

ASSEMBLEUR (ARM)

B. Miramond – Polytech Nice Sophia

Introduction

- utilisation d'un mnémonique + opérandes réduisant l'effort de codage
(ex : `mov R0,R1`)
- le mnémonique + op n'est qu'une représentation différente de la même information
- le microprocesseur n'utilise que la première représentation, c'est l'opération d'assemblage qui traduit (c'est un compilateur qui génère un code du premier type) le code assembleur en code machine
- L'ensemble (mnémonique + opérandes) constitue le langage ASSEMBLEUR
- Rem : pour un même microprocesseur ? plusieurs langages assembleurs possibles (ils se distinguent surtout par les directives, macros, les pseudo-instructions ...) (`org 300`), (`a : ds.b 1`), (`.data`)

Points clefs du ARM

- Architecture load-store
- Instructions de traitement à 2 ou 3 adresses
- Instructions conditionnelles (et branchement)
- Load et store multiples
- APCS (ARM Procedure Call Standard)
- En un cycle horloge: opération ALU+shift
- Interfaçage avec coproc. simplifié (ajout de registres, de type, ...)
- Combine le meilleur du RISC et du CISC
- On le trouve dans la GBA, routeurs, Pockets PC, PDA, ...

Que manipule-t-on dans ce langage ?

- Des registres
- Des adresses
- Des données
- Des instructions assembleurs (32 bits)
- Des directives
- Des pseudo-instructions

Exemple de programme

```
.text ← directive
.globl _main

_start: ← label
        b _main
_main:
    ⇒ code op  opmrs R0,CPSR ← opérandes

#attention à la taille des immédiats !!
    ldr R1,=0xDFFFFFFF
    and R0,R0,R1
    msr CPSR_f,R0

#autre solution:pas de masquage, on force tous les bits
#
    ldrb R0,mys
    ldrb R1,re
    adds R0,R0,R1
    strb R0,de
```

```
ldrb R0,te
ldrb R1,boul
adc R0,R0,R1
strb R0,gom

mov pc,lr

.align
mys:    .space 1
te:     .space 1
re:     .space 1
boul:   .space 1
de:     .space 1
gom:    .space 1

.ltorg
.end
```

Environnement de Prog.

Vous pouvez programmer en assembleur

- En « compilant » le code C avec l'option `-S` (outil GNU)
- En « cross-compilant » vers une autre architecture (embarquée)
- En simulant le processeur ARM sur des environnements commerciaux ou libres

Embest IDE Pro -- Education Edition - Disassembly

File Edit View Project Build Debug Tools Window Help

Workspace 'TD1': 3 proj

- ex1 files
 - Project Source Fil
 - Project Header Fi
- ex2 files
 - Project Source Fil
 - Project Header Fi
 - TD1ex2.s
- TD1 files
 - Project Source Fil
 - Project Header Fi

C:\EmbestIDE\Examples\arm\asm\TD1ex2.s

```
.text
.globl _main

_start:

b _main
_main:
    mrs R0,CPSR

#attention à la taille des immédiats !!
    ldr R1,=0xDFFFFFFF
    and R0,R0,R1
    msr CPSR_f,R0

#autre solution:pas de masquage, on force
#    msr CPSR_f,#0xD
```

Disassembly

Address	Op	Op2	Op3	Op4
0x02000000	b		0x2000004	
0x02000004	mrs	r0, cpsr		
0x02000008	ldr	r1, [pc, #30]		; 0x2000040
0x0200000c	and	r0, r0, r1		
0x02000010	msr	cpsr_flg, r0		
0x02000014	ldrb	r0, [pc, #1c]		; 0x2000038
0x02000018	ldrb	r1, [pc, #1a]		; 0x200003a
0x0200001c	adds	r0, r0, r1		
0x02000020	strb	r0, [pc, #14]		; 0x200003c
0x02000024	ldrb	r0, [pc, #d]		; 0x2000039
0x02000028	ldrb	r1, [pc, #b]		; 0x200003b
0x0200002c	adc	r0, r0, r1		
0x02000030	strb	r0, [pc, #5]		; 0x200003d
0x02000034	mov	pc, lr		
mys :				
0x02000038	andeq	r0, r0, r0		
de :				

Rec

--- Current ---

- R0: 0x00000000
- R1: 0x00000000
- R2: 0x00000000
- R3: 0x00000000
- R4: 0x00000000
- R5: 0x00000000
- R6: 0x00000000
- R7: 0x00000000
- R8: 0x00000000
- R9: 0x00000000
- R10: 0x00000000
- R11: 0x00000000
- R12: 0x00000000
- R13: 0x00000000
- R14: 0x00000000
- R15: 0x02000000**
- SP: 0x00000000
- LR: 0x00000000
- PC: 0x02000000**
- CPSR: 0x000000d3
- SPSR: 0x00000000

--- User ---

Register Peripheral

Address: 0x02000038

Address	Op	Op2	Op3	Op4
02000039	00	.		
0200003A	00	.		
0200003B	00	.		
0200003C	00	.		
0200003D	00	.		
0200003E	00	.		
0200003F	00	.		
02000040	FF	.		
02000041	FF	.		
02000042	FF	.		
02000043	DF	.		
02000044	FF	.		
02000045	FF	.		
02000046	FF	.		
02000047	FF	.		
02000048	FF	.		

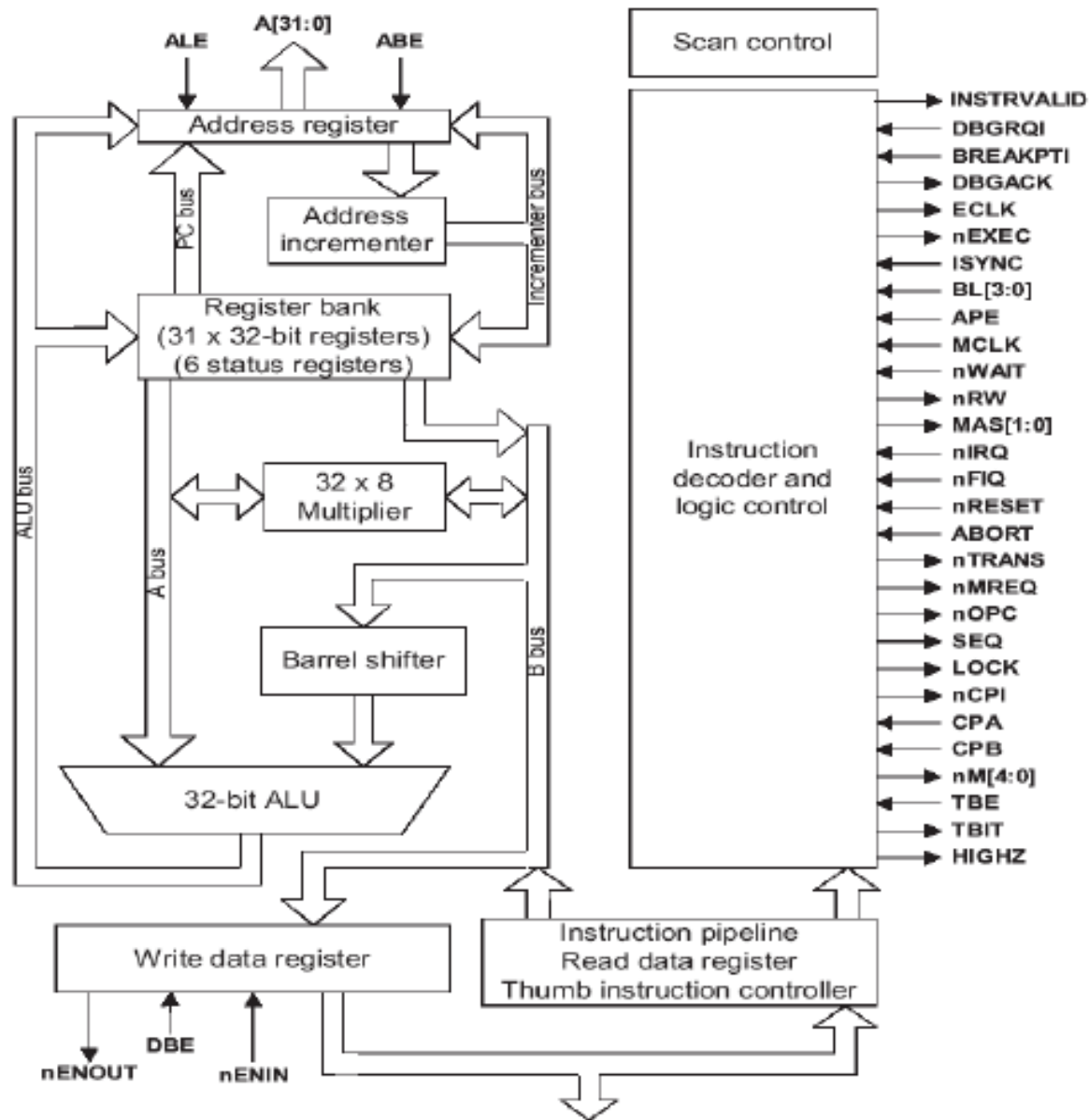
Address	Op	Op2	Op3	Op4
24 0024 0D00DFE5	ldrb	R0, te		
25 0028 0B10DFE5	ldrb	R1, boul		
26 002c 0100A0E0	adc	R0, R0, R1		
27 0030 0500CFE5	strb	R0, gom		
28				
29				
30				
31 0034 0EF0A0E1	mov	pc, lr		
32				
33				
34				
35 0038 00	mys:	.space 1		
36 0039 00	te:	.space 1		
37 003a 00	re:	.space 1		
38 003b 00	boul:	.space 1		
39 003c 00	de:	.space 1		
40 003d 00	gom:	.space 1		
41				
42 003e 0000FFFF	ltorg			

Ready

Ln 1, Col 1

DOS

Read



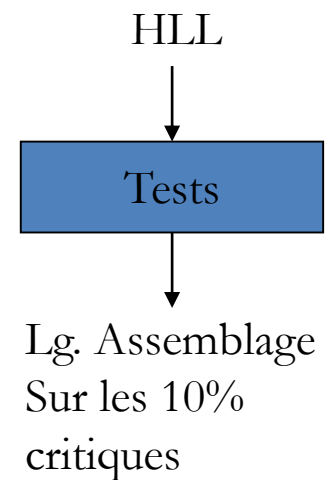
Pourquoi ce retour en arrière ?

Pourquoi s'intéresser au langage d'assemblage ?

- Savoir écrire en langage d'assemblage lorsque l'application demande d'être optimisée en performances.
- Pour le portage d'OS sur de nouvelles architectures
- Dans la plupart des ordinateurs embarqués, les systèmes disposent de quelques Mo de mémoire. Comprendre le code embarqué pour comprendre l'espace occupé.
- Comprendre comment fonctionne un compilateur.
- Comprendre la dynamique d'une architecture matérielle.

Exemple

	Programmer-years to produce the program	Program execution time in seconds
Assembly language	50	33
High-level language	10	100
Mixed approach before tuning		
Critical 10%	1	90
Other 90%	9	10
Total	<u>10</u>	<u>100</u>
Mixed approach after tuning		
Critical 10%	6	30
Other 90%	9	10
Total	<u>15</u>	<u>40</u>



LE LANGAGE

Structure d'un programme ASM

- Des directives de compilation
 - Allocation de variables
 - Réserve, initialisation mémoire
 - De segmentation de la mémoire
- Des instructions à assembler en langage machine, éventuellement préfixées d'étiquettes
- Des directives de compilation
 - Allocation de textes en mémoire (affichage : printf)

Directives de compilation

- .int permet d'allouer un ou plusieurs mots de 32 bits en mémoire
- .halfword idem pour 16bits
- .byte idem pour 8 bits
 - {<label>} .int <expression> {, <expression>...}
- .fill permet d'effectuer une réservation mémoire avec initialisation éventuelle
 - <label> .fill <repeat>{, <size>{, <value>}}
- .EQU permet d'associer une expression à un symbole
 - {<label>} .equ <expression>

Les directives principales

Les directives sont des instructions pour l'assembleur (compilateur) uniquement, elles commencent par « . ».

- `.align`: permet d'aligner en mémoire des données de types différents
`.byte 0x55`
`.align` sinon erreur à l'assemblage
`.word 0xAA55EE11`
- `.ascii`: permet de saisir des caractères (sans le NULL)
`.ascii "JNZ"` insère les octets `0x4A, 0x4E, 0x5A`
- `.asciz`: comme `.ascii` mais insère à la fin le caractère NULL
`.Asciz "JNZ"` insère les octets `0x4A, 0x4E, 0x5A, 0x00`

Les directives principales

- .byte: déclare une variable type octet et initialisation

var1: .byte 0x11

var2: .byte 0x22,0x33

- .hword: déclare une variable type deux octets et initialisation

var3: .hword 0x1122

var4: .hword 0x44

- .word: déclare une variable type word (32bits) et initialisation

var5: .word 0x11223344

- .space: déclare une variable de taille quelconque (pas d'initialisation)

var6: .space 10 réserve 10 octets

Les directives principales

- `.equ`: associe une valeur à un symbol
`.equ dix, 5+5`
`mov R3,#dix`
- `.global` (ou `.globl`): l'étiquette (symbol) est visible globalement
`.global _start` **`_start` est reconnue par le linker GNU**
`_start` doit apparaître le code principal
- `.text`, `.data`, `.bss` : début de section text, data, bss (pour le linker pour générer l'ELF)
- `.end` : fin du code source
- `.ltorg` : insère « ici » les constantes temporaires pour **LDR =**
`ldr R1,=0x11111111`
`.ltorg`

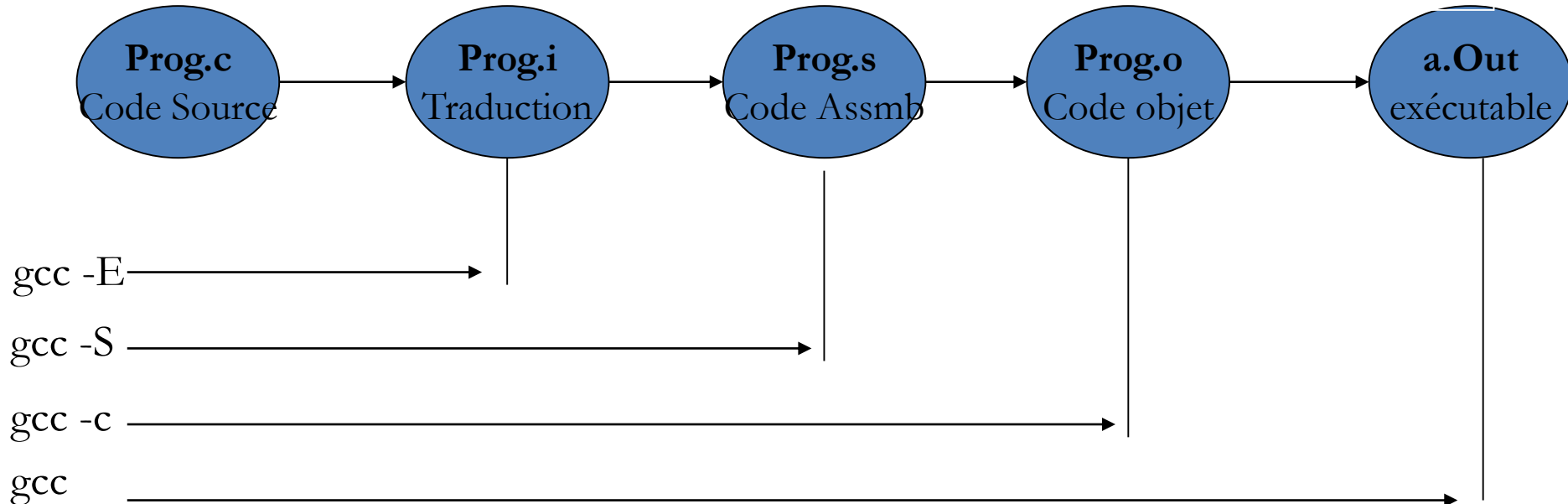
Etapes du compilateur GNU

Preprocesseur

Compilation

Assembleur

Edition de lien



Format des modules objets (.o)

- De manière à pouvoir utiliser des compilateurs, assembleurs et éditeurs de liens provenant de vendeurs différents (interopérabilité), 2 formats de fichiers objets (sections) ont été standardisés
 - COFF (Common Object File Format)
 - ELF (Executable and Linker Format)

Structure d'un module objet ELF

www.x86.org/ftp/manuals/tools/eld.pdf

- En-tête
 - Nom de fichier, Taille, adresse de début
- Espace objet (divisé en sections)
 - Code binaire
 - Zone de données
- Table des symboles
 - Symboles utilisables et à satisfaire
- Informations complémentaires
 - Auteurs, outils utilisé, versions, environnement...

Types de contenu

- Le compilateur organise le programme par types de contenus appelés *sections* :
- `.text/.code` = Instructions binaires
- `.data` = Données binaires initialisées
- `.bss` = (Block Started by Symbol) Données globales non initialisées
- `.rodata` = Read Only Data (Chaîne de caractères)
- `.comment` = commentaires
- `.symtab` = table des symboles
- `.rel` = table de résolution
- ...
- Le Standard ELF permet de définir autant de sections qu'on le souhaite avec n'importe quel nom
- Outils pour lire les sections : **objdump** (tous fichiers binaires) et **readelf** (ELF seulement)

Types de contenu affichés par la commande **'nm'**

- B – dans la zone .bss
- D – dans la zone .data
- C – non initialisé
- T – dans la zone .text
- U – Undefined symbol

Rangement des variables

			.data	.bss	.rodata	Pile
Globale	static	initialisée				
		non init.				
	dyna	init.				
		non init				
Locale	static	init				
		non init				
	dyna	init				
		non init				
G/L	const					

Rangement des variables

			.data	.bss	.rodata	Pile
Globale	static	initialisée				
		non init.				
	dyna	init.				
		non init				
Locale	static	init				
		non init				
	dyna	init				
		non init				
G/L	const					

Instructions

On les rassemble en 3 groupes:

- I. Instructions de contrôles
- II. Mouvements de données
- III. Opérations de traitements (arith. & log.)

Inst./struct. de contrôles

- Elles déroutent le cours normal du programme
- Basées sur le couple (CMP, B{cond})
- Tests complexes (CMP{cond}) (voir TD)
- Elles cassent le pipeline (peut mieux faire ?)
- Elles servent à réaliser
 - Les alternatives (if, case)
 - Les itérations (for, while, repeat)
 - Les procédures (BL) \Rightarrow normalisation

Branchements

The encoding of conditional branch and supervisor call instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	opcode											

Table A5-8 Branch and supervisor call instructions

opcode	Instruction	See
not 111x	Conditional branch	<i>B</i> on page A7-207
1110	Permanently UNDEFINED	<i>UDF</i> on page A7-471
1111	Supervisor call	<i>SVC</i> on page A7-455

Encoding T1

All versions of the Thumb instruction set.

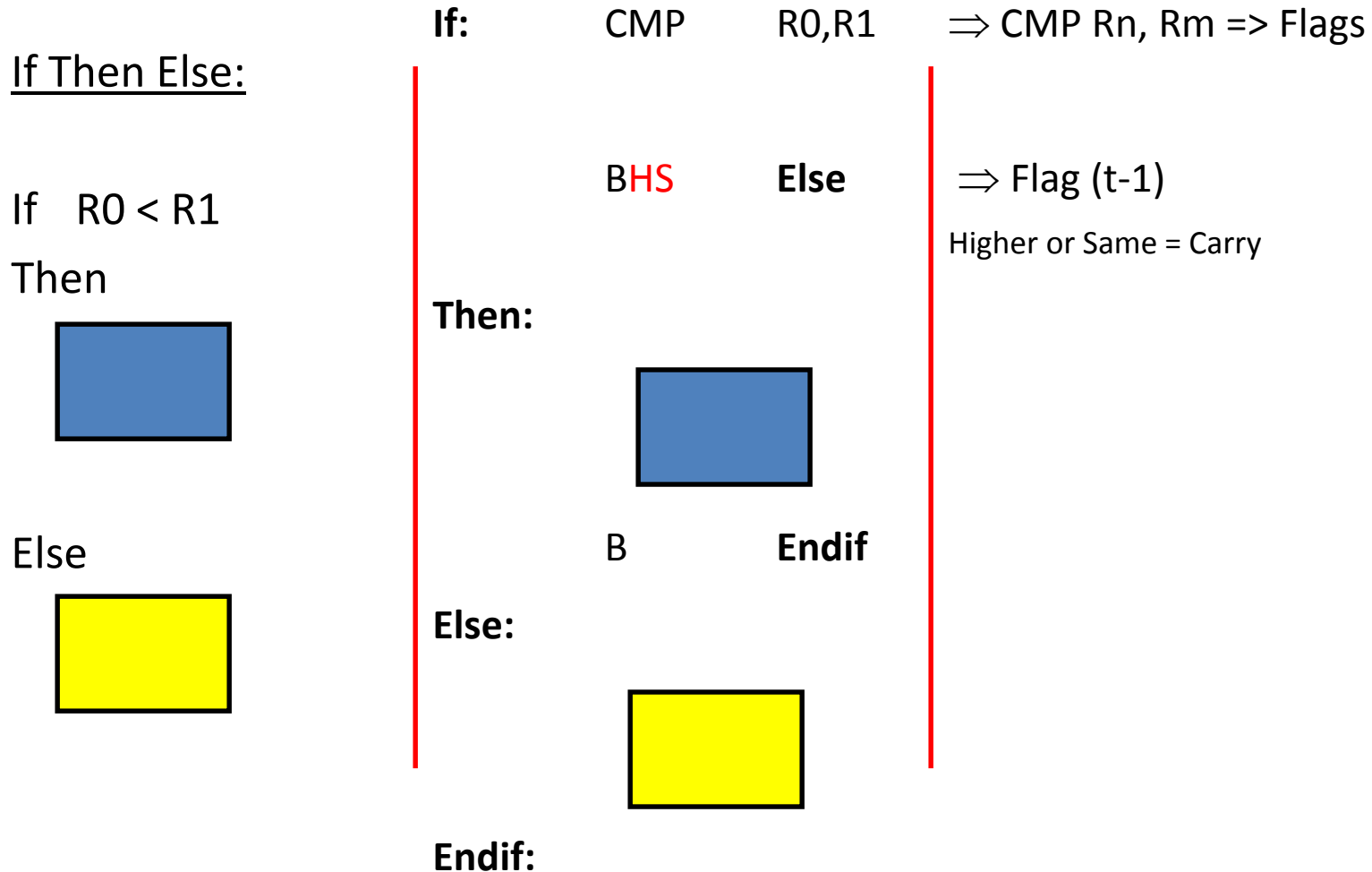
B<c> <label>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

Cond (toutes les instructions !)

Opcode [31:28]	monic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Inst./struct. de contrôles



Inst./struct. de contrôles

Case Of:

Case R2 Of

-2:



-3:



Otherwise:



Case:

mDeux: CMP R2,#-2 (remplacé par CMN R1,#2) !!
BNE mTrois



B Endcase

mTrois: CMP R2,#-3
BNE Otherwise



B Endcase

Otherwise:




Endcase:

Inst./struct. de contrôles

While do:

While $R3 \geq R1$
do



While:	CMP	R3,R1	⇒ CMP Rn,<Operand2>
	BLT	Endwhile	⇒ si Rn LT Operand2 en signé
#	BLO	Endwhile	⇒ si Rn LO Operand2 en nonsigné
do:			
			
	B	While	
Endwhile:			

Inst./struct. de contrôles

For:

For R3 = 1 to n fin
do



For:

LDR R3,=n



SUBS R3,R3,#1

BNE **For**

Endfor:

⇒ décrémentation de 1

Appels de procédures, les Piles

Les procédures sont appelées avec Branch + Link (BL)

Les paramètres sont passés par:

- Adresses (noms de variables)
- Par registres (taille et nombre limités)
- Par adresse dans des registres (nombre limité)
- Par adresse ou valeur dans la pile \Rightarrow notion de pile

Appels de procédures, les Piles

.globl _start

.globl _main

_start: B _main

_main:

MOV R1,#2

MOV R2,#3

BL proc ⇒ branchement à **proc** et LR (R14)=PC-4 (Pipeline)

STR R3,res

proc:

ADD R3,R1,R2

MOV PC,LR

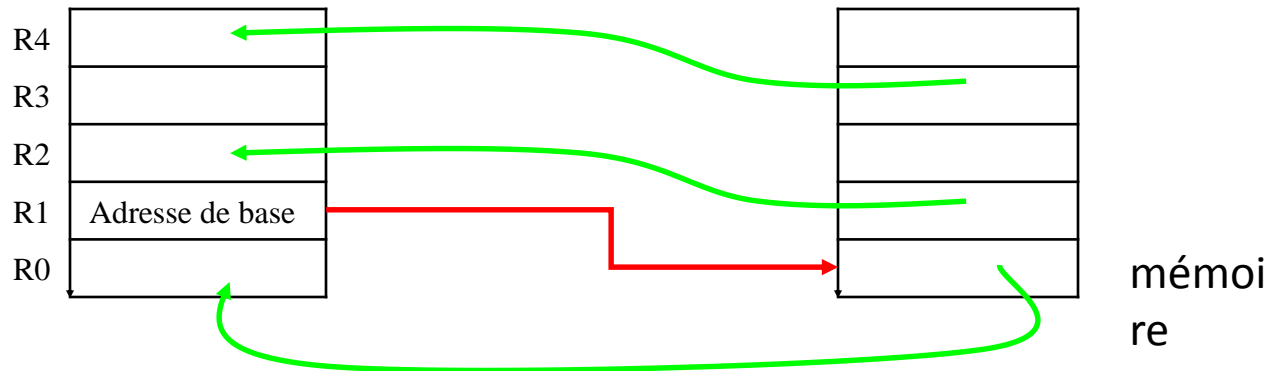
Nombre de registres limité ⇒ utilisation des piles (C, JAVA)

Appels de procédures, les Piles

LDR et STR n'utilisent qu'un seul registre

LDM et STM utilisent plusieurs registres

LDMIA R1,{R0,R2,R4} ($r0 = \text{mem}_{32}[r1]$, $r2 = \text{mem}_{32}[r1+4]$, $r4 = \text{mem}_{32}[r1+8]$)



De même

STMIA R1,{R0,R2,R4} ($\text{mem}_{32}[r1] = r0$, $\text{mem}_{32}[r1+4] = r2$, $\text{mem}_{32}[r1+8] = r4$)

Appels de procédures, les Piles

Ces instructions permettent l'implémentation de Piles (PUSH et POP) :

- PUSH : STMIA
- POP : LDMDB

Le sommet de pile est un emplacement vide et on empile de manière croissante

- Et R13 (SP) est considéré comme le pointeur de pile

Ces piles sont utilisées pour le passage de paramètres aux procédures avec normalisation (ARM Procedure Call Standard) ou non

Appels de procédures, les Piles

Utilisation efficace des piles en appel de procédure :

- On passe les paramètres par la pile, ou les registres
- La procédure sauvegarde tous les registres de travail (le prog appelant ne voit pas les changements)
- La procédure sauvegarde le LR (elle peut appeler une autre procédure)
- La procédure restaure les registres sauvegardés avant le retour

Appels de procédures, APCS (réduit)

Le APCS (ARM Procedure Call Standard) fournit un mécanisme standardisé permettant d'associer des routines C, assembleur, ...

En définissant:

- Restrictions sur l'utilisation des registres
- Convention d'utilisation de la pile
- Convention de passage de paramètres
- Structuration du contexte de la procédure dans la pile

Il existe plusieurs versions de APCS

Appels de procédures, APCS

Les registres prennent des noms particuliers:

R10 : SL (stack limit)

R11 : FP (frame pointer)

R12 : IP (scratch register)

R13 : SP (stack pointer)

R14 : LR (link register)

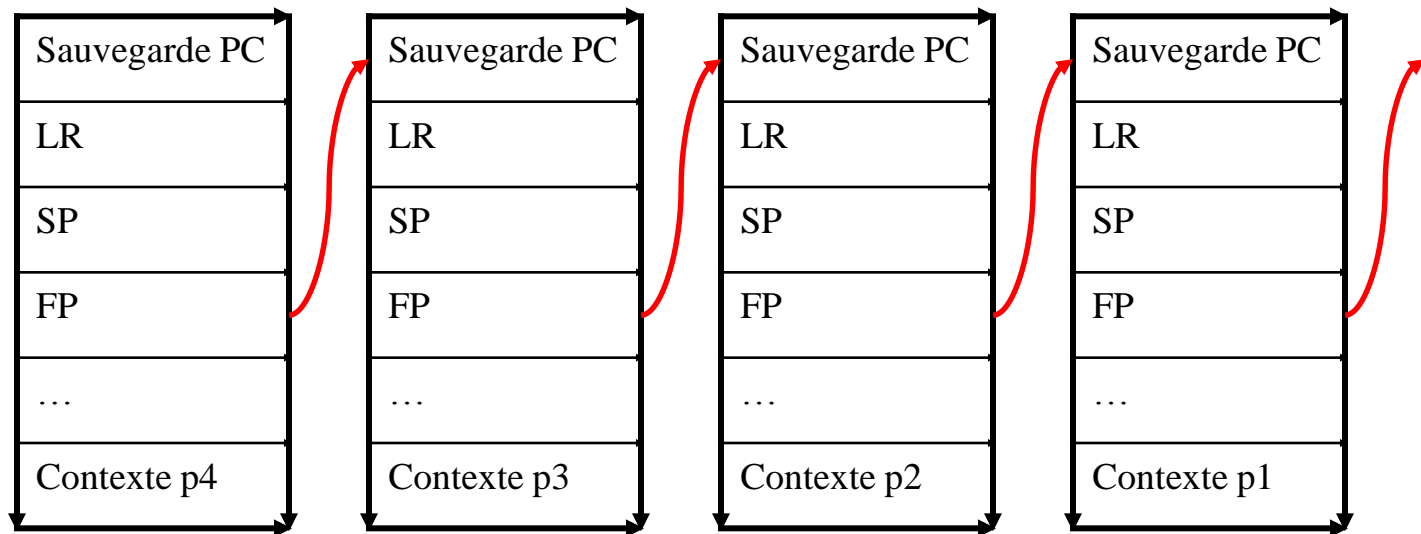
R15 : PC

Et R0-R3 : A1-A4 arguments de proc/registres de travaux/résultats

R4-R9 : V1-V6 variable registre

Appels de procédures, APCS

La pile (et les différents registres) permet de conserver une structure d'appel pour les appels imbriqués:



Appels de procédures, APCS

Exemple

```
int somme(int x1, int x2, int x3, int x4, int x5)
{ return x1+x2+x3+x4+x5; }
//extern int somme(int x1, int x2, int x3, int x4, int x5);
int __main()
{ somme(2,3,4,5,6); }
```

.global	__main
__main:	mov ip, sp
	stmfd sp!, {fp, ip, lr, pc}
	sub fp, ip, #4
	sub sp, sp, #4
	mov r3, #6
	str r3, [sp, #0]
	mov r0, #2
	mov r1, #3
	mov r2, #4
	mov r3, #5
	bl somme
	ldmea fp, {fp, sp, pc}

	.text
	.global somme
	.type somme,function
somme:	mov ip, sp
	stmfd sp!, {fp, ip, lr, pc}
	sub fp, ip, #4
	sub sp, sp, #16
	str r0, [fp, #-16]
	str r1, [fp, #-20]
	str r2, [fp, #-24]
	str r3, [fp, #-28]
	ldr r2, [fp, #-16]
	ldr r3, [fp, #-20]
	add r3, r2, r3
	ldr r2, [fp, #-24]
	add r3, r3, r2
	ldr r2, [fp, #-28]
	add r3, r3, r2
	ldr r2, [fp, #4]
	add r3, r3, r2
	mov r0, r3
	ldmea fp, {fp, sp, pc}

II. INSTRUCTIONS DE DÉPLACEMENT DE DONNÉES

Instructions load/store (data flow)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opA					opB										

These instructions have one of the following values in opA:

- 0b0101.
- 0b011x.
- 0b100x.

Encoding T1 All versions of the Thumb instruction set.
STR<c> <Rt>, [<Rn>{, #<imm5>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		

Encoding T2 All versions of the Thumb instruction set.
STR<c> <Rt>, [SP, #<imm8>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							

Table A5-5 16-bit Load/store instructions

opA	opB	Instruction	See
0101	000	Store Register	STR (register) on page A7-428
0101	001	Store Register Halfword	STRH (register) on page A7-444
0101	010	Store Register Byte	STRB (register) on page A7-432
0101	011	Load Register Signed Byte	LDRSB (register) on page A7-286
0101	100	Load Register	LDR (register) on page A7-256
0101	101	Load Register Halfword	LDRH (register) on page A7-278
0101	110	Load Register Byte	LDRB (register) on page A7-262
0101	111	Load Register Signed Halfword	LDRSH (register) on page A7-294
0110	0xx	Store Register	STR (immediate) on page A7-426
0110	1xx	Load Register	LDR (immediate) on page A7-252
0111	0xx	Store Register Byte	STRB (immediate) on page A7-430
0111	1xx	Load Register Byte	LDRB (immediate) on page A7-258
1000	0xx	Store Register Halfword	STRH (immediate) on page A7-442
1000	1xx	Load Register Halfword	LDRH (immediate) on page A7-274
1001	0xx	Store Register SP relative	STR (immediate) on page A7-426
1001	1xx	Load Register SP relative	LDR (immediate) on page A7-252

Instructions (Data Flow)

En fait LDR/STR \Rightarrow

- LDR/STR simple (un seul registre)
- SWAP simple
- LDR/STR multiple (une liste de registres)

On utilise un registre contenant une adresse mémoire \Rightarrow registre de base

LDR R0,[R1] \Rightarrow On charge dans R0 le word (32bits) mem[R1]

STR R0,[R1] \Rightarrow On range dans mem[R1] (4 octets) R0

L'adresse est placée dans R1 avec un ADR ou LDR (voir directives)

Chargement/rangement multiples

- Permet de manipuler des blocs mémoires de manière itérative
- 4 cas selon que l'adresse est modifiée avant ou après l'accès mémoire
 - IA (Increment After)
 - IB (Increment Before)
 - DA (Decrement After)
 - DB (Decrement Before)
- LDM {<cond>}<addr_mode><Rn>{!}, <registers>

Instructions (Data Flow)

- Des LDR (indirect pour source), STR (indirect pour destination)
- Des MOV (constantes ou registre)

Les LDR sont souvent remplacés par des MOV (sauf adressage indirect)

LDR R1,R2	⇒	Error
MOV R1,R2	⇒	OK
LDR R1,=0x00FFFFFF	⇒	ldr r1, [pc, #4]
LDR R1,=1020	⇒	mov r1, #1020
LDR R1,[R2]	⇒	ldr r1, [r2]

Instructions (Data Flow)

Résumé (avec $R_t = R0$, $R_n = R1$) :

- `LDR R0, Imm8` \Rightarrow direct
- `LDR R0, [R1]` \Rightarrow indirect
- `LDR R0, [R1, #offset]` \Rightarrow indirect pré-indexé
- `LDR R0, [R1, #offset]!` \Rightarrow indirect pré-indexé et auto-incrémenté
- `LDR R0, [R1], #offset` \Rightarrow indirect post-indexé et auto-incrémenté
- `LDR R0, [R1, -R2]!` \Rightarrow indirect pré-indexé et auto-incrémenté
- `LDR R0, [R1], -R2` \Rightarrow indirect post-indexé et auto-incrémenté

Instructions (Data Flow)

Si R1 « pointe » (ADR R1,tab) sur une zone mémoire (vecteur, matrice)

Rem: une zone mémoire = vecteur, matrice, tableau, ...

ADD R1, R1, #2 \Rightarrow R1 pointe maintenant sur le prochain half-word

LDRH R0, [R1] \Rightarrow R0 reçoit le half-word : $R0 = \text{mem}_{16}[R1]$

On peut utiliser une **pré**-indexation (R1 est modifié avant le transfert):

LDRH R0, [R1,#2] $\Rightarrow R0 = \text{mem}_{16}[R1+2]$

\Rightarrow R1 mis à jour ??? Quelle syntaxe ???

III. INSTRUCTIONS DE TRAITEMENT

Instructions de traitement

Règles pour le traitement des données:

- Les opérandes sont 32 bits, constantes ou registres
- Le résultat 32 bits dans un registre
- 3 opérandes: 2 sources et 1 destination
- En signé ou non signé
- Peut mettre à jour les Flags (S)
- Le registre destination peut être un des registres sources

Instructions arithmétiques et logiques

The encoding of data processing instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	opcode									

Encoding T1

All versions of the Thumb instruction set.

CMP<C> <Rn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm		Rn			

Table A5-3 16-bit data processing instructions

opcode	Instruction	See
0000	Bitwise AND	AND (register) on page A7-201
0001	Exclusive OR	EOR (register) on page A7-239
0010	Logical Shift Left	LSL (register) on page A7-300
0011	Logical Shift Right	LSR (register) on page A7-304
0100	Arithmetic Shift Right	ASR (register) on page A7-205
0101	Add with Carry	ADC (register) on page A7-187
0110	Subtract with Carry	SBC (register) on page A7-380
0111	Rotate Right	ROR (register) on page A7-368
1000	Set flags on bitwise AND	TST (register) on page A7-466
1001	Reverse Subtract from 0	RSB (immediate) on page A7-372
1010	Compare Registers	CMP (register) on page A7-231
1011	Compare Negative	CMN (register) on page A7-227
1100	Logical OR	ORR (register) on page A7-336
1101	Multiply Two Registers	MUL on page A7-324
1110	Bit Clear	BIC (register) on page A7-213
1111	Bitwise NOT	MVN (register) on page A7-328

a) Instructions Logiques

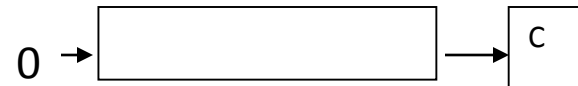
Elles opèrent bit à bit

(Elles sont très utilisées pour les E/S)

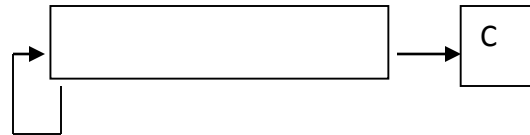
- Des instructions en soit:
 - TST, TEQ (and et eor) \Rightarrow pas de destination, CPSR mis à jour
 - AND, EOR, ORR, BIC (and not)
 \Rightarrow destination, CPSR mis à jour si {S}
- Des instructions indirectes:
 - MOV R2,R1 LSL #5
 - MOV R3,R4 LSR R6
 - \Rightarrow LSL, LSR, ASR, ROR, RRX

Instructions Logiques

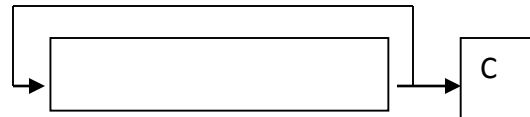
- LSR: logical shift right



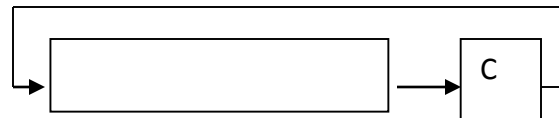
- ASR: arithmetic shift right



- ROR: rotate right



- RRX: rotate right extended



Un seul bit \Rightarrow remplacé par ROR #0

Instructions logiques

The encoding of Shift (immediate), add, subtract, move, and compare instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	opcode													

Table A5-2 16-bit shift (immediate), add, subtract, move and compare encoding

Encoding T1

All versions of the Thumb instruction set.

LSLS <Rd>, <Rm>, #<imm5>

LSL<c> <Rd>, <Rm>, #<imm5>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5			Rm		Rd					

opcode	Instruction	See
000xx	Logical Shift Left ^a	<i>LSL (immediate)</i> on page A7-298
001xx	Logical Shift Right	<i>LSR (immediate)</i> on page A7-302
010xx	Arithmetic Shift Right	<i>ASR (immediate)</i> on page A7-203
01100	Add register	<i>ADD (register)</i> on page A7-191
01101	Subtract register	<i>SUB (register)</i> on page A7-450
01110	Add 3-bit immediate	<i>ADD (immediate)</i> on page A7-189
01111	Subtract 3-bit immediate	<i>SUB (immediate)</i> on page A7-448
100xx	Move	<i>MOV (immediate)</i> on page A7-312
101xx	Compare	<i>CMP (immediate)</i> on page A7-229
110xx	Add 8-bit immediate	<i>ADD (immediate)</i> on page A7-189
111xx	Subtract 8-bit immediate	<i>SUB (immediate)</i> on page A7-448

b) Instructions Arithmétiques

- Elles permettent l'addition, la soustraction, la multiplication
- Elles peuvent prendre en compte le flag C (retenue) et le mettre à jour

-ADD, ADC: addition, addition plus C

-SUB, SBC, RSB, RSC: soustraction, soustraction –NOT(C), inversée, ...

SUBS R1,R2,R3 $\Rightarrow R1=R2-R3$

RSBS R1,R2,R3 $\Rightarrow R1=R3-R2$

-multiplications

Instructions Arithmétiques

Un exemple utilisant l'instruction MLA:

```
MOV    R11, #20
MOV    R10, #0
LOOP:  LDR    R0,[R8], #4
        LDR    R1,[R9], #4
        MLA    R10,R0,R1,R10  $\Rightarrow$   $R10=R10+R0*R1$ 
        SUBS   R11,R11,#1  $\Rightarrow$  SUB et CMP car {S}
        BNE    LOOP
```

\Rightarrow C'est le produit scalaire de deux vecteurs

Instructions Arithmétiques

Les flags sont mis à jour avec {S}

⇒ on peut également utiliser CMP (SUBS) ou CMN (ADDS)

CMP R1,#5 ⇒ R1-5 : les flags sont mis à jour, résultat non stocké

- Utilisé avant une instruction conditionnelle (ARM)
- Utilisé avant un branchement conditionnel (B{cond} ou B{cond}L)
- Associé aux structures de contrôles

REM: TST (and) et TEQ (eor) pour comparaisons logiques

ASSEMBLEUR (ARM)

B. Miramond – Polytech Nice Sophia