### ■ Processeur 32 bits :

- Les registres généraux font 32 bits
- Les données manipulées sont des word (32 bits), half-word (16 bits) ou byte (8 bits).

*Il existe aussi des variantes 64 bits des processeurs ARM (architecture ARMv8-A) que nous n'abordons pas dans ce cours*

## Modèle du programmeur



- Espace mémoire unifié extérieur au processeur.
  - L'espace mémoire est commun aux instructions et aux données.
  - Les *périphériques* font aussi partie de l'espace mémoire.
  - adressable par octet
- Des registres internes au processeur.
- Instructions pour le transfert entre les registres et la mémoire.
- Opérations sur le contenu des registres.

# Caractéristiques de l'ARM

## Différentes tailles pour les instructions

Historiquement les instructions faisaient toutes 32 bits.

Pour rendre le code plus compact et permettre des réductions de la taille du code et donc des coûts, des jeux d'instructions compacts ont été ajoutés :

- Les instructions font :
  - 32 bits (mode ARM)
  - 16 bits (mode THUMB)
  - mixte 32/16 bits (mode THUMB 2)

Ces variantes compactes favorisent les instructions utilisées souvent. Les instructions moins courantes sont remplacées par une séquence d'instructions.

# Modèle du programmeur

## Les registres généraux

Un programmeur a accès à :

- 16 registres généraux
- dont 3 avec des rôles particuliers :
  - $r_{15}$  (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

## En fonction de l'état du processeur

Les “shadow registers” de l'ARM7TDMI (ARMv4T) :

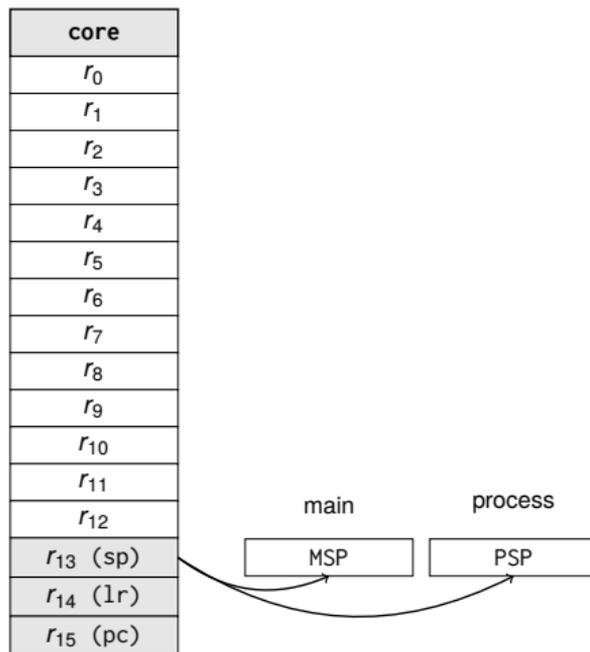
User/Syst.	Superv.	FIQ	IRQ	Abort	Undef
$r_0$	$r_0$	$r_0$	$r_0$	$r_0$	$r_0$
$r_1$	$r_1$	$r_1$	$r_1$	$r_1$	$r_1$
$r_2$	$r_2$	$r_2$	$r_2$	$r_2$	$r_2$
$r_3$	$r_3$	$r_3$	$r_3$	$r_3$	$r_3$
$r_4$	$r_4$	$r_4$	$r_4$	$r_4$	$r_4$
$r_5$	$r_5$	$r_5$	$r_5$	$r_5$	$r_5$
$r_6$	$r_6$	$r_6$	$r_6$	$r_6$	$r_6$
$r_7$	$r_7$	$r_7$	$r_7$	$r_7$	$r_7$
$r_8$	$r_8$	$r_{8fiq}$	$r_8$	$r_8$	$r_8$
$r_9$	$r_9$	$r_{9fiq}$	$r_9$	$r_9$	$r_9$
$r_{10}$	$r_{10}$	$r_{10fiq}$	$r_{10}$	$r_{10}$	$r_{10}$
$r_{11}$	$r_{11}$	$r_{11fiq}$	$r_{11}$	$r_{11}$	$r_{11}$
$r_{12}$	$r_{12}$	$r_{12fiq}$	$r_{12}$	$r_{12}$	$r_{12}$
$r_{13}$ (sp)	$r_{13svc}$	$r_{13fiq}$	$r_{13irq}$	$r_{13abt}$	$r_{13und}$
$r_{14}$ (lr)	$r_{14svc}$	$r_{14fiq}$	$r_{14irq}$	$r_{14abt}$	$r_{14und}$
$r_{15}$ (pc)					

# Modèle du programmeur

## En fonction de l'état du processeur

Les deux pointeurs de pile des Cortex-M (ARMv6-M) :

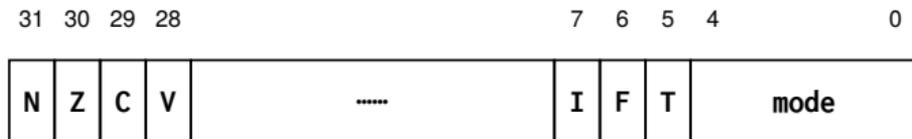
- Main Stack Pointer (MSP)
- Process Stack Pointer (PSP)



# Modèle du programmeur

## Le registre d'état xPSR

CPSR : Current Program Status Register de l'ARM7TDMI



### ■ Flags :

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

### ■ Interruptions :

- F : désactiver les FIQ
- I : désactiver les IRQ

### ■ T : Thumb

### ■ Mode de fonctionnement





# Modèle du programmeur

## Le registre d'état xPSR

Les registres de statu est un registre particulier accessible par des instructions spéciales.



## Plan

Généralités

Spécificités de l'architecture ARM

**Jeu d'instructions**

- Manipulation des données

- Transfert des données

- Branchements

- Exécution conditionnelle

Encodage

Directives d'assemblage

### Instructions ARM/Thumb

- À l'origine les syntaxes des instructions **thumb** et **arm** étaient différentes.
- Une syntaxe unifiée (**unified**) est supportée par les versions récentes des outils de développement.
- Du fait de la taille des instructions **thumb** (16-bits) certaines restrictions d'usage existent.

Dans ce cours nous présentons la syntaxe unifiée. Pour les restrictions du mode thumb se référer à la documentation.

On peut classer les instructions en trois grandes catégories :

1. Traitement et manipulation des données :
  - Arithmétiques et logiques
  - Tests et comparaisons
2. Transfert de données depuis et vers la mémoire
3. Contrôle de flot
  - Branchements



# Plan

Généralités

Spécificités de l'architecture ARM

**Jeu d'instructions**

**Manipulation des données**

Transfert des données

Branchements

Exécution conditionnelle

Encodage

Directives d'assemblage

## Opérations arithmétiques et logiques

### Opération sur 3 registres

OPE r\_dest, r\_s1, r\_s2

### Exemples

AND r0, r1, r2 pour ( $r_0 = r_1 \& r_2$ )

ADD r5, r1, r5 pour ( $r_5 = r_1 + r_5$ )

# 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^r2	Ou exclusif binaire
BIC r0,r1,r2	→ r0=r1&~r2	Met à 0 les bits de r1 indiqués par r2

# Opérations de déplacement de données entre registres

## 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 déplacement de données entre registres

### Les instructions

MOV r0,r1 → r0=r1 Déplacement

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

## Opérations de décalage

### Opération sur 3 registres

OPE r\_dest, r\_s, r\_m

### Exemples

LSL r0, r1, r2 pour ( $r_0 = r_1 \ll r_2[7:0]$ )

ASR r3, r4, r5 pour ( $r_3 = r_4 \gg r_5[7:0]$ )

Seul l'octet de poids faible de r\_m est utilisé.

## Opérations de décalage

### Les instructions

LSL	→	Décalage logique vers la gauche
LSR	→	Décalage logique vers la droite
ASR	→	Décalage arithmétique vers la droite
ROR	→	Décalage circulaire vers la droite

## Remarque !

### Modification des indicateurs du PSR

Par défaut, les opérations arithmétiques et logiques ne modifient pas les indicateurs (*flags*) (N,Z,C,V) du PSR.

Il faut ajouter le suffixe "S" au mnémonique de l'instruction si nécessaire.

### 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 ( $psr \leftarrow r_0 - r_1$ )

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

## Opérations de comparaison

### Les instructions

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

Ces instructions ne modifient que les bits (N,Z,C,V) du PSR, le résultat n'est pas gardé.

## Opérandes immédiats

Un immédiat est une valeur constante qui sera encodée **dans l'instruction** elle même.

- Un seul des opérandes source peut être un immédiat.
- On doit les précéder du symbole '#'.  
■ Il peut être décimal, hexadécimal (0x) ou octal (0).

### Exemples

```
MOV r0,#0x20
```

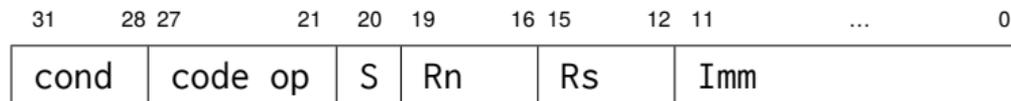
```
CMP r0,#32
```

```
ADD r0,r1,#1
```

## Opérandes immédiats

### Comment est-ce encodé ?

En mode ARM les instructions sont codées sur 32 bits. Pour la majorité des instructions il reste 12 bits qui peuvent être utilisés pour coder l'immédiat.



Ces 12 bits sont utilisés de la façon suivante :

- 8 bits (0 → 0xFF)
- 4 bits pour un décalage circulaire (valeurs paires de 0 à 30)

## Opérandes immédiats

### Exemples

ADD r0,r1,#100	(0x64 $\ll$ 0)
ADD r0,r1,#0xFF00	(0xFF $\ll$ 8)
ADD r0,r1,#0x3FC	(0xFF $\ll$ 2)
ADD r0,r1,#0xF000000F	(0xFF $\ll$ 28)
ADD r0,r1,#0x102	Interdit !!

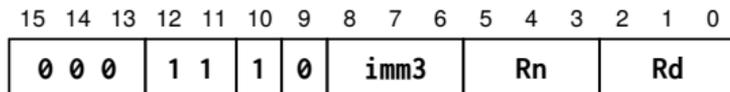
Nous verrons par la suite que ce n'est pas vraiment un problème.

## Opérandes immédiats

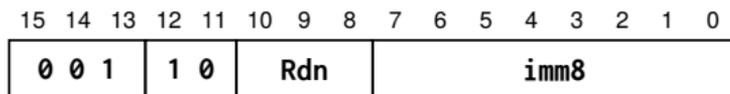
### En mode thumb-16bits

Ça dépend de l'instruction. Par exemple :

- ADDS Rd,Rn,#imm3 (source et destination différentes)



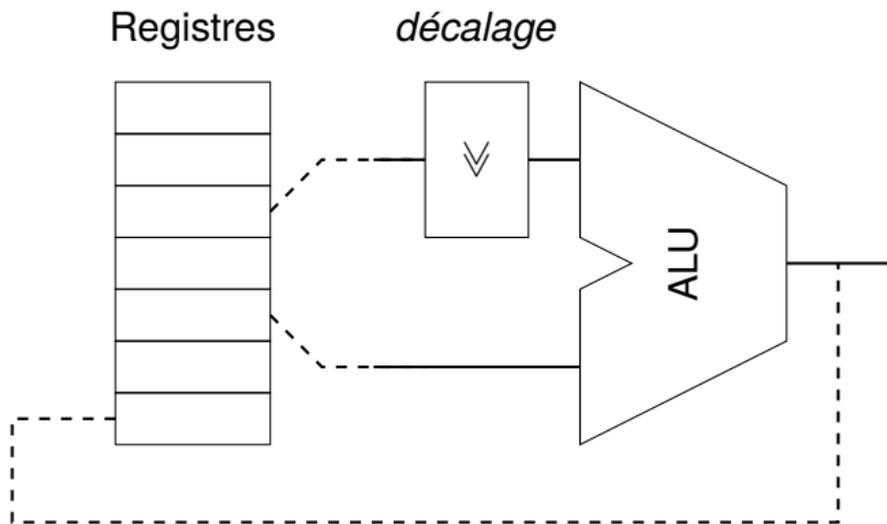
- ADDS Rdn,#imm8 (même source et destination)



## Combiner une opération avec un décalage

### ALU + Barrel shifter

Les processeur ARM ont la particularité d'avoir un *barrel shifter* devant une des entrées de l'ALU.



## 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

### Remarques

Certaines instructions sont équivalentes :

`LSL rd,rs,#i`  $\Leftrightarrow$  `MOV rd,rs,LSL #i`

Rotate Right with Extend

`RRX rd,rs`  $\Leftrightarrow$  `MOV rd,rs,RRX`

Rotation à droite **d'une position** en prenant le bit de retenue C.

Les opérations équivalentes seront encodées de la même façon.

## Les multiplications

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

## 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



# Plan

Généralités

Spécificités de l'architecture ARM

## Jeu d'instructions

Manipulation des données

**Transfert des données**

Branchements

Exécution conditionnelle

Encodage

Directives d'assemblage

## Instructions pour transférer les données

Deux instructions de transfert de données entre la mémoire et les registres.

- LDR : charger un registre avec une donnée en mémoire
- STR : enregistrer la valeur du registre en mémoire

L'adresse doit être dans un registre.

### Exemples

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

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

# Instructions pour transférer les données

## Taille des données et alignement

- LDR/STR : mots de 32 bits (words)
- LDRH/STRH : mots de 16 bits (half words)
- LDRB/STRB : mots de 8 bits (byte)

Généralement, les adresses **doivent** être alignées :

- LDR/STR : adresse multiple de 4
- LDRH/STRH : adresse multiple de 2
- LDRB/STRB : adresse quelconque

## Modes d'adressage

- Adressage indirect

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

- Adressage indirect 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 indirect avec déplacement et pré-incrémentation

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

- Adressage indirect 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 les transferts multiples.

### 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}	( $RAM[r_0] = r_1$ )
	( $RAM[r_0 + 4] = r_2$ )
	( $RAM[r_0 + 8] = r_3$ )

### Variantes

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

- IA pour la post-incrémentation (*Increment After*)
- IB pour la pré-incrémentation (*Increment Before*)
- DA pour la post-décrémentation (*Decrement After*)
- DB pour la pré-décrémentation (*Decrement Before*)

Pour que la valeur du registre d'adresse soit modifiée il faut ajouter (!)

```
LDMIA r0!,{r1-r3}
```



## Transferts multiples

### Alias pour les piles (stacks)

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





# Plan

Généralités

Spécificités de l'architecture ARM

**Jeu d'instructions**

Manipulation des données

Transfert des données

**Branchements**

Exécution conditionnelle

Encodage

Directives d'assemblage

Il existe deux instructions de branchement :

- B adresse            Aller à l'adresse
- BX registre        Aller à l'adresse pointée par le registre  
et éventuellement changer de mode  
(ARM/THUMB interworking)

## Branchements

### Appel de sous-programme

Les instructions de branchement modifient le compteur programme "pc" (r15)

BL(X)      Sauvegarde l'adresse de retour dans "lr" (r14)

L'adresse de retour est celle de l'instruction suivant le BL.

Pour revenir d'un branchement BL il suffit de remettre lr dans pc

- BX lr
- MOV pc,lr (deprecated)

## Branchements

- pour les instructions B et BL l'adresse est stockée comme un immédiat qui représente un décalage (*offset*) par rapport à la position actuelle :
  - l'offset est forcément limité
  - le branchement ne peut aller très loin
- L'instruction BX permet de changer de mode (ARM/Thumb) en fonction du **bit de poids** faible de l'adresse normalement non utilisé (*voir interworking*)



# Plan

Généralités

Spécificités de l'architecture ARM

**Jeu d'instructions**

Manipulation des données

Transfert des données

Branchements

Exécution conditionnelle

Encodage

Directives d'assemblage

## Exécution conditionnelle des instructions

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

EQ	Equal	$Z == 1$
NE	Not equal	$Z == 0$
CS/HS	Carry set/unsigned higher or same	$C == 1$
CC/LO	Carry clear/unsigned lower	$C == 0$
MI	Minus/negative	$N == 1$
PL	Plus/positive or zero	$N == 0$
VS	Overflow	$V == 1$
VC	No overflow	$V == 0$
HI	Unsigned higher	$C == 1$ and $Z == 0$
LS	Unsigned lower or same	$C == 0$ or $Z == 1$
GE	Signed greater than or equal	$N == V$
LT	Signed less than	$N != V$
GT	Signed greater than	$Z == 0$ and $N == V$
LE	Signed less than or equal	$Z == 1$ 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 $r_0 = r_1 - r_0$
<hr/>	
SUBS $r_0, r_1, r_2$	$r_0 = r_1 - r_2$
BEQ address	aller à adresse si le résultat est nul



## Plan

Généralités

Spécificités de l'architecture ARM

Jeu d'instructions

- Manipulation des données

- Transfert des données

- Branchements

- Exécution conditionnelle

Encodage

Directives d'assemblage

# Encodage des instructions

## En mode ARM

	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
Data processing and FSR transfer	Cond	0	0	1	Opcode			S	Rn			Rd			Operand 2																	
Multiply	Cond	0	0	0	0	0	0	0	A	S	Rd			Rn			Rs		1	0	0	1	Rm									
Multiply long	Cond	0	0	0	0	1	U	A	S	RdHi			RdLo			Rn			1	0	0	1	Rm									
Single data swap	Cond	0	0	0	1	0	B	0	0	Rn			Rd			0	0	0	0	1	0	0	1	Rm								
Branch and exchange	Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				
Halfword data transfer, register offset	Cond	0	0	0	P	U	0	W	L	Rn			Rd			0	0	0	0	1	S	H	1	Rm								
Halfword data transfer, immediate offset	Cond	0	0	0	P	U	1	W	L	Rn			Rd			Offset			1	S	H	1	Offset									
Single data transfer	Cond	0	1	1	P	U	B	W	L	Rn			Rd			Offset																
Undefined	Cond	0	1	1																							1					
Block data transfer	Cond	1	0	0	P	U	S	W	L	Rn			Register list																			
Branch	Cond	1	0	1	L	Offset																										
Coprocessor data transfer	Cond	1	1	0	P	U	N	W	L	Rn			CRd			CP#			Offset													
Coprocessor data operation	Cond	1	1	1	0	CP Opc			CRn			CRd			CP#			CP	0	CRm												
Coprocessor register transfer	Cond	1	1	1	0	CP Opc			L	CRn			Rd			CP#			CP	1	CRm											
Software interrupt	Cond	1	1	1	1	Ignored by processor																										

# Encodage des instructions

## En mode thumb

	Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move shifted register	01	0	0	0	Op	Offset5					Rs	Rd					
Add and subtract	02	0	0	0	1	1	Op	Rn/ offset3		Rs	Rd						
Move, compare, add, and subtract immediate	03	0	0	1	Op	Rd	Offset8										
ALU operation	04	0	1	0	0	0	0	Op	Rs	Rd							
High register operations and branch exchange	05	0	1	0	0	0	1	Op	H1/H2	Rs/Hs	Rd/Hd						
PC-relative load	06	0	1	0	0	1	Rd	Word8									
Load and store with relative offset	07	0	1	0	1	L	B	0	Ro	Rb	Rd						
Load and store sign-extended byte and halfword	08	0	1	0	1	H	S	1	Ro	Rb	Rd						
Load and store with immediate offset	09	0	1	1	B	L	Offset5			Rb	Rd						
Load and store halfword	10	1	0	0	0	L	Offset5			Rb	Rd						
SP-relative load and store	11	1	0	0	1	L	Rd	Word8									
Load address	12	1	0	1	0	SP	Rd	Word8									
Add offset to stack pointer	13	1	0	1	1	0	0	0	0	S	SWord7						
Push and pop registers	14	1	0	1	1	L	1	0	R	Rlist							
Multiple load and store	15	1	1	0	0	L	Rb	Rlist									
Conditional branch	16	1	1	0	1	Cond				Softset8							
Software interrupt	17	1	1	0	1	1	1	1	1	Value8							
Unconditional branch	18	1	1	1	0	0	Offset11										
Long branch with link	19	1	1	1	1	H	Offset										



## Plan

Généralités

Spécificités de l'architecture ARM

Jeu d'instructions

- Manipulation des données

- Transfert des données

- Branchements

- Exécution conditionnelle

Encodage

Directives d'assemblage

## Directives d'assemblage

- Les directives d'assemblage sont des indications supplémentaires pour aider le programmeur.
  - Elles ne correspondent pas à des instructions.
  - Elles permettent de manipuler simplement des adresses ou des données.
  - Elles sont interprétées par l'outil et dépendent de l'outil utilisé.
- En plus des directives, les outils supportent des pseudo-instructions et macros.
  - Elles permettent de simplifier l'écriture du code.
  - Elles sont transformées en instructions, éventuellement différentes en fonction du contexte.
  - La correspondance n'est pas forcément 1 ↔ 1

# Syntaxe de l'assembleur GNU pour ARM

- Nous utiliserons *Gnu Arm As*
  - `arm-none-eabi-as` est installé dans les salles de TP
- La documentation officielle :
  - <http://sourceware.org/binutils/docs/as>
- Convention pour l'extension des fichiers :
  - `.s` : contient assembleur et directives.
  - `.S` : peut contenir des macros préprocesseur C.

## Syntaxe de l'assembleur GNU pour ARM

- La forme générale des instructions est alors :

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

---

as supporte aussi les commentaires C (`/* ... */`).

## Syntaxe de l'assembleur GNU pour ARM

- Les lignes ne contenant que des commentaires ou étiquettes ne sont pas comptées.
- Les étiquettes (labels) seront remplacées par l'adresse de l'instruction qui suit.
- Un “*symbole*” local ayant le nom de l'étiquette est défini.

# 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

syntaxe, cible, mode

**.syntax** unified

Pour préciser qu'on utilise la syntaxe unifiée

**.cpu** cpu\_model

Pour préciser le modèle du processeur (arm7tdmi, cortex-m4, cortex-m0 ...)

**.arch** cpu\_arch

Pour préciser l'architecture du processeur (armv5t, armv7-m, armv6-m ...)

**.thumb/.arm**

Pour préciser qu'on veut générer du code ARM ou THUMB/THUMB2

## Quelques directives d'assemblage utiles

### Des macros

**.EQU** SYMBOLE, VALEUR ou

**.SET** SYMBOLE, VALEUR

- La macro est remplacée par la valeur.

### Exemple :

```
.EQU COMPTEUR,10
```

```
...
```

```
MOV r0, #COMPTEUR
```

des macros plus complexes sont aussi possibles.

## Quelques directives d'assemblage utiles

### Des pseudo-instructions

**LDR** r0,=IMMEDIAT

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

- MOV r0,#IMMEDIAT
- LDR r0, [pc,#offset]
- ...
- .word IMMEDIAT

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

### Des pseudo-instructions

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.

**LDR** r0, =ETIQUETTE

- Cette directive réservera un espace pour stocker une adresse de 32 bits et sera remplacée par un chargement.

Exemple :

```
ADR r0, str
```

```
str:
```

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



## Quelques directives d'assemblage utiles

### Gérer les symboles

Définition de symboles globaux

- Pour déclarer un symbole qui sera utilisé ailleurs :

```
.global NOM_DU_SYMBOLE
```

Le symbole est *exporté* et sera visible au moment de l'édition de liens comme une fonction en C par exemple.

# Quelques directives d'assemblage utiles

## Gérer les sections

`.section` nom, "flags", %type

Permet de préciser la section dans l'objet final

Exemples de flags :

"a" : allouer de l'espace,

"x" : contient du code exécutable,

"w" : est modifiable.

Exemples de types :

`%progbits` : contient des données

`%nobits` : ne contient rien, il faut juste allouer l'espace nécessaire

Par défaut, si la directive n'est pas utilisée, le code assemblé se retrouve dans la section `.text`

Équivalent de `.section .text, "ax", %progbits`

## Quelques directives d'assemblage utiles

### Des données verbatim

Directives de remplissage (dans le code) :

- mettre une valeur arbitraire 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

### Des contraintes

Directives d'alignement :

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

`.align log2(nbr_octets), motif`

`.balign nbr_octets, motif`



## Exercice

Pour préparer le TD

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

À la fin du programme, on doit empêcher le programme de continuer.

Ajouter du code pour déplacer ces données vers une zone mémoire commençant à l'adresse  $0x300$ .