

Assembleur ARM et convention d'appel

Pablo de Oliveira <pablo.oliveira@uvsq.fr>

March 23, 2015

I

Introduction

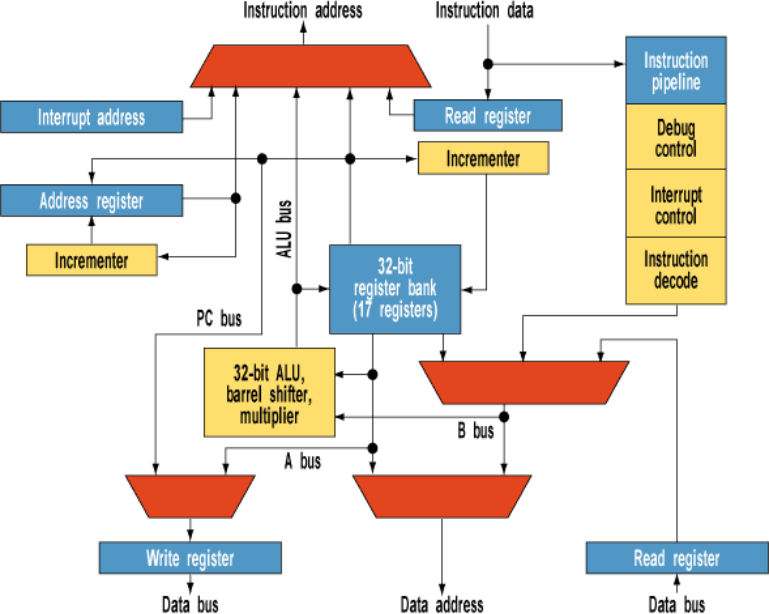
ARM Historique

- ▶ Développé par Acorn computers en 1983
- ▶ Architecture simple, très versatile
- ▶ ARM vend des licences complètes de son cœur:
- ▶ Facilement intégrable dans un SoC (Système sur puce)
- ▶ Un des processeurs les plus utilisés au monde (75% des puces 32 bits embarquées)

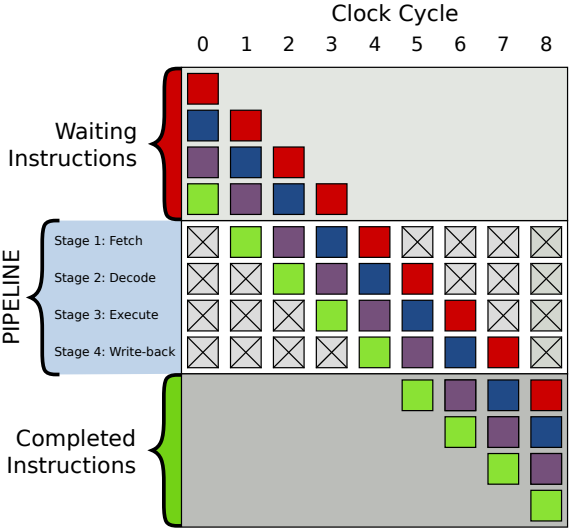
Architecture ARM

- ▶ RISC (Reduced Instruction Set Computer), jeu d'instruction réduit et facile à décoder
- ▶ Abondance de registres généraux
- ▶ Instructions de taille fixe 32 bits en mode normal (16 bits en mode Thumb, 8 bits en mode Jazelle)
- ▶ Modified Harvard (Sépare cache programme et cache données)
- ▶ Architecture Pipeline à 8 étages
- ▶ Consommation énergétique faible $\sim 0.4 \text{ mW} / \text{MHz} + \text{cache}$

Micro-architecture

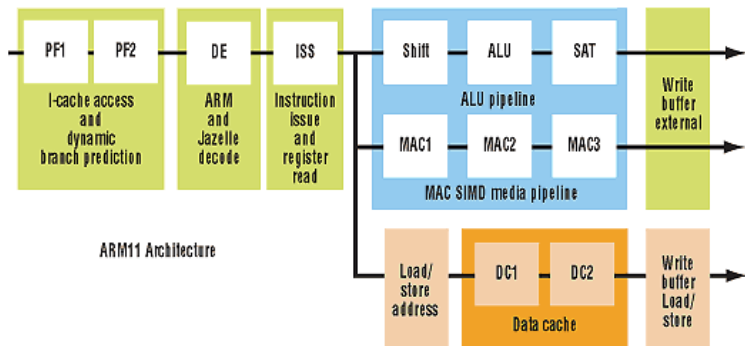


Pipeline



Source: C. Burnett

Pipeline détaillé ARM11 (Armv6)



Source: ARM Architecture Reference Manual

Registres

- ▶ Sur tous les processeurs ARM on a les registres 32 bits suivants:
 - ▶ 11 registres d'usage général R0 à R10
 - ▶ 1 registre pour le FP (frame pointer) FP / R11
 - ▶ 1 registre réservé pour le linker IP / R12 (utilisé lors de l'appel de fonctions "éloignées")
 - ▶ 1 pointeur de pile SP / R13
 - ▶ 1 registre pour sauver l'adresse de retour LR / R14
 - ▶ 1 registre pour le PC (program counter) PC / R15

Instruction Assembleur

- ▶ Voici un exemple d'instruction ARM
11100001101000000000000000000001
- ▶ Pour programmer en assembleur on utilise des mnémoniques:

```
mov r0, r1 # déplace le contenu de r1 dans r0
```

- ▶ Un assembleur (sorte de compilateur) transforme les mnémoniques en code machine

Drapeaux d'état (Condition flags)

- ▶ Un registre spécial signale plusieurs conditions après l'exécution d'une opération
- ▶ Chaque condition peut valoir 1 ou 0:
 - ▶ N: vaut 1 si le dernier résultat était négatif
 - ▶ Z: vaut 1 si le dernier résultat était nul
 - ▶ C: vaut 1 si le dernier résultat dépasse 32 bits
 - ▶ V: vaut 1 si le dernier résultat dépasse 32 bits en complément à deux

Exemple 1

Soit $r1 = r2 = 0x7fffffff (= 2^{31} - 1)$, le plus grand entier positif en complément à deux sur 32 bits.

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

N = 1 (car le résultat $0xffffffffe = -2$ est négatif)

Z = 0 (car -2 est non-nul)

C = 0 (car le résultat tient en 32 bits non signés)

V = 1 (car le résultat dépasse l'addition en CA2)

Exemple 2

Soit $r1 = 1$ et $r2 = 0xffffffff$ ($= -1$ en CA2)

Exemple 2 (réponse)

Soit $r1 = 1$ et $r2 = 0xffffffff$ ($= -1$ en CA2)

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

N = 0 (car le résultat 0 est non-négatif)

Z = 1 (car le résultat est nul)

C = 1 (car le résultat dépasse l'addition non-signée
=> wrap-around)

V = 0 (car le résultat est correct en CA2)

Instructions Arithmétiques

- ▶ $\text{ADD}\{\text{S}\} \ r1, r2, r3 \implies r1 \leftarrow r2 + r3$
- ▶ $\text{SUB}\{\text{S}\} \ r1, r2, r3 \implies r1 \leftarrow r2 - r3$
- ▶ Si le suffixe S est utilisé, les instructions mettent à jour les drapeaux
- ▶ SBC et ADC avec retenue

Addition 64 bits: $\{r1,r0\} += \{r3,r2\}$

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

```
ADDC r1, r1, r3
```

Multiplications

- ▶ $\text{MUL}\{S\} \ r1, r2, r3 \implies r1 \leftarrow r2 \times r3$
- ▶ $\text{MLA}\{S\} \ r1, r2, r3, r4 \implies r1 \leftarrow r4 + r2 \times r3$
- ▶ Attention les registres $r3$ et $r1$ doivent être différents

Opérations Logiques

- ▶ AND, ORR, EOR respectivement ET, OU et XOR bit à bit
- ▶ TST r0, r1 réalise un ET logique entre r0 et r1
- ▶ Le résultat est jeté, mais les drapeaux d'état sont mis à jour.
- ▶ Utile pour tester un masque de bits.

Comparaison

- ▶ `CMP r0, r1`
- ▶ Soustrait `r1` et `r0`, et mets les drapeaux à jour selon le résultat

Utilisation de valeurs immédiates

- ▶ `ADD r0, r0, #1` incrémente une valeur de 1
- ▶ À la place du troisième registre on peut utiliser des immédiats
- ▶ La valeur de l'immédiat est codée sur 12 bits
- ▶ Premier choix: se limiter aux valeurs $-2^{11} \dots 2^{11} - 1$
- ▶ Problème, on ne peut pas modifier les 32 bits d'un registre
- ▶ Codage retenu:
 - ▶ 4 bits de poids fort: position
 - ▶ 8 bits de poids faible: valeur

~ Immédiat = valeur (permutés circulairement de 2^{position}) ~

Exemple: Valeurs immédiates

Pos. Valeur.

0000 01111111 -> 00000000 00000000 00000000 01111111
=> 127

0001 01111111 -> 11000000 00000000 00000000 00011111
=> 3221225503

0010 11111111 -> 11110000 00000000 00000000 00000111
=> 4026531847

L'utilisateur n'a pas à coder position + valeur, l'assembleur s'en charge.

- ▶ Comment est codée la valeur 256 ?
- ▶ Comment est codée la valeur 257 ?

Exemple: Valeurs immédiates

Comment est codée la valeur 256 ?

1010 10000000

Comment est codée la valeur 257 ?

Impossible ! Toutes les valeurs ne peuvent pas être représentées :(
Mais il existe un autre système que l'on étudiera plus tard.

Instructions prédicatées

- ▶ Comment influencer le comportement du programme selon les drapeaux ?
- ▶ Toutes les instructions ARM sont prédicatées. On peut leur adjoindre un suffixe qui conditionne leur exécution:
 - ▶ AL: toujours exécutée
 - ▶ NV: jamais exécutée
 - ▶ EQ (resp. NE): exécutée si le drapeau Z est à 1 (resp. 0)
 - ▶ MI (resp. PL): exécutée si le drapeau N est à 1 (resp. 0)
 - ▶ VS (resp. VC): exécutée si le drapeau V est à 1 (resp. 0)
 - ▶ CS (resp. CC): exécutée si le drapeau C est à 1 (resp. 0)
- ▶ Après un CMP,
 - ▶ GE, LT, GT, LE: greater-equal, less-than, greater-than, less-equal
- ▶ D'autres combinaisons existent pour comparer des nombres non-signés

Exemple: instructions prédicatées

```
if ( x == 0 ) x--;
```

```
CMP r1, #0
```

```
SUBEQ r1, r1, #1
```

Exercice: Comment écrire la fonction suivante ?

```
//  $x = \max(x, y)$ 
```

```
if (x < y) x=y;
```

Exemple: instructions prédicatées

Exercice: Comment écrire la fonction suivante ?

```
// x = max(x, y)  
if (x < y) x=y;
```

```
CMP r1, r2  
MOVLT r1, r2
```

Contrôle: branches

- ▶ Les branchent sautent d'un endroit du programme à un autre.
- ▶ Permettent de modifier le flot des instructions.
- ▶ $B \text{ offset} \implies PC = PC + \text{offset} * 4$

```
boucle_infinie:
```

```
    add r1, r1, 1
```

```
    b boucle_infinie
```

- ▶ `boucle_infinie` est une étiquette qui permet à l'assembleur de calculer de combien il faut déplacer le compteur ordinal.
- ▶ Ici il faut se déplacer une instruction en arrière (4 octets)
- ▶ Offset $\frac{-4}{4} = -1$?
- ▶ Attention au pipeline. La valeur $PC + \text{offset}$ est calculée 2 cycles plus tard après le décodage de l'instruction. La valeur du PC sera donc en avance de 2 instructions.
- ▶ Offset $\frac{-4-8}{4} = \frac{-12}{4} = -3$ OK

Contrôle: if / else

- ▶ Comment faire une structure if /else en assembleur ARM ?
- ▶ Utilisation des prédicats sur B.

```
if ( a < 0) {/*then*/} else {/*else*/}
```

```
CMP r0, #0  
BGE else  
..then..  
B out  
else:  
..else..  
out:
```

Contrôle: boucle

- ▶ Comment faire une structure de boucle ?
- ▶ Exercice:
 - ▶ Le registre r1 contient la valeur entière positive n .
 - ▶ Le registre r2 contient la valeur entière m .
 - ▶ Écrire un programme assembleur qui calcule: m^n
 - ▶ On ne se préoccupera pas ici des problèmes de dépassement.

Contrôle: boucle (solution)

```
    mov r0, #1
    cmp r1, #0
mult:
    beq out
    mul r0,r0,r2
    subs r1,r1,#1
    b mult
out:
```

Barrel Shifter

- ▶ L'ALU (Unité arithmétique et logique) possède un étage de Shift (décalage)
 - ▶ Les instructions peuvent être préfixées d'un code pour le décalage.
 - ▶ `MOV r0, r1, LSR #2`
1. Décale la valeur dans r1, sans extension de signe, vers la droite (Logical Shift Right) de 2 positions
 2. Déplace le résultat dans r0
- ▶ $r0 = \frac{r1}{4}$

Barrel Shifter

	<u>Code</u>	<u>Effet</u>
LSL		Logical Shift Left
RSR		Logical Shift Right
ASR		Arithmetic Shift Right
ROR		Rotate Right
RRX		Rotate Right (inclue le bit C de retenue)

Exemple: Shifter

- ▶ Soit $r2 = -16$ et $r1 = 10$ en Complément à 2

ADD r0, r1, r2, ASR #2

- ▶ Divise r2 par 4 avec extension de signe:

$$\begin{aligned} -16 \gg 2 &= 111111111111111111111111111111110000 \gg 2 \\ &= 11111111111111111111111111111111100 \\ &= -4 \end{aligned}$$

- ▶ Ajoute 10, le résultat final est -6

Accès mémoire

- ▶ La mémoire est adressée par octets.
- ▶ On peut accéder à la mémoire en écriture (store) ou en lecture (load):
 - ▶ LDR/STR lit écrit un mot (32 bits)
 - ▶ LDRH/STRH lit écrit un demi-mot (16 bits)
 - ▶ LDRB/STRB lit écrit un octet (8 bits)
 - ▶ LDRSB/STRSB lit écrit un octet en étendant le signe (8 bits)

Modes d'adressage

- ▶ `LDR r0, [r1]` charge dans le registre r0, le contenu à l'adresse r1
- ▶ `LDR r0, [r1, #16]` charge dans le registre r0, le contenu à l'adresse $r1 + 16$
- ▶ `STR r0, [r1, r2, LSL#2]` écrit le contenu du registre r0, à l'adresse $r1 + 4 * r2$
- ▶ `STR r0, [r1, #-16] !`: pratique pour se déplacer dans un tableau

1. Écrit le contenu dans r0 à l'adresse r1-16.
2. $r1 \leftarrow r1 - 16$

Adressage relatif au PC

```
.section data
.align 2
message:
    .string "Hello"

.section text
.align 2
code:
    ldr r0, =message
```

- ▶ Charge dans r0, l'adresse de message.
- ▶ Le compilateur exprime le chargement de manière relative au PC.

Exemple: code généré

```
83cc: e59f0008  ldr r0, [pc, #8]
```

```
...
```

```
83dc: 00008450
```

```
...
```

```
8450: 6c6c6548  11eH
```

```
8454: 0000006f  000o
```

Codage des instructions ARM

- ▶ Dans le mode normal, une instruction est représentée comme un mot mémoire de 32 bits.
- ▶ Comment coder une instruction:
 - ▶ Le décodage doit être rapide et simple
 - ▶ Codage de taille fixe
 - ▶ Un champ pour le prédicat d'instruction
 - ▶ Un champ pour l'opcode
 - ▶ Plusieurs champs pour les opérandes, les shifts et les offsets

Codage des instructions ARM (détail)

ARM Instruction Formats

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00		
Conditional	0	0	I	Opcode				S	Rn				Rd				Operand 2						ALU										
Conditional	0	0	0	0	0	0	A	S	Rd				Rn				Rs			1	0	0	1	Rm				Multiply					
Conditional	0	0	0	1	0	B	0	0	Rs				Rd				0	0	0	0	1	0	0	1	Rm				Swap				
Conditional	0	0	0	1	0	PS	0	0	1	1	1	1	Rd				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MRS
Conditional	0	0	0	1	0	PS	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	Rm				MSR(all)			
Conditional	0	0	I	1	0	PS	1	0	1	0	0	1	1	1	1	Source Operand						MSR(flag)											
Conditional	0	1	I	Pr	U	B	W	LS	Rn				Rd				Offset						LDR/STR										
Conditional	0	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	X	X	X	X	Undefined		
Conditional	1	0	0	Pr	U	S	W	LS	Rn				Registers (R15-R0)												LDM/STM								
Conditional	1	0	1	Ln	Offset												Branch																
Co-processor instructions																																	
Conditional	1	1	0	Pr	U	N	W	LS	Rn				CRd	CP#	Offset						Transfer												
Conditional	1	1	1	0	CP Op				CRn	CRd	CP#	CP	0	CRm	Op																		
Conditional	1	1	1	0	CP Op	LS	CRn	CRd	CP#	CP	1	CRm	RTransfer																				
Conditional	1	1	1	1	Ignored By ARM																					SWI							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00		

Source: Timothy Roddis

Convention d'appel (ABI)

- ▶ En assembleur un appel de fonction est un simple saut de programme.
- ▶ Comment passer les arguments ?
- ▶ Où stocker l'adresse de retour ?
- ▶ Comment éviter que les registres soient écrasés par la fonction appelée ?
- ▶ Plusieurs solutions, pour pouvoir interfacier différents programmes et bibliothèques on définit une norme commune: Application Binary Interface.
- ▶ On va étudier la nouvelle norme ABI pour ARM appelée EABI.

Passage des arguments

- ▶ Les quatre premiers arguments sont passés dans les registres $r0, r1, r2, r3$
- ▶ Attention: Si les arguments sont des mots 64 bits, on ne pourra en passer que 2.
- ▶ Les arguments supplémentaires sont passés sur la pile.
- ▶ Le résultat d'une fonction est passé également dans les registres $r0, r1, r2, r3$.

La pile

- ▶ La pile est une région mémoire propre à un thread. La convention en ARM est que la pile croît vers le bas.
- ▶ Le dernier élément de la pile est toujours pointé par le registre SP.
- ▶ Pour sauvegarder ou restaurer des registres depuis la pile on peut utiliser les instructions:

@ sauve les registres lr,r0,r1,r2

@ sur la pile et mets à jour l'adresse

@ de sp (ici décrémente de 32 octets)

```
stmfd sp!, {lr, r0, r1, r2}
```

@ restaure les registres lr,r0,r1,r2

@ depuis la pile et mets à jour

@ l'adresse de sp (ici incrémente de 32 octets)

```
ldmfd sp!, {lr, r0, r1, r2}
```

Jump and Link

- ▶ Pour appeler une fonction en assembleur, on va utiliser une branche
- ▶ Mais contrairement à un simple saut, il faut pouvoir retourner à l'appelant
- ▶ L'instruction 'bl étiquette:
- ▶ Saute à étiquette
- ▶ Enregistre dans le registre LR l'adresse de retour (PC + 4 au moment du saut" l'adresse de retour (PC + 4 au moment du saut)

Retour de fonction

- ▶ Pour retourner à l'appellant on utilisera l'instruction:

```
bx lr
```

- ▶ Cette instruction copie le contenu du registre lr dans le registre pc.

Problème: Sauver le contexte

- ▶ Considérez le programme suivant

```
fct3:  
    bx lr  
fct2:  
    bl fct3  
    bx lr  
fct1:  
    bl fct2
```

- ▶ Que va t'il se passer ?

Problème: Sauver le contexte

- ▶ L'instruction `b1` dans `fct2` écrase l'ancienne valeur de `1r`.
- ▶ On ne peut donc jamais retourner à `fct1`.
- ▶ Solution:
- ▶ préserver la valeur de `1r` sur la pile à l'entrée d'une fonction
- ▶ restaurer la valeur de `1r` depuis la pile avant d'appeler `bx`

Problème: Sauver le contexte

```
fct3:  
    bx lr  
fct2:  
    stmfd sp!, {lr}  
    bl fct3  
    ldmfd sp!, {lr}  
    bx lr  
fct1:  
    stmfd sp!, {lr}  
    bl fct2  
    ldmfd sp!, {lr}  
    bx lr
```

- ▶ Les fonctions feuilles comme fct3 n'ont pas besoin de sauvegarder lr.

Registres callee-save et caller-save

- ▶ De la même manière que `lr` peut être écrasé par les appels de fonctions, d'autres registres peuvent l'être.
- ▶ Il faut sauvegarder le contexte:
- ▶ Certains registres (caller-save) doivent être sauvegardés par l'appellant, `r0-r3`
- ▶ D'autres registres (callee-save) doivent être sauvegardés par l'appelé, `r5-r12`
- ▶ Si une fonction appelée n'utilise pas certains registres callee-save, elle n'est pas obligé de les sauver sur la pile.

Organisation de la pile

