

Programmation assembleur (ASM)

Instructions de traitement

Prof. Daniel Rossier
Version 1.3 (2015-2016)

Plan

- Jeu d'instructions (*x86 & ARM*)
- Instructions de **traitement** (*x86 & ARM*)
- Macro-instructions

2

Cours ASM - Institut REDS/HEIG-VD

Dans les prochains chapitres, nous découvrirons les jeux d'instructions des processeurs *x86* et *ARM*. Il ne s'agira en aucun cas d'une étude exhaustive des instructions, mais il s'agira plutôt d'aborder les instructions fondamentales. Les instructions sont divisés en **3 groupes fondamentaux**:

- Les instructions de **traitement**

Ce sont les instructions agissant sur les registres uniquement dans le contexte d'opérations arithmétiques ou logiques. Elles comprennent également les instructions de comparaisons et de tests.

- Les instructions de **transfert de données**

Ce sont les instructions utilisées dans le cadre d'échange de données entre la mémoire et les registres; ces instructions permettent de gérer les interactions entre le cœur de processeur et les périphériques (*RAM, I/O, etc.*). Les instructions de transfert de données sont associées à différents modes d'adressage.

- Les instructions de **branchement**

Finalement, les instructions de branchement permettent de contrôler le déroulement d'un programme durant son exécution.

Ce chapitre se concentre sur les instructions de traitement.

Jeu d'instructions x86 (1/3)

- Jeux d'instructions x86
 - 8086/8088
 - IA-32 (ou x86)
 - x86-64 (ou x64 ou AMD64)
- Evolution des jeux d'instructions
 - 8086/8088, 80286, 80386, 80486, Pentium, Pentium Pro, AMD K7, Pentium III, Pentium 4, Pentium 4/SSE3, Pentium 4 6x2, x86-64, etc.
 - x87, 80287, 80387, Pentium Pro, Pentium 4/SSE3
 - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4
- 2 types d'assembleur
 - Norme *Intel*
 - *NASM*
 - Norme *AT&T*
 - *GNU Assembler*

3

Cours ASM - Institut REDS/HEIG-VD

Comme nous l'avons vu dans le chapitre précédent, l'architecture x86 a été reprise par de nombreux fabricants et a subi d'importantes évolutions matérielles; il en va également de son jeu d'instructions qui n'a sans cesse évolué.

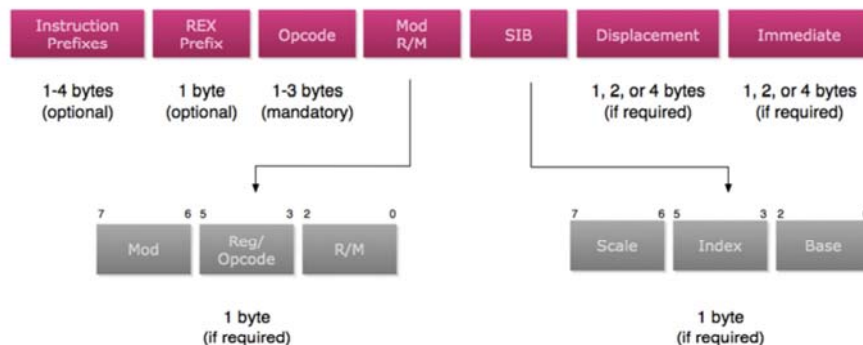
Le jeu d'instructions IA-32 utilisé par la société *Intel* représente l'une des implémentations pour l'architecture x86. C'est un jeu d'instruction largement répandu pour les architectures 32 bits.

Les compilateurs d'assemblage (ou *assembleurs*) se déclinent principalement en deux familles : ceux suivant la norme d'*Intel* (assembleur *nasm*, *yasm*, etc.) et ceux suivant la norme *AT&T* (*GNU assembler*), norme qui sera considérée dans ce cours. Ces deux normes diffèrent principalement au niveau de la syntaxe du langage, mais un code *Intel* peut être facilement converti en un code *AT&T*, et vice-versa.

Par la suite, nous utiliserons le terme "*instructions x86*" pour faire référence en fait au jeu d'instructions IA-32.

Jeu d'instructions x86 (2/3)

- Encodage des instructions **entre 1-17 octets**
- **Opcode** Code de l'opération
- **SIB** Mode d'adressage
- **Immediate** Valeur immédiate



4

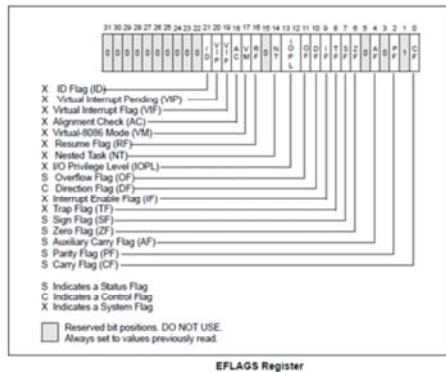
Cours ASM - Institut REDS/HEIG-VD

Une instruction représente **l'assemblage** d'une instruction avec ses opérandes et autres valeurs associées (valeur immédiate, *offset*, etc.); du point de vue de la programmation, les opérandes correspondent aux *arguments* de l'instruction. En revanche, du point de vue du processeur, l'instruction exécutée **doit contenir** toutes les informations nécessaires à l'exécution de celle-ci.

L'encodage d'une instruction x86 est présenté ci-dessus; la taille d'une instruction peut être variable, entre 1 et 17 *bytes*, et comprend différents champs dont les plus importants sont les suivants: **rex prefix** (informations sur le-s registre-s utilisé-s comme opérande-s), **opcode** (code opératoire de l'instruction), et **immediate** (valeur immédiate). De plus, les champs (*scale*, *index*, *base*, *displacement*) sont liés aux modes d'adressage et donnent une très grande flexibilité lors d'accès mémoire; dans ce cas on se réfère à des instructions de transfert qui seront étudiées ultérieurement.

Jeu d'instructions x86 (3/3)

- Cartes de référence **IA-32**
- Instructions, opérandes, modes d'adressage



IA32 Instructions

movl Src, Dest	Dest = Src
movzbl Src, Dest	Dest (long) = Src (byte), sign extend
addl Src, Dest	Dest = Dest + Src
subl Src, Dest	Dest = Dest - Src
imull Src, Dest	Dest = Dest * Src
xall Src, Dest	Dest = Dest << Src
rarrl Src, Dest	Dest = Dest >> Src arithmetic shift
rrl Src, Dest	Dest = Dest >> Src logical shift
rrcl Src, Dest	Dest = Dest >> Src
andl Src, Dest	Dest = Dest & Src
orl Src, Dest	Dest = Dest Src
incl Dest	Dest = Dest + 1
deci Dest	Dest = Dest - 1
negl Dest	Dest = - Dest
notl Dest	Dest = ~ Dest
leal Src, Dest	Dest = address of Src
cmls Src1, Src2	Sets CCs Src1 - Src2
cmlsl Src1, Src2	Sets CCs Src1 & Src2
jmp label	Jump
je label	Jump equal
jne label	Jump not equal
js label	Jump negative
jns label	Jump non-negative
jg label	Jump greater (signed)
jge label	Jump greater or equal (signed)
jl label	Jump less (signed)
jle label	Jump less or equal (signed)
ja label	Jump above (unsigned)
jb label	Jump below (unsigned)
push Src	keesp = keesp - 4; Mem(keesp) = Src
pop Dest	Dest = Mem(keesp); keesp = keesp + 4
call label	push address of next instruction; jmp label
ret	keesp = Mem(keesp); keesp = keesp + 4

Addressing modes

Immediate	
val Val	val: constant integer value
movl \$17, %eax	
Normal	
(R)	Mem(Reg[R])
R	register R specifies memory address
movl (%eax), %eax	
Displacement	
(R)	Mem(Reg[R]+D)
R	register specifies start of memory region
D	constant displacement D specifies offset
movl 5(%ebp), %edx	
Indexed	
(R)(R)(D)	Mem(Reg[Rb]+S*Reg[Ri]+D)
D	constant displacement 1, 2, or 4 bytes
Rb	base register: any of 8 integer registers
Ri	index register: any except %esp
S	scale: 1, 2, 4, or 8
movl 0x100(%ecx, %eax, 4), %edx	

Instruction suffixes

b	byte
w	word (2 bytes)
l	long (4 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Registers

%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

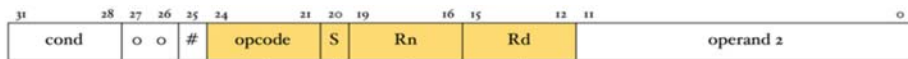
5

Cours ASM - Institut REDS/HEIG-VD

La carte de référence rapide ci-dessus donne un résumé des instructions x86 de base ainsi que les différents *flags* du registre d'état et modes d'adressage.

Jeux d'instructions ARM (1/4)

- Différents jeux d'instructions
 - **v4** (ARM-7), **v5** (ARM-9), **v6** (ARM-11, Cortex-Mx), **v7** (Cortex-Ax)
- Instructions sur **32 bits** (ou 16 bits en mode *Thumb*)



- Exécution des instructions en **un seul cycle machine**
- **Exécution conditionnelle** des instructions
 - Un code d'exécution fait partie de l'instruction.

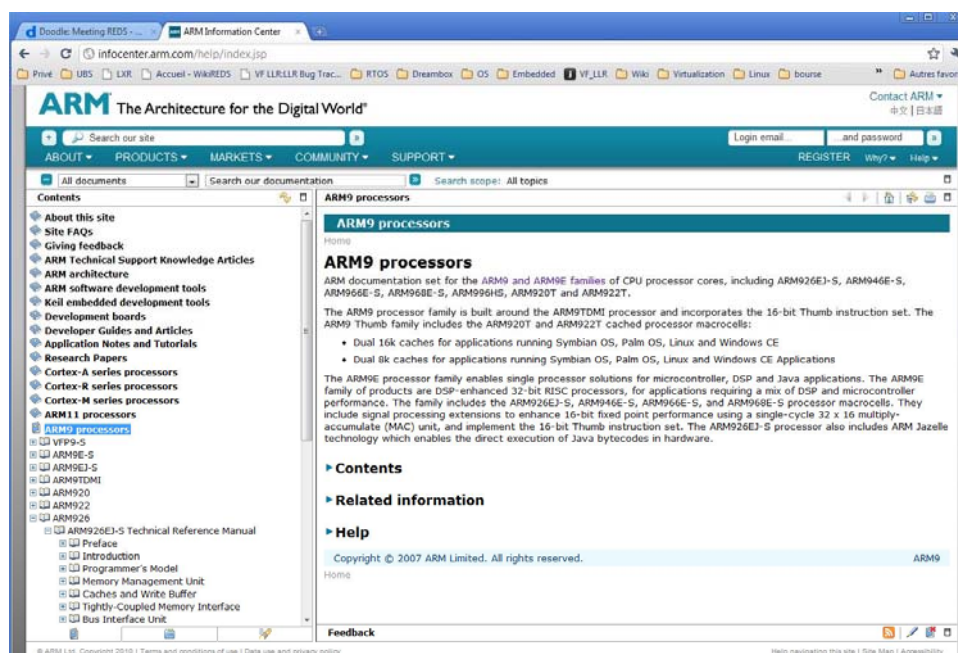
6

Cours ASM - Institut REDS/HEIG-VD

Toutes les instructions ARM sont codées sur 32 bits; cela comprend les différentes parties de l'instruction, à savoir le code opératoire (*opcode*), les opérandes, les *flags*, les valeurs immédiates (constantes), etc.

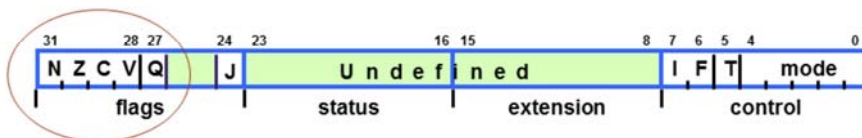
La majeure partie des instructions sont exécutées en un seul cycle (à l'exception des instructions de transferts et de quelques instructions de traitement particulières) et peuvent être exécutées de manière conditionnelle.

Un site de référence pour les processeurs ARM: <http://infocenter.arm.com>



Jeux d'instructions ARM (2/4)

- Code conditions
 - Si une instruction doit être effectuée sous certaines conditions, le *code de condition* fait partie de l'instruction
 - Les *codes de condition* portent sur les valeurs de certains bits du registre d'état (*flags*).



Les *flags* du registre d'état (CPSR) sont mis à jour lors d'une exécution d'une instruction arithmétique par exemple, ou lors d'une comparaison.

Toutefois, comme nous le verrons plus tard, une instruction arithmétique nécessite l'ajout d'un suffixe spécial (lettre 's') pour indiquer que la mise à jour doit s'effectuer.

Jeux d'instructions ARM (3/4)

<u>Suffixe</u>	<u>Flags de l'ALU</u>	<u>Signification</u>
EQ	Z set	Equal
NE	Z clear	Not equal
CS/HS	C set	Higher or same (unsigned >=)
CC/LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear and Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and differ	Signed <=

Par exemple, le code de condition **eq** est utilisé lorsqu'il y a égalité lors d'une comparaison. L'instruction de comparaison effectuant une soustraction, le résultat de cette soustraction aura pour effet de mettre le flag **Z** à 1 dans le registre CPSR, et donc l'instruction avec ce code de condition teste la valeur de ce flag. La valeur 1 correspond à "**Z set**" dans le tableau ("**Z clear**" signifie que le bit Z est à 0).

Jeux d'instructions ARM (4/4)

Key to Tables	
[cond]	Refer to Table Condition Field (cond)
<Opnd2>	Refer to Table Operand 2
<fields>	Refer to Table PSR fields
[S]	Updates condition flags if S present
C*, V*	Flag is unpredictable after these instructions in Architecture v4 and earlier
Q	Sticky flag. Always updates on overflow (no S option). Read and reset using MRS and MSR
X, Y	H meaning half-register [15:0], or T meaning [31:16]
<1mod_8r>	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits
<1mod_8*4>	A 10-bit constant, formed by left-shifting an 8-bit value by two bits

- Carte de référence rapide (extrait)

Arithmetic	Add	with carry saturating double saturating	SE	QDADD{cond} Rd, Rn, Rn	N Z C V	Rd := Rn + Opnd2 Rd := Rn + Opnd2 + Carry Q Rd := SAT(Rn + Rn) Q Rd := SAT(Rn + SAT(Rn * 2))	
Subtract	with carry reverse subtract reverse subtract with carry saturating double saturating	SE	QDSUB{cond} Rd, Rn, Rn	N Z C V	Rd := Rn - Opnd2 Rd := Rn - Opnd2 - NOT(Carry) Rd := Opnd2 - Rn Rd := Opnd2 - Rn - NOT(Carry)	No shift/rotate. No shift/rotate.	
Multiply	SE	QDML{cond} Rd, Rn, Rn	N Z C*	Rd := (Rn * Rn)[31:0]			

Addressing Mode 2 - Word and Unsigned Byte Data Transfer			
Pre-indexed	Immediate offset	[Rn], #<1mod_12>{1}	Equivalent to [Rn, #0]
	Zero offset	[Rn], #0	
	Register offset	[Rn], +/-Rn{1}	
	Scaled register offset	[Rn], +/-Rn, LSL #<1mod_5>{1}	
		[Rn], +/-Rn, LSR #<1mod_5>{1}	
Post-indexed	Immediate offset	[Rn], #<1mod_12>	
	Register offset	[Rn], +/-Rn	
	Scaled register offset	[Rn], +/-Rn, LSL #<1mod_5>	
		[Rn], +/-Rn, LSR #<1mod_5>	
		[Rn], +/-Rn, ASR #<1mod_5>	

ARM architecture versions	
n	ARM architecture version n and above.
nT	T variants of ARM architecture version n and above.
M	ARM architecture version 3M, and 4 and above excluding xM variants
nE	E variants of ARM architecture version n and above.

Operand 2		
Immediate value	#<1mod_8r>	
Logical shift left immediate	Rn, LSL #<1mod_5>	Allowed shifts 0-31
Logical shift right immediate	Rn, LSR #<1mod_5>	Allowed shifts 1-32
Arithmetic shift right immediate	Rn, ASR #<1mod_5>	Allowed shifts 1-32
Rotate right immediate	Rn, ROR #<1mod_5>	Allowed shifts 1-31
Register	Rn	
Rotate right extended	Rn, RRRX	

9

Cours ASM - Institut REDS/HEIG-VD

Tout comme pour l'assembleur x86, il existe des cartes de référence rapide présentant l'ensemble des instructions les plus importantes. Ces cartes sont très utiles lors du développement, en particulier avec l'assembleur ARM, car les instructions sont plutôt rigides au niveau des opérandes qu'elles manipulent. Le tableau indique de manière très rigoureuse le format des opérandes, modes d'adressage et les différentes formes des instructions qui peuvent être validés par le compilateur.

Dans ce sens, l'assembleur ARM peut être considéré comme plus contraignant que l'assembleur x86.

Instructions de traitement x86 (1/6)

- **Opérandes**

- **Registre**

- Précédé avec un %
 - `%eax, %ebx, %ecx, %edx, %eax, %eax, ...`

- **Valeur immédiate**

- Précédée avec un \$
 - `$4, $0x98, $0b10110, $msg, ...`

nom d'un label par exemple

- Taille des opérandes définie avec le dernier caractère de l'instruction

- **b** indique une taille de **8 bits**
 - **w** indique une taille de **16 bits**
 - **l** indique une taille de **32 bits**

Lorsqu'une instruction peut contenir un registre comme opérande, celui-ci doit être préfixé avec %

La valeur immédiate représente une valeur numérique qui sera encodée avec l'instruction. Cette valeur doit être préfixée avec le symbole \$

Instructions de traitement x86 (2/6)

- `mov <src>, <dst>`
- La donnée est **copiée de <src> vers <dst>**
 - L'instruction correspond à une **affectation**.
 - La valeur dans source est **inchangée**.
- `mov $250, %eax` # `eax = 250`
- `mov $0x25ab, %ecx` # `ecx = 0x25ab`
- `mov $0x10, %cl` # `cl = 0x10, ecx = ?`
- `mov %eax, %ebx`
- `mov $msg, %eax`
- `movb $25, %ah` # 8 bits
- `movw %ax, %bx` # 16 bits
- `movl $0xab1324ff, %eax` # 32 bits

La taille des opérandes peut être intégrée à l'instruction à l'aide d'un suffixe (cf ci-dessus). S'il n'y a pas d'ambiguïté, il n'est en principe pas nécessaire de rajouter le suffixe, bien que le rajout soit vivement recommandé pour faciliter la lecture. Normalement, il n'y a pas d'ambiguïté dans le cas de transfert mémoire (nous verrons les instructions de transfert dans un autre chapitre).

Instructions de traitement x86 (3/6)

- Conversion de données avec des tailles différentes

- `mov $0xffe8, %ax` # 0xffe8 = -24 ou 65'512 ?
- `movswl %ax, %ebx` # ebx = ?
- `movzwl %ax, %eax` # eax = ?
- `movsbl %ah, %ecx` # byte -> dword
- `movsbw %al, %ax` # byte -> word
- `movzbw %ch, %dx`
- `movzwl %dx, %eax` # word -> dword

- La conversion requiert des opérandes de type registre.

12

Cours ASM - Institut REDS/HEIG-VD

L'instruction *mov* peut être utilisé avec un suffixe composé de trois lettres permettant de convertir la taille d'une donnée à une **taille supérieure**. Ce type d'instruction nécessite l'utilisation de deux opérandes de type registre.

La première lettre du suffixe détermine s'il faut considérer le signe de la valeur source, *i.e.* s'il faut tenir compte de son bit de poids fort ou non, durant sa conversion. Si la valeur est négative, la valeur convertie à la nouvelle taille conservera le signe négatif et sa représentation sera conforme à une valeur négative. Si la valeur est positif, les bits de poids fort de la valeur cible seront nuls. La lettre **s** (*signed-extended*) signifie que l'on souhaite préserver le signe, alors que la lettre **z** (*zero-extended*) considère une valeur non signée et les bits de poids fort vaudront 0.

Les deux lettres suivantes indiquent le type de conversion; **bw** signifie *byte-to-word* (8-to-16), **bl** *byte-to-dword* (8-to-32), **wl** *word-to-dword* (16-to-32), etc.

Instructions de traitement x86 (4/6)

- **add** <src>, <dst> # <dst> = <dst> + <src>
- **addl** \$25, %eax # eax = eax + 25
- **addl** %ebx, %ecx # ecx = ecx + ebx
- **subb** \$0x10, %ah # ah = ah - 0x10
- **subl** \$0x10, %ah # ah = ah - 0x10
- **and** \$0x80000, %eax # eax = eax & 0x80000
- **shll** \$20, %ecx # ecx = ecx * 2²⁰
- **shrl** \$10, %edx # edx = edx / 2¹⁰
- **cld** # *Clear direction flag*
- **std** # *Set direction flag*

13

Cours ASM - Institut REDS/HEIG-VD

Les exemples ci-dessus montrent l'utilisation d'instructions de traitement logiques. La taille des opérandes peut être également explicite ou implicite.

Les instructions *cld* et *std* sont des exemples d'instructions qui agissent uniquement sur le registre d'état *eflags*. Dans le cas présent, elles touchent uniquement le bit de direction.

Ces deux instructions peuvent donc être utilisées conjointement avec d'autres instructions qui se réfèrent à ce *flag* de direction; nous découvrirons plus tard que c'est le cas, par exemple, pour une instruction permettant la copie rapide d'un bloc mémoire, le *flag* de direction indiquant alors si les adresses doivent être incrémentées ou décrémentées (sens de la copie).

Instructions de traitement x86 (5/6)

- `cmp <src2>, <src1>`
- Instruction de comparaison
 - $\text{<src1>} - \text{<src2>}$
- Résultat dans le registre d'état *eflags*
- Généralement suivi d'une instruction de branchement conditionnel

L'instruction de comparaison ***cmp*** permet de comparer les valeurs contenues dans deux registres, qui constituent les deux opérandes de l'instruction. Il s'agit en fait d'une soustraction dont le résultat sera évalué à l'aide du registre *eflags*.

Les codes de condition seront étudiés avec les instructions de branchement, dans le chapitre suivant.

Instructions de traitement x86 (6/6)

- Soit les variables x, y, z associées aux registres eax, ebx, ecx respectivement.

⇒ Codez en assembleur les opérations ci-dessous. Aidez-vous des opérateurs de décalage arithmétique.

```
# z = x + y + 4
```

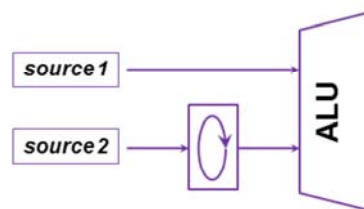
```
# z = 3x - 5
```

```
# z = (x + y) / 4
```

```
# z = 4x + 3
```

Instructions de traitement ARM (1/9)

- Manipulation des **registres**
- Instructions avec 0, 1, 2 ou 3 opérandes
 - *<destination>*, *<source 1>*, *<source 2>*
 - Registres, valeurs immédiates
 - Encodage dans l'instruction (32 bits)
- La dernière opérande *<source 2>* est un **registre à décalage**.
 - Une opération de *décalage/rotation* est possible sur cette opérande.
 - Ne s'applique pas sur une valeur immédiate



16

Cours ASM - Institut REDS/HEIG-VD

Comme nous l'avons vu précédemment, les instructions de **traitement** permettent de manipuler des données au niveau de **registres**. Alors que sur *x86*, les mnémoniques correspondant aux instructions de traitement pourront aussi être utilisés pour les instructions de transfert, ce n'est pas le cas sur *ARM*. Les instructions de traitement sont uniquement dédiées aux opérandes de type registre.

Une particularité des instructions de traitement sur *ARM* porte sur l'utilisation de la deuxième opérande : cette opérande est associée à un registre de décalage permettant d'effectuer un décalage ou une rotation de bits sur cette opérande.

Une opérande peut être soit un registre, soit une valeur numérique appelée **valeur immédiate**.

Une valeur immédiate ne peut être utilisée que sur la dernière opérande de l'instruction.

Les opérateurs de décalages **ne s'appliquent pas** sur une valeur immédiate.

Instructions de traitement ARM (2/9)

- **Valeur immédiate**
- Encodage dans l'instruction (32 bits)
 - Ne peut pas valoir n'importe quelle valeur!
- Codage sur **12 bits** seulement selon la formule suivante

n (4 bits)

val (8 bits)

$$\text{Valeur immédiate} = \text{val} * 2^n$$

avec val = 0 à 255, décalé de 0, 2, 4, ... ou 30 bits à gauche

- L'encodage de la valeur sur 8 bits peut être signé selon le contexte.
- Une valeur immédiate est préfixée avec #
 - #34
 - #0xc0000000
 - #-56

17

Cours ASM - Institut REDS/HEIG-VD

Nous avons vu qu'une instruction ARM est codée sur 32 bits, comprenant tout ce qui caractérise l'instruction (*opcode*, opérandes, valeur immédiate, etc.). Cette approche ne permet donc pas d'utiliser une valeur immédiate sur 32 bits, mais nécessite un codage particulier de celle-ci.

Seul 12 bits sont réservés pour la valeur immédiate. Avec 12 bits, il est possible d'exprimer une valeur dans un intervalle de 0 à $2^{12} - 1$ (4095); ce qui serait un peu limitatif. C'est pourquoi les 12 bits sont utilisés d'une autre manière: on utilise les 8 bits de poids faible pour coder une valeur entre 0 et 255, puis les 4 bits de poids fort pour coder un exposant. Ainsi une valeur immédiate (*imm*) est calculée selon la formule suivante: $\text{imm} = \text{val}(8 \text{ bits}) * 2^{2^n}$, n étant l'exposant.

De cette manière, une valeur immédiate peut prendre les valeurs entre 0 et 255, puis tous les nombres qui peuvent s'exprimer comme étant le décalage de 2 bits d'une valeur entre 0 et 255.

- Si n vaut 0, on obtient toutes les valeurs entre 0 et 255.
- Si n vaut 1, on obtient toutes les valeurs entre 0 et 255 multipliées par 4
- Si n vaut 2, on obtient toutes les valeurs entre 0 et 255 multipliées par 16
- Si n vaut 3, on obtient toutes les valeurs entre 0 et 255 multipliées par 64

...et ainsi de suite. On voit donc apparaître les multiples de 4, 16, 64, etc. (jusqu'à la valeur maximale donnée par le facteur 255).

Cela permet ainsi de donner une meilleure *amplitude* de la valeur immédiate.

Lorsqu'il n'est pas possible de coder une valeur immédiate sous cette forme, il faut utiliser une autre approche, détaillée ci-après et dans le chapitre suivant.

Instructions de traitement ARM (3/9)

- **mov** <dst>, <src> @ <dst> = <src>
- **Copie** de la valeur de <src> vers <dst>
 - L'instruction correspond à une **affectation**.
 - La valeur dans <src> est **inchangée**.
 - **mov** r1, r3 @ r1 = r3
- **movw, movt**
 - Disponibles sur coeurs ARM avec une version supérieure ou égale à v7

```
movw r1, #0x12ab @ r1 = 0x12ab
movt r1, #0x56ff @ r1 = 0x56ff12ab
```

- **Registres d'état**
 - Instructions *priviliégées*
 - **mrs** r2, cpsr @ r2 = cpsr
 - **msr** spsr, r2 @ spsr = r2
- **Exécution conditionnelle** de toutes les instructions
 - Suffixe de 2 lettres

18

Cours ASM - Institut REDS/HEIG-VD

Une différence essentielle avec la norme AT&T de l'assembleur x86 réside dans le sens de fonctionnement des opérandes : alors que l'opérande source apparaît en second (à droite de la virgule) sur x86, elle apparaît toujours en premier (à gauche de la virgule) en assembleur ARM. A l'instar du x86, l'instruction *mov* permet de copier une valeur dans un registre. L'opérande source peut être soit un registre, soit une valeur immédiate. Nous avons découvert que dans ce dernier cas, la valeur immédiate est encodée d'une certaine manière. Depuis les version v7 des architectures ARM, deux nouvelles instructions ont fait leur apparition : il s'agit de *movw* et *movt* qui permettent de stocker une valeur quelconque sur 32 bits. Ces deux instructions prennent un registre commun comme première opérande, puis une valeur immédiate sur 16 bits comme seconde opérande. Elles doivent être exécutées dans cet ordre (*movw*, *movt*) et ne pas être "*trop distantes*" l'une de l'autre (en principe, les deux instructions se suivent). La première instruction permet de stocker les 16 bits de poids faible de la valeur, alors que la seconde les 16 bits de poids fort; cette dernière opère un décalage de 16 bits à gauche d'une valeur immédiate de 16 bits.

Il existe une autre manière générale de manipuler des valeurs 32 bits que nous étudierons dans le chapitre portant sur les instructions de transfert.

Une autre différence majeure porte sur l'**exécution conditionnelle** des instructions ARM. En effet, un code de condition sur deux lettres peut être utilisé pour décider si l'instruction doit être exécuté ou non. Bien entendu, ce code de condition se basera sur l'état des *flags* du registre d'état (registre *cpsr*).

Instructions de traitement ARM (4/9)

- **add** r0, r1, #5 @ r0 = r1 + 5
- **adds** r4, r4, #0x20 @ r4 = r4 + 0x20
- **subs** r0, r1, #5
- **subeq** r5, r5, r6
- **add** r0, r1, r2 @ r0 = r1 + r2
- **mov** r1, r3, **lsl #4** @ r1 = r3 • 2⁴
- **addeqs** r6, r7, r2, lsr #3 @ r6 = r7 + (r2 / 2³)

Valeur immédiate

Mise à jour automatique des *flags*

Exécution conditionnelle

Décalage à gauche de 4 bits

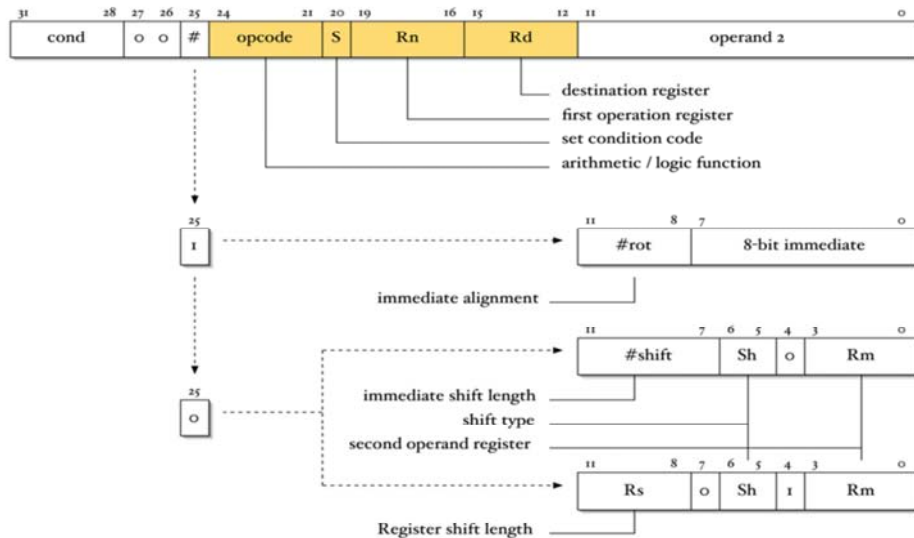
Dans les exemples ci-dessus, on voit qu'une instruction ARM peut se décliner différemment (différents mnémoniques) selon que l'on veut une mise à jour automatique des *flags*, ou une exécution conditionnelle de l'instruction.

Dans le premier cas, l'instruction *adds* aura pour effet de mettre à jours les *flags* du *cpsr* en fonction du résultat de l'addition (un résultat nul mettra le bit **Z** à 1 (*Z set*)).

L'exécution conditionnelle de l'instruction *sub* dépendra des *flags* du *cpsr*. Ainsi, dans cet exemple, en fonction du résultat de la première soustraction, la seconde soustraction sera effectuée **seulement si le bit Z est à 1** (donc résultat nul de la première instruction).

Instructions de traitement ARM (5/9)

- Format des instructions



20

Cours ASM - Institut REDS/HEIG-VD

L'encodage d'une instruction *ARM* est strictement limité à 32 bits (4 octets), contrairement aux instructions *x86* (nous ne considérons pas ici le mode *Thumb* du processeur *ARM* qui possède des instructions de 16 bits).

L'encodage présenté ci-dessus montre bien un champ de bits servant à stocker le **code de condition** (s'il y en a un), le code opératoire (**opcode**), **deux registres** pouvant faire partie des opérandes, ainsi qu'un champ réservé pour le stockage d'une **valeur immédiate**.

Instruction de traitement ARM (6/9)

- `cmp r0, r1`
 - Comparaison de `r0` avec `r1` (`r0 - r1`)
- L'instruction comporte 2 opérandes.
- Résultat dans le registre d'état *cpsr*
- L'exécution de *cmp* peut être aussi conditionnelle.
 - `cmpeq r0, r1`

Comme sur *x86*, l'instruction de comparaison effectue une soustraction entre les deux opérandes et le résultat de la comparaison est caractérisé par les *flags* du registre d'état *cpsr*.

Instruction de traitement ARM (7/9)

Opcode[24:21]	Mnemonic	Signification	Effet
0000	and	Logical bit-wise AND	$Rd := Rn \text{ AND } Op2$
0001	eor	Logical bit-wise exclusive OR	$Rd := Rn \text{ EOR } Op2$
0010	sub	Subtract	$Rd := Rn - Op2$
0011	rsb	Reverse subtract	$Rd := Op2 - Rn$
0100	add	Add	$Rd := Rn + Op2$
0101	adc	Add with carry	$Rd := Rn + Op2 + C$
0110	sbc	Subtract with carry	$Rd := Rn - Op2 + C - 1$
0111	rsc	Reverse subtract with carry	$Rd := Op2 - Rn + C - 1$
1000	tst	Test	Scc on $Rn \text{ AND } Op2$
1001	teq	Test equivalence	Scc on $Rn \text{ EOR } Op2$
1010	cmp	Compare	Scc on $Rn - Op2$
1011	cmn	Compare negated	Scc on $Rn + Op2$
1100	orr	Logical bit-wise OR	$Rd := Rn \text{ OR } Op2$
1101	mov	Move	$Rd := Op2$
1110	bic	Bit clear	$Rd := Rn \text{ AND NOT } Op2$
1111	mvn	Move negated	$Rd := \text{NOT } Op2$

22

Cours ASM - Institut REDS/HEIG-VD

Sur ARM, il existe également une instruction de multiplication: *mul Rd, Rm, Rs*

Cette instruction est particulière et son implémentation matérielle peut varier d'une version du processeur à l'autre. Elle est plus complexe et nécessite plusieurs cycles.

Les registres des opérandes doivent être obligatoirement différents les uns des autres (il n'est pas possible d'effectuer: *mul r2, r2, r3* par exemple. De plus, la dernière opérande **ne peut pas** être une valeur immédiate.

Instruction de traitement ARM (8/9)

- Commentez le programme ci-dessous et dessinez l'organigramme correspondant.

```
cmp r5, #5  
  
cmpne r4, #-5  
  
addeq r0, r1, r1, lsl #2  
  
cmp r0, #0  
  
rsbmi r0, r0, #0
```

Instruction de traitement ARM (9/9)

- Soit les registres suivants associés aux variables x, y et z respectivement: $r4$, $r5$ et $r6$. Aidez-vous des opérateurs de décalage arithmétique.

⇒ Codez en assembleur les opérations ci-dessous.

$$@ z = x + y + 4$$

$$@ z = 3x - 5$$

$$@ z = (x + y) / 4$$

$$@ z = 4x + 3$$

Macro-instructions (1/4)

- Bloc d'instructions associé à un nom symbolique
 - Début d'une macro : `.macro`
 - Fin de la macro : `.endm`
 - Une sortie prématurée de la macro est possible avec `.exitm`
- Possibilité de passer des arguments
- **Attention ! Ce n'est pas une fonction !**
 - Traitement purement textuel
 - Pris en charge par le préprocesseur

Une macro-instruction est la description d'une inscription *logique* (qui n'existe pas du tout au niveau du processeur) à partir d'instructions assembleur.

C'est le préprocesseur qui remplacera **chaque occurrence** de l'instruction logique (définie par un nom symbolique) avec le contenu correspondant. Il s'agit donc bien d'une manipulation du code source effectuée par le préprocesseur.

Macro-instructions (2/4)

```
.macro macroABC Param1,Param2

    ...                @ code (corps de la macro)
                        @ On se référera aux paramètres avec \
                        @ (exemple: \Param1)

    mov r0, #\Param2
    ...
    add r7, r0, \Param1
    ...

.endm

macroABC r0, 20
```

26

Cours ASM - Institut REDS/HEIG-VD

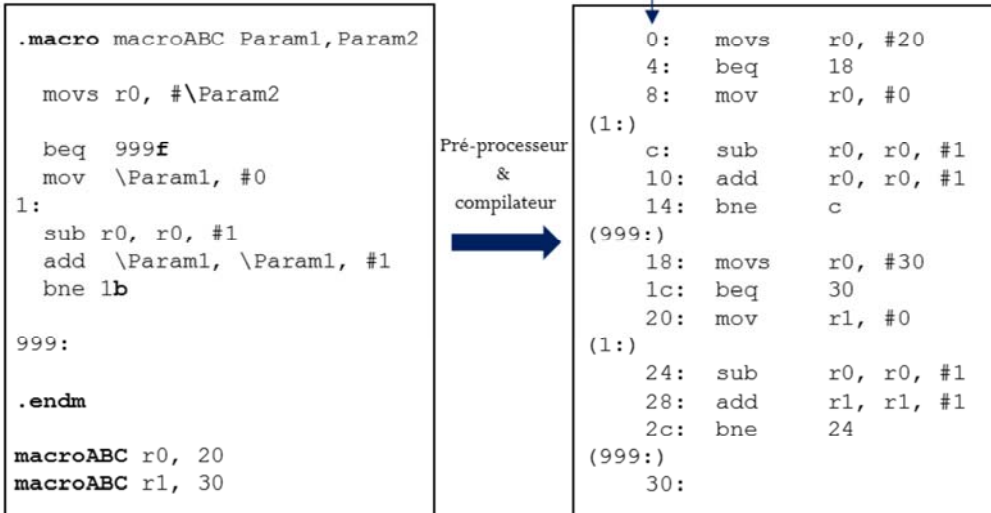
Des arguments peuvent être passés à la macro-instruction. Dans ce cas, chaque occurrence faisant référence aux arguments sera remplacé par l'argument effectif correspondant.

Les arguments sont préfixés avec le symbole "\" dans le corps de la macro.

Lorsqu'on utilise une macro-instruction, et que celle-ci est utilisée à différents endroits dans le code source, il est toujours très important de "visualiser" le texte en sortie avant compilation. Si des *labels* sont utilisés dans le corps de la macro, ces *labels* apparaîtront à plusieurs reprises. Cela est permis sous certaines conditions (**labels numériques**) et il faut être prudent lorsque l'on s'y réfère.

Macro-instructions (3/4)

- Etiquettes (*labels*) numériques



27

Cours ASM - Institut REDS/HEIG-VD

Le code de gauche est traduit par le préprocesseur et le compilateur vers le code de droite. Dans le code source, on remarque deux appels successifs à la macro *macroABC*, ce qui a comme conséquence l'apparition de deux fois le *label 1:* et deux fois le *label 999:* dans le code produit.

Il faut donc indiquer au compilateur à quel *label* l'instruction se réfère, soit le premier qui apparaît en reculant (*backward*), soit celui qui apparaît en avant (*forward*). L'utilisation des lettres **b** et **f** accolées directement au *label* en suffixe permet de définir la direction dans laquelle rechercher le *label*.

Macro-instructions (3/4)



- Ecrire en assembleur `x86` une macro-instruction qui stocke dans un registre la multiplication de deux entiers 32 bits ($x*y$), qui incrémente x et décrémente y
- Utilisez une fois cette macro une fois dans un programme.

Macro-instructions (4/4)



- Ecrire une macro-instruction *compare* en assembleur *ARM* qui accepte trois paramètres (*<src>*, *<dest>*, *<cond>*), et qui compare *<src>* avec *<dest>* selon la condition *<cond>*. Si la condition est satisfaite, on souhaite additionner *<dest>* à *<src>* augmenté de 20 Mo.

Références

- ARM Infocenter. Site ARM principal : <http://infocenter.arm.com>
- Intel x86 32 & 64 bits.
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- Rajat Moona, *Assembly Language Programming in GNU/Linux for IA32 Architectures*, Eastern Economy Edition, New Dehli, 2009.
- Andrew N.Sloss, Dominic Symes, *ARM System Developer's Guide, Designing and Optimizing System Software*, Elsevier, 2004.