



Programmation assembleur ARM

Dr. Eng. Chiheb Ameer ABID

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

Assembleur

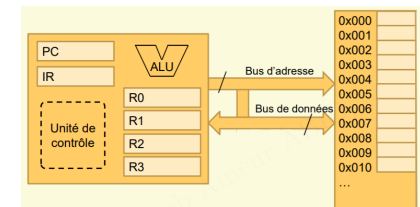
Assembleur, c'est quoi ?

- Un processeur interprète des instructions "numériques" généralement codées en binaires
 - ☞ Langage machine
- L'assembleur est un langage de programmation bas niveau
 - ☞ Lisible par l'homme
 - ☞ Équivalent au langage machine
- Il est spécifique à chaque processeur et lié à son architecture
 - ☞ Un processeur reconnaît un nombre limité d'instructions
 - ☞ L'ensemble d'instructions reconnu forme ce qu'on appelle le jeu d'instructions du processeur

Plan

- 1 Introduction
- 2 Historique
- 3 Les différentes architectures du processeur ARM
- 4 Architecture du processeur ARM 32 bits
- 5 Structure d'un programme assembleur, compilation et débogage
- 6 Directives
- 7 Les opérations de base de l'assembleur ARM

Assembleur

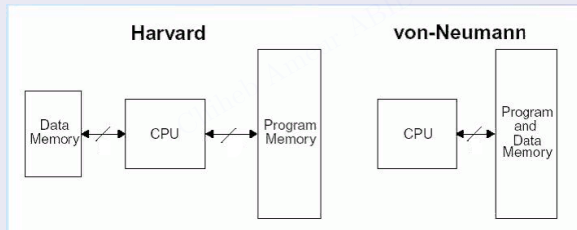


Les instructions d'un processeur

- Les instructions processeurs peuvent être classées en 3 groupes
 - 1 Le chargement de données de l'extérieur ou une mémoire vers un bloc de registres
 - 2 La transformation des données grâce à une unité arithmétique et logique
 - 3 L'entreposage de données en mémoire ou leur transfert vers l'extérieur

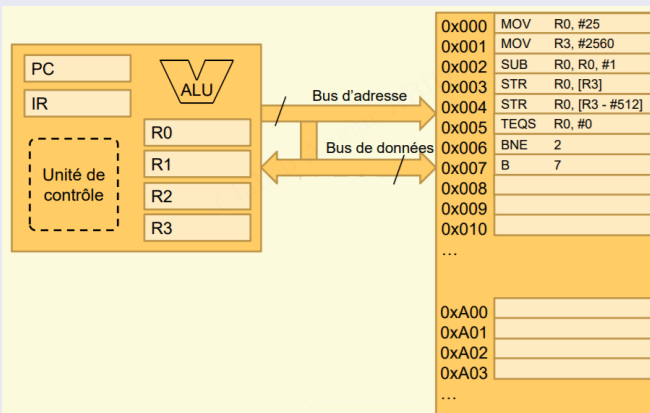
Fonctionnement des processeurs

Architectures des processeurs



Fonctionnement des processeurs

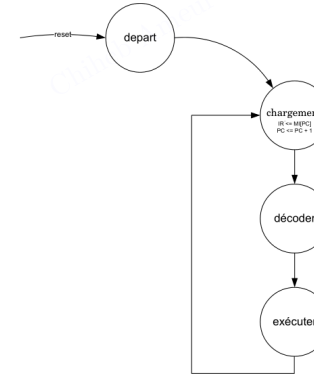
Cycle d'exécution d'une instruction



Cycle d'exécution d'une instruction

➤ L'exécution d'une instruction se fait en 3 étapes

- 1 **Chargement (fetch) :** charger l'instruction à partir de la mémoire ayant l'adresse C0 dans le registre d'instruction RI, puis incrémenter C0
- 2 **Décodage (decode) :** décoder l'instruction en identifiant l'instruction et ses opérandes
- 3 **Exécution (execute) :** réaliser une opération arithmétique ou logique, transférer des données, effectuer un branchement.

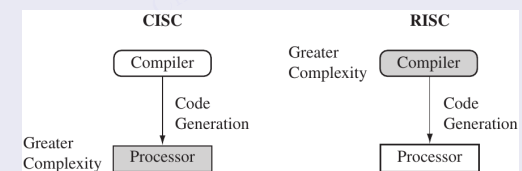


Historique

Processeurs RISC

➤ Un processeur RISC se base sur les concepts suivants :

- 1 **Jeu d'instructions réduit**
- 2 **Exécution en pipeline**
- 3 **Un grand nombre de registres à usage général**
- 4 **Architecture LOAD/STORE**



Pourquoi apprendre l'assembleur ?

Pourquoi apprendre l'assembleur ?

- Les premières étapes du boot
- Coder pour gérer les interruptions
- Coder pour des machines ne possédant pas de compilateur
- Travailler sur des machines ayant des ressources limitées
- Optimiser le code
- Écrire le code pour les pilotes

Plan

- 1 Introduction
- 2 Historique
- 3 Les différentes architectures du processeur ARM
- 4 Architecture du processeur ARM 32 bits
- 5 Structure d'un programme assembleur, compilation et débogage
- 6 Directives
- 7 Les opérations de base de l'assembleur ARM

Historique

L'histoire d'ARM

- Tandis qu'Intel produit des microprocesseurs 8086 et 80286, des employés d'une compagnie anglaise se sont inspirés par les travaux de l'heure travaillent à la conception d'un ordinateur
- En 1985, la compagnie ACORN produit un premier processeur qui résulte de ces travaux : le Acorn RISC Machine (ARM)
- La complexité du ARM est réduite par choix et par faute de moyens, ainsi le jeu d'instructions est simplifié : naissance des processeurs RISC
 - Acorn Archimedes (1987) : ordinateur conçu à base d'ARM
- Alors que le 8086 gagne en popularité, ACORN et Apple collaborent pour fonder la compagnie ARM ltd en 1990
- ARM et Intel feront évoluer leurs produits en parallèle dans des marchés différents

La philosophie d'ARM

La philosophie d'ARM

- ARM Holdings
 - Développe des architectures de micro-processeurs et des jeux d'instructions
 - Ne construit aucun micro-processeur comme tel!
 - La compagnie licencie la technologie à d'autres qui les fabriquent à leur façon clients : Apple, Nvidia, Samsung, Texas Instruments, etc.
- Micro-processeurs ARM
 - Supportent 32 et 64 bits
 - L'architecture la plus utilisée au monde (100 milliards de processeurs produits en 2017)

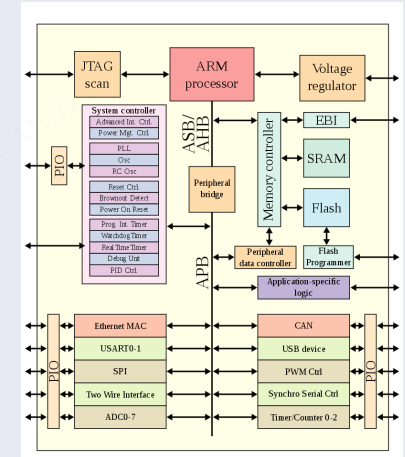
ARM est partout

- Plusieurs versions de processeurs, utilisées partout !
- ☞ ARM7TDMI(-S) : Nintendo DS, Lego NXT
- ☞ ARM946E-S : Canon 5D Mark ii (caméra)
- ☞ ARM1176JZ(F)-S : Raspberry Pi
- ☞ Cortex-A9 : Apple iPhone 4S, iPad2
- ☞ Cortex-A15 : Nexus 10



ARM est partout

- ☞ Aujourd'hui, ARM est surtout connu pour ses systèmes sur puce (SoC), intégrant sur une seule puce :
 - microprocesseur,
 - processeur graphique (GPU),
 - DSP,
 - FPU,
 - SIMD,
 - et contrôleur de périphériques.
 Ceux-ci sont présents dans la majorité des smartphones et tablettes.
- ☞ ARM propose des architectures, qui sont vendues sous licence de propriété intellectuelle aux concepteurs.
- ☞ Différentes options sont possibles ainsi les constructeurs peuvent prendre ce qui les intéresse pour les compléter avec leurs options propres ou de concepteurs tiers



Architecture des ARMs 32 bits

- Architecture RISC pour laquelle tout passe par des registres
- ARM possède 37 registres de 32 bits (seulement 17 sont visibles)
 - ☞ 16 registres généraux
 - ☞ le compteur ordinal R15 (PC) de 32 bits
 - ☞ le registre d'état CPSR de 32 bits
- Chaque instruction est codée sur 32 bits
- Accès à 2^{32} octets = 4GB de mémoire
- Organisation des données dans la mémoire "Little ou Big endian au choix"
- Opère sur 8, 16 et 32 bits



Plan

- 1 Introduction
- 2 Historique
- 3 Les différentes architectures du processeur ARM
- 4 Architecture du processeur ARM 32 bits
- 5 Structure d'un programme assembleur, compilation et débogage
- 6 Directives
- 7 Les opérations de base de l'assembleur ARM



Versions et implémentation de v1 à v6

Versions et implémentation

- Version 1 : ARM1
 - ☞ Pas vraiment commercialisée (quelques centaines d'exemplaires)
- Version 2 : ARM2
 - ☞ 27 registres (16 accessibles simultanément)
 - ☞ 4 modes de fonctionnement (mode utilisateur avec certaines ressources non disponibles, mode interruption pour gérer les interruptions externes, mode interruption rapide avec plus de ressources dédiées, mode superviseur pour l'exécution du système d'exploitation)
 - ☞ Pipeline d'exécution à trois étages (lecture, décodage, exécution)
 - ☞ 8MHz, 4 à 5 MIPS
- Version 2aS : ARM250 et ARM3
 - ☞ Ajout d'un cache unifié (données et instructions) de 4Ko
 - ☞ Ajout d'une instruction d'échange de données monolithique et atomique entre un registre et la mémoire (environnement multiproc)
 - ☞ Version mise en œuvre dans l'ARM3 (26 à 33MHz) et ARM250 (12MHz)



Versions et implémentation de v1 à v6

Versions et implémentation de v1 à v6

- Version 5 : ARM10, Xscale
 - ☞ V4 + instructions supplémentaires, unités d'exécution multiples, pipeline à 6 étages
 - ☞ ARM10 : 2*32Ko de cache, gestionnaire de mémoire compatible avec les OS embarqués, 4000MIPS, 350 MHz
 - ☞ Introduction de Jazelle DBX : technique permettant d'exécuter le bytecode Java
 - ☞ Intel Xscale : gestion de plusieurs périphériques, jusqu'à 624MHz
- Version 6 : ARM11
 - ☞ Amélioration multimédia
 - ☞ Extension SIMD (Single Instruction Multiple Data stream)
 - ☞ Support multiprocesseurs
 - ☞ Pipeline 8 étages, unité de prédiction de branchement.



Versions et implémentation de v1 à v6

Versions et implémentation de v1 à v6

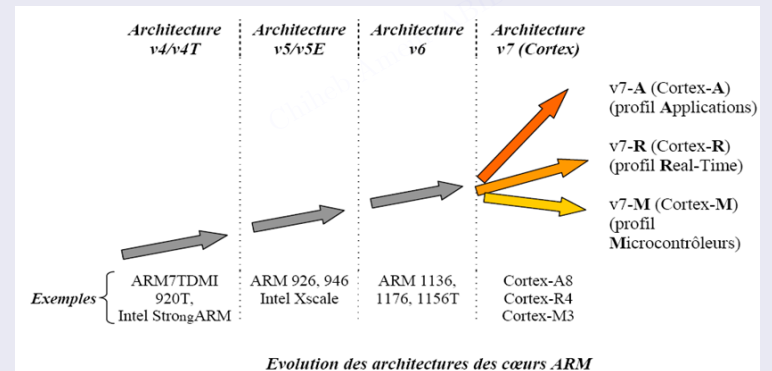
- Version 3 : ARM6, et ARM7
 - ☞ Véritable adressage 32 bits
 - ☞ Ensemble de registres pour le maintien de l'état du processeur
 - ☞ ARM6 : plusieurs variantes (coprocesseur, gestionnaire de la mémoire, cache modifié, etc.)
 - ☞ fréquence : 26-33 MHz
 - ☞ ARM7 fonctionnellement identique à l'ARM6 mais avec des fréquences plus élevées, cache énergétiquement plus performant, meilleure gestion de la mémoire,
 - ☞ fréquence : 40MHz et plus
- Version 4 : ARM8, ARM9 et StrongARM
 - ☞ Certaines versions optionnelles dans la v3 sont intégrées dans la v4 : multiplication étendue, etc.
 - ☞ Pipeline 5 étages : Lecture instruction décodage, exécution, mémoire et écriture registre.
 - ☞ ARM8 : ARM7 + unité d'exécution spéculative, politique d'écriture retardée pour le cache, multiplication 64 bits, 80MIPS à 80MHz
 - ☞ StrongARM : cache Harvard (données et instructions séparés), 100-200MHz
 - ☞ ARM9 : ARM8 + cache Harvard



Versions et implémentation

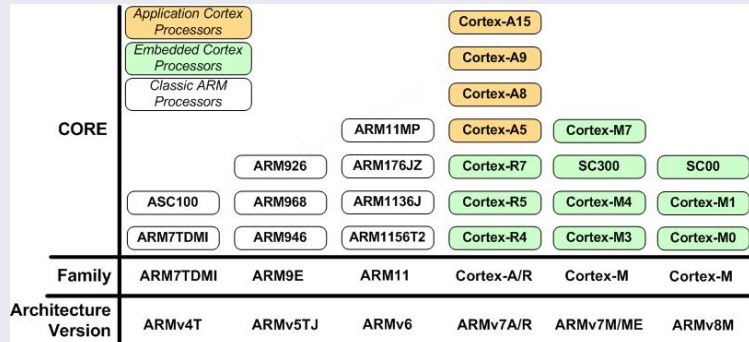
Versions et implémentation

- L'architecture et le jeu d'instructions du cœur ARM ont évolué depuis la première version ARM-v1 jusqu'à la version ARM-v7 depuis laquelle on observe l'apparition de l'appellation Cortex



Versions et implémentation

Versions et implémentation



Versions et implémentation v7

Versions et implémentation v7

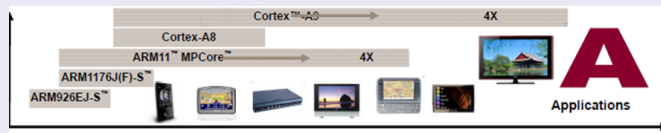
Version 7

- Données manipulées : Word (32 bits), Halfword (16 bits) ou Byte (8 bits)
- Pas de contrainte sur le fait d'être Big ou little endian
- Communication avec les périphériques via un mécanisme de mapping mémoire
- Utilisation : le groupe ST Microelectronics utilise l'ARM7 pour les DSP (Digital Signal Processor) qui vont constituer l'étage de décodage numérique des prochains récepteurs radio DRM (Digital Radio Mondiale)

Les séries Cortex

La série A

- Le profil A (séries Cortex-A) : destiné pour faire tourner des applications complexes telles que les systèmes d'exploitation embarqués (linux, Windows) nécessitant une puissance de traitement élevée et un système de gestion de mémoire virtuelle (MMU : Memory Management Unit)



Les séries Cortex

La série R

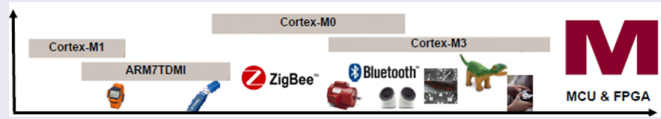
- Le profil R (séries Cortex-R) : destiné principalement aux applications temps réel à contraintes très sévères et exigeant une haute fiabilité et un temps de réponse faible



Les séries Cortex

La série M

Le profil M (séries Cortex-M) : optimisé pour des applications nécessitant une basse consommation en énergie, un coût réduit et un comportement déterministe pour le traitement des interruptions. C'est le profil qui est destiné à être intégré dans les microcontrôleurs (utilisés surtout pour la commande des machines)



Plan

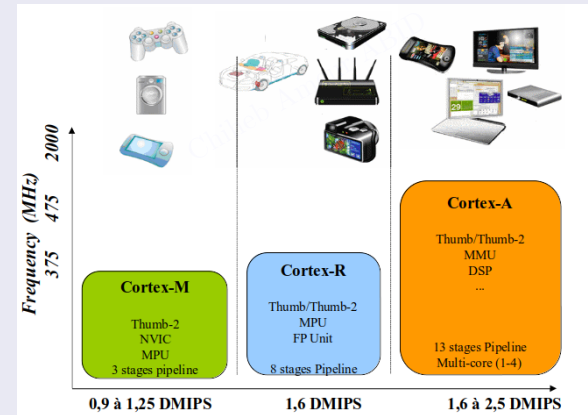
- 1 Introduction
- 2 Historique
- 3 Les différentes architectures du processeur ARM
- 4 Architecture du processeur ARM 32 bits
- 5 Structure d'un programme assembleur, compilation et débogage
- 6 Directives
- 7 Les opérations de base de l'assembleur ARM



Les séries Cortex

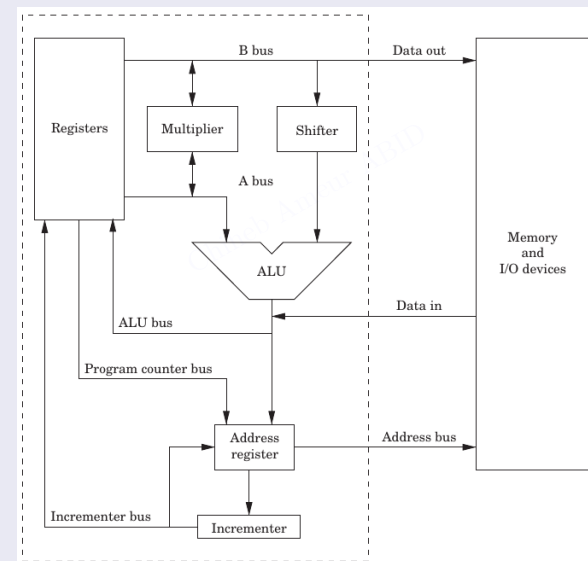
Les séries Cortex

Cortex-XN
X: Profile (A,R,M) | N: Performance level (0..9)



Architecture simplifié d'un processeur ARM 32 bits

Architecture simplifié d'un processeur ARM 32 bits



Les registres

Les registres

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11 (fp)
r12 (ip)
r13 (sp)
r14 (lr)
r15 (pc)
CPSR

- Thirteen general-purpose registers (r0-r12)
- The stack pointer (r13 or sp)
- The link register (r14 or lr)
- The program counter (r15 or pc)
- Current Program Status Register (CPSR)



Architecture LOAD/STORE

Les instructions LOAD et STORE

- Les instructions ne traitent que des données en registre et placent les résultats en registre.
- Les seules opérations accédant à la mémoire sont celles qui copient une valeur mémoire vers un registre (load) et celles qui copient une valeur registre vers la mémoire (store).
- Ces instructions se basent sur le mode d'adressage indirect par registre. On utilise la valeur contenue dans un registre comme adresse mémoire



Architecture LOAD/STORE

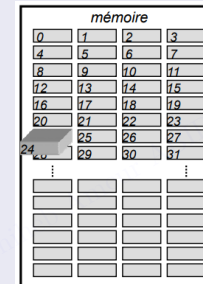
Les instructions LOAD et STORE

- Les instructions ne traitent que des données en registre et placent les résultats en registre.
- Les seules opérations accédant à la mémoire sont celles qui copient une valeur mémoire vers un registre (load) et celles qui copient une valeur registre vers la mémoire (store).
- Ces instructions se basent sur le mode d'adressage indirect par registre. On utilise la valeur contenue dans un registre comme adresse mémoire



Adressage avec ARM

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



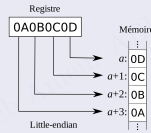
- Les échanges entre la mémoire et les registres se font sur des données de 32 bits, les types de données sont en format 8, 16 ou 32
- Il faut certaines règles pour garder un accès cohérent aux données
 - ☞ L'accès à un mot mémoire (32 bits) doit être aligné sur des adresses divisibles par 4
 - ☞ L'accès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



Arrangement des données dans la mémoire

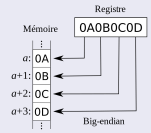
Little et Big Endian

- 1 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

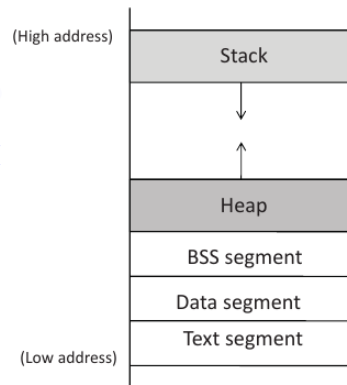
- 2 Big-Endian : la partie la plus significative se trouve à l'adresse la plus petite



Modèle mémoire

Modèle mémoire d'un programme

- Code/Text : contient les instructions du programme (lecture seule)
- Data : variables globales et statiques initialisées par le programmeur
- BSS (Block Started by Symbol) : variables globales et statiques non initialisées
- Heap : allocation dynamique
- Stack : fonctionne en mode LIFO ; stockage de variables locales (définies dans une fonction) et les informations relatives aux appels des fonctions



Plan

- 1 Introduction
- 2 Historique
- 3 Les différentes architectures du processeur ARM
- 4 Architecture du processeur ARM 32 bits
- 5 Structure d'un programme assembleur, compilation et débogage
- 6 Directives
- 7 Les opérations de base de l'assembleur ARM



Structure d'un programme assembleur

Structure d'un programme assembleur

- Un programme assembleur se construit à partir de 5 éléments
 - 1 Les directives : des commandes destinées au compilateur (réservation d'espace mémoire aux variables, inclure d'autres fichiers, etc.)
 - 2 Les instructions assembleurs (mnémoniques)
 - 3 Les pseudo-instructions : ensemble d'instructions et directives
 - 4 Les étiquettes : adresses associées aux instructions ou symboles
 - 5 Les commentaires



Structure d'un programme assembleur

Structure d'un programme assembleur

```

.data ← directives (commence par un .)
@ Define a null-terminated string
str: .asciz "Bonjour_ARM\n"
v: .word 0 ← étiquettes (se termine par :)
.text
.global main
@ Point d'entree principale ← commentaires
main:
@ load pointer to format string
ldr r0, =str
bl printf ← instructions (mnémoniques et opé-
@ Quitter le programme randes)
mov r0, #0 @Valeur de retour
mov r7, #1 @Quitter
svc 0 @Appel system

```



Désassemblage

L'outil objdump

- L'outil objdump permet d'effectuer le désassemblage d'un programme binaire : à partir des instructions ARM, on retrouve le code assembleur
- Désassembler le code binaire du programme précédent
objdump -s -d HelloWorld
- Déduire le codage binaire de l'instruction `mov r0, #0`
- Modifier la valeur de retour, et vérifier à partir du shell la valeur renvoyée



Compilation et débogage

Compilation et exécution d'un programme assembleur

- Écrire le programme précédent et le sauvegarder dans le fichier `HelloWorld.s`
- Compilation et édition de liens
`g++ -o HelloWorld HelloWorld.s`
- Exécution
`chmod +x HelloWorld`
`./HelloWorld`
- Compilation en incluant des informations de débogage
`g++ -o HelloWorld HelloWorld.s -ggdb`



Débogage

Débogage

- Pour effectuer le débogage, il est nécessaire de passer le flag `-g` lors de la compilation
- On utilise l'outil `gdb` pour le débogage
- Quelques commandes utiles pour le débogage
 - ☞ `b main` : placer un point d'arrêt au niveau du `main`
 - ☞ `r` : lancer l'exécution du programme
 - ☞ `s` : exécuter la prochaine instruction (un seul pas)
 - ☞ `i r` : afficher le contenu des registres
 - ☞ `i b` : afficher la liste des points d'arrêt
 - ☞ `disassemble main` : effectuer le désassemblage à partir de `main`
 - ☞ `l` : lister les 10 lignes suivantes
 - ☞ `p (type) <nom variable>` : afficher le contenu de la variable `nomvariable` selon le type spécifié
 - ☞ `x <adresse mémoire>` : afficher le contenu d'une adresse mémoire



Plan

- 1 Introduction
- 2 Historique
- 3 Les différentes architectures du processeur ARM
- 4 Architecture du processeur ARM 32 bits
- 5 Structure d'un programme assembleur, compilation et débogage
- 6 Directives
- 7 Les opérations de base de l'assembleur ARM



Les directives

Les directives

Introduction aux directives

- Les directives sont des commandes destinées au compilateur
 - ☞ Contrôler le comportement du compilateur
 - ☞ Définir des symboles (les variables)
- Directives pour la création des symboles

Directive	Description
.ascii	A string contained in double quotes
.asciz	A zero-byte terminated ascii string
.byte	1-byte integers
.double	Double-precision floating-point values
.float	Floating-point values
.octa	16-byte integers
.quad	8-byte integers
.short	2-byte integers
.word	4-byte integers



Les directives

Déclaration des données

- La déclaration des données en assembleur et son équivalence en C

<code>i: .word 0</code>	<code>static int i = 0;</code>
<code>j: .word 1</code>	<code>static int j = 1;</code>
<code>fmt: .asciz "Hello\n"</code>	<code>static char fmt[] = "Hello\n";</code>
<code>ch: .byte 'A','B',0</code>	<code>static char ch[] = {'A','B',0};</code>
<code>ary: .word 0,1,2,3,4</code>	<code>static int ary[] = {0,1,2,3,4};</code>

- Le fichier binaire généré pour des déclarations en assembleur

line	addr	value	code
1			.data
2	0000	00000000	i: .word 0
3	0004	01000000	j: .word 1
4	0008	48656C6C	fmt: .asciz "Hello\n"
4		6F0A00	
5	000f	414200	ch: .byte 'A','B',0
6	0012	00000000	ary: .word 0,1,2,3,4
6		01000000	
6		02000000	
6		03000000	
6		04000000	



Alignement des données

- Le processeur ARM peut transférer les données vers/de la mémoire par octet, demi-mot ou mot
- Le transfert est optimal lorsque l'adresse d'un mot est divisible par 4 ou lorsque l'adresse d'un demi-mot est divisible par 2
 - ☞ Pour des meilleures performances, il faut placer les mots et les demi-mots dans des emplacements appropriés
- Les directives suivantes permettent d'ajouter des espacements afin d'aligner l'item qui suit dans l'emplacement approprié
 - ☞ `.align log2(nbr0ctets)[, motif]` avec `log2(nbr0ctets)` est le nombre de bits de poids le plus faible à mettre à zéro
 - ☞ `.balign nbr0ctets[, motif]`



Les directives

Exemple des données bien alignées

- En insérant deux octet avec la directive `.align 2`, on arrive à bien aligner les mots du tableau `ary`

line	addr	value	code
1			.data
2	0000	00000000	i: .word 0
3	0004	01000000	j: .word 1
4	0008	48656C6C	fmt: .asciz "Hello\n"
4		6F0A00	
5	000f	414200	ch: .byte 'A','B',0
6	0012	0000	.align 2
7	0014	00000000	ary: .word 0,1,2,3,4
7		01000000	
7		02000000	
7		03000000	
7		04000000	

Les directives

Définir des segments

- Réserver de l'espace mémoire
`.space size, fill`
- Directives pour la structuration d'un programme
 - `.data` soussection : définit un segment de données initialisées par le programmeur
 - `.text` soussection : définit un segment d'instructions
 - `.bss` soussection : définit un segment de données non initialisées par le programmeur. Les données sont par défaut initialisées à zéro
 - `.section` soussection : pour une section personnalisée

Les directives

Les macros

- Les macros permettent de définir de nouvelles instructions basées sur des instructions de base.
- Les macro sont exclusivement des compositions textuelles
 - Elles ne sont pas directement compilées en tant que telles
 - Elles sont traitées par le préprocesseur en appliquant un remplacement

Définir une macro

```
1 .macro macname [macargs ...]
2 @Corps de la macro
3 ...
4 .endm
```

- Pour accéder à un argument `macarg` dans le corps d'une macro, il faut mettre `\macarg`

Les directives

Exemple : les macros

```
1 .macro afficher chaine
2   ldr r0,=\chaine
3   bl printf
4 .endm
5
6 .data
7   str: .asciz "Bonjour_ARM\n"
8   str2: .asciz "Les_macros...\n"
9 .text
10 .global main
11 main:
12   afficher str
13   afficher str2
14   @ Quitter le programme
15   mov r0,#0 @Valeur de retour
16   mov r7,#1 @Quitter
17   svc 0
```

Les directives

Exemple : les macros

```
(gdb) disassemble main
Dump of assembler code for function main:
0x00010490 <+0>: ldr r0, [pc, #20] ; 0x104a4 <main+28>
0x0001049c <+4>: bl 0x10368 <printf@plt>
0x00010490 <+0>: ldr r0, [pc, #10] ; 0x104a8 <main+32>
0x00010494 <+12>: bl 0x10368 <printf@plt>
0x00010498 <+16>: mov r6, #0
0x0001049c <+20>: mov r7, #1
0x000104a0 <+24>: svc 0x00000000
```

Débogage

Mise en application

- Ecrire un programme qui permet de créer les variables suivantes :
 - Le tableau t1 d'octet : 5,8,10,7
 - Le tableau t2 de mot : 0xFFFF,0x000A,0xBBBB
 - La chaîne de caractère str : "Bonjour ARM"
 - La variable vvv codé sur 32 bits ayant la valeur 0xFFFF
- Afficher tous les éléments de t1
- Afficher tous les éléments de t2
- Afficher la chaîne de caractères str
- Afficher vvv en format décimal et hexadécimal

Plan

- 1 Introduction
- 2 Historique
- 3 Les différentes architectures du processeur ARM
- 4 Architecture du processeur ARM 32 bits
- 5 Structure d'un programme assembleur, compilation et débogage
- 6 Directives
- 7 Les opérations de base de l'assembleur ARM

Format d'une instruction assembleur ARM

Format d'une instruction assembleur ARM

- Une instruction assembleur ARM 32 bits est structurée sous formes de champs

31 - 28	27 - 25	24 - 21	20	19 - 16	15 - 12	11 - 0
Condition	Operand type	OpCode	Set Condition Codes	Operand Register	Destination Register	Immediate Operand

- ☞ Condition : l'exécution de l'instruction est conditionnée selon les bits du registre d'état CPSR
- ☞ Operand type : types des opérandes spécifiées dans les bits de 0 à 19
- ☞ Opcode : l'instruction (multiplication, addition, etc.)
- ☞ Set condition code : Indique si l'instruction va mettre à jour ou non le registre CPSR
- ☞ Operand register : le registre utilisé comme donnée (de 0 à 15)
- ☞ Destination register : registre destination du résultat (de 0 à 15)
- ☞ Immediate operand : peut spécifier une valeur comme opérande

Instructions de mouvements de données entre registres

Instructions de mouvements de données entre registres

Les instructions de mouvements permettent de copier une valeur d'un registre à un autre

Syntaxe

```
1 <op>{<cond>}{s} Rd,operand2
2 movt{<cond>} Rd,#immed16
```

Opérations

Name	Effect
mov	$Rd \leftarrow operand2$
mvn	$Rn \leftarrow \neg operand2$
movt	$Rn \leftarrow (immed16 \ll 16) \vee (Rd \wedge 0xFFFF)$

- op spécifie soit **mov** ou **mvn**
- La présence de **s** met à jour le registre d'état **CPSR**
- cond spécifie une condition sur l'exécution de l'instruction



Instructions de transferts registres / mémoire

Les instructions LOAD et STORE

LDR : lire la valeur contenue à une adresse mémoire et la placer dans un registre

```
LDR r0, [r1]  r0 ← mem[r1]
```

STR : écrire la valeur d'un registre à une adresse mémoire

```
STR r0, [r2]  mem[r2] ← r0
```



Instructions de mouvements de données entre registres

Instructions de mouvements de données entre registres

MOV (Move), MVN (Move not)

Affecter une valeur ou la valeur d'un registre dans un registre

```
MOV Rd, #valeur  Rd ← valeur
```

```
MOV r3, #2  R3 ← 2
```

```
MOV Rd, Rn  Rd ← Rn
```

```
MOV r3, r4  R3 ← R4
```

```
MOV Rd, Rm, shift
```

shift : spécifie une opération de décalage sur Rm

```
Rd ← shift(Rn)
```

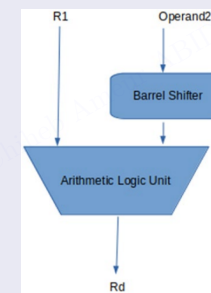
```
MOV r3, r4, LSL #2
```



Les opérations de décalage sur les opérandes

Les opérations de décalage

L'entrée du deuxième opérande de l'UAL est connectée à un **barrel shifter**



Un barrel shifter est un circuit électronique capable d'effectuer des opérations de décalage et de rotation



Les opérations de décalage sur les opérands

Mnémonique de décalage

➤ Décalage : déplacement du mot vers la droite ou vers la gauche. Les positions libérées sont remplies avec des zéros (logique) ou avec le bit de signe (arithmétique)

➤ LSL #n : Logical Shift Left ($1 \leq n \leq 31$)

0011, LSL #1
 0xF000, LSL #1

➤ LSR #n : Logical Shift Right ($1 \leq n \leq 31$)

0111 LSR #1 0xF000, LSR #1

➤ ASR #n : Arithmetic Shift Right ($1 \leq n \leq 31$); préserve le bit de signe

1011, ASR #1
 0xF000, ASR #1

➤ Rotation : décalage circulaire, les bits perdus sont réinjectés

➤ ROR #n : Rotate Right ($0 \leq n \leq 31$)

0011, ROR #1
 0xF000, ROR #1

➤ RRX #n : Rotate Right Extended (ROR étendu de 1 bit avec le bit de retenue C du registre d'état)

0111 RRX #1
 0xF000, RRX #1



Les instructions arithmétiques

Les instructions arithmétiques

Les instructions arithmétiques

➤ Réalisent des opérations arithmétiques binaires sur 2 opérands 32 bits

Instruction	Description	Commentaire
ADD Rd, Rn, operand2	Add	Rd := Rn + operand2
ADC Rd, Rn, operand2	Add with carry	Rd := Rn + operand2 + C
SUB Rd, Rn, operand2	Subtract	Rd := Rn - operand2
SBC Rd, Rn, operand2	Subtract with carry	Rd := Rn - operand2 + C - 1
RSB Rd, Rn, operand2	Reverse subtract	Rd := operand2 - Rn
RSC Rd, Rn, operand2	Rev. sub. With carry	Rd := operand2 - Rn + C - 1

➤ C désigne la valeur du bit de retenue dans le registre d'état CPSR

➤ L'ordre des opérands est inversé pour la soustraction inverse

➤ Il est possible d'appliquer le barrel shifter pour le troisième opérande



Les instructions arithmétiques

Opérations de Multiplication

① Multiplication de deux registres 32 bits pour produire un résultat sur 32 bits

➤ MUL Rd, Rm, Rs ➔ Rd = Rm * Rs

➤ MLA Rd, Rm, Rs, Rn ➔ Rd = Rm * Rs + Rn

② Multiplication de deux registres 32 bits pour produire un résultat sur 64 bits

➤ SMUL RdLo, RdHi, Rm, Rs : multiplication signée

➤ UMUL RdLo, RdHi, Rm, Rs : multiplication non signée

$$RdHi:RdLo = Rm * Rs$$



Opérations de division

➤ Disponibles qu'à partir de Raspberry Pi 2 : ARM Cortex-A53 et Cortex-A72

Name	Effect	Description
sdiv	$Rd \leftarrow Rm \div Rn$	Signed Divide
udiv	$Rd \leftarrow Rm \div Rn$	Unsigned Divide



Les instructions arithmétiques

Les instructions logiques

- Réalisent des opérations logiques sur chaque paire de bits des 2 opérandes : ET, OU, OU-EXCLUSIF et BIT CLEAR

Instruction	Description	Commentaire
AND Rd, Rn, <i>operand2</i>	And	Rd := Rn and <i>operand2</i>
ORR Rd, Rn, <i>operand2</i>	Or	Rd := Rn or <i>operand2</i>
EOR Rd, Rn, <i>operand2</i>	Exclusive or	Rd := Rn xor <i>operand2</i>
BIC Rd, Rn, <i>operand2</i>	Bit clear	Rd := Rn and not <i>operand2</i>



Merci pour votre attention



Questions?

