Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

a.11 - Jeu d'instructions du µP ARM

Classes T-2/I-2 // 2018-2019

Daniel Gachet | HEIA-FR/TIC

a.11 | 16.11.2018

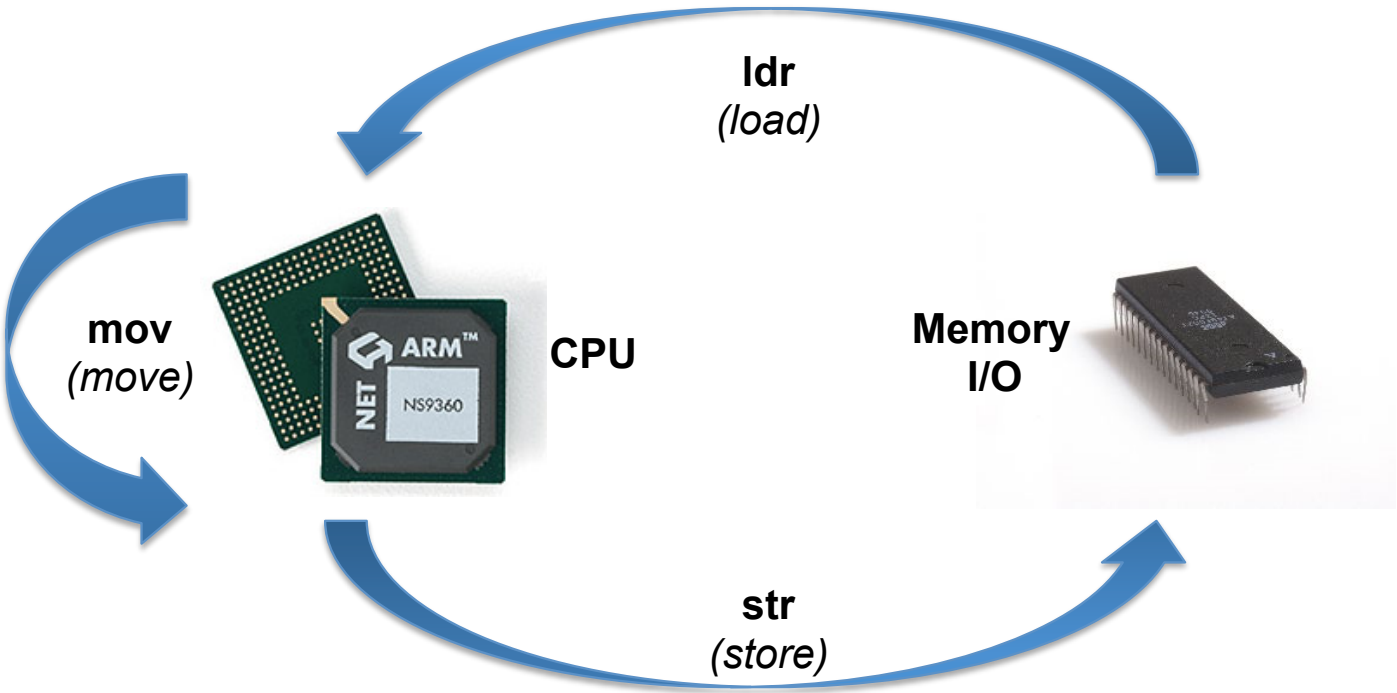
Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

Contenu

- Les instructions de transfert
  Load and Store instruction
- Les instructions pour les opérations arithmétiques et logiques
  Arithmetical and logical instructions
- Les instructions pour des boucles et des branchements
  Loop and branch instructions

Instruction de transfert

- On trouve 3 familles d'instructions
  - *Load* pour les transferts de données de la mémoire vers le CPU (registres Rx)
  - *Move* pour les transferts de données à l'intérieur du CPU (entre registres Rx)
  - *Store* pour les transferts de données du CPU (registres Rx) vers la mémoire



MOV Instructions

- MOV (move) place une donnée dans le registre destination. La donnée peut être une valeur immédiate ou une donnée contenue dans un autre registre. Elle peut être décalée (shifted) avant d'être stockée.

```
MOV{<cond>}{S} <Rd>, <shifter_operand>
```

- Utilisation
  - copie d'une donnée d'un registre vers un autre

```
mov r1, r4          // copie de la donnée contenue dans le
                    // registre R4 dans le registre R1
```

  - décalage ou rotation sur un registre

```
mov r1, r1, lsl #5 // décale sur la gauche de 5 bits la donnée
                   // contenue dans le registre R1
```

  - retour de sous-routine en copiant l'adresse de retour (LR/R14) dans le compteur ordinal (PC/R15)

```
mov pc, lr          // retour de sous-routine si l'adresse a été
                    // précédemment sauvée dans le registre LR
```

”move”, d’autres instructions…

MVN	Copie le complément à 1 de la donnée (inversion binaire de la valeur)	MVN{<cond>}{S} <Rd>, <shifter_operand>
MRS	Copie la donnée d’un registre de statut dans un registre général	MRS{<cond>} <Rd>, CPSR MRS{<cond>} <Rd>, SPSR
MSR	Copie la donnée d’un registre général dans les champs (c, x, s, f) d’un registre de statut (CPSR current ou SPSR saved)	MSR{<cond>} CPSR_<fields>, #<immediate> MSR{<cond>} CPSR_<fields>, <Rm> MSR{<cond>} SPSR_<fields>, #<immediate> MSR{<cond>} SPSR_<fields>, <Rm>

<fields> :

c = control (bits 0..7)

x = extended (bits 8..15)

s = status (bits 16..23)

f = flags (bits 24..31)

MSR/MRS, un exemple...

- Instructions forçant le processeur en mode user (bit 4 – bit 0) du CPSR (0b10000)

```
mrs  r0,cpsr      // read CPSR
bic  r0,r0,#0xf   // modify by removing current mode
msr  cpsr_c,r0    // write the result back to CPSR
```

LDR Instruction

- LDR (load) charge le contenue d'une cellule mémoire à une adresse donnée dans un registre général du CPU

```
LDR{<cond>} <Rd>, <addressing_mode>
```

- Utilisation
  - copie d'une constante dans un registre interne

```
ldr r0,=0x1201      // copie la constante 0x1201 dans le registre R0
```

  - copie d'une adresse dans un registre interne

```
ldr r4,=var         // copie de la valeur de l'adresse de la
                    // variable var dans le registre R4
```

  - copie d'une donnée stockée en mémoire dans un registre général

```
ldr r1,[r4]         // copie de la donnée stockée en mémoire à
                    // l'adresse contenue dans le registre R4 (var)
                    // dans le registre R1
```

”load”, une autre instruction...

LDM	Transferts multiples de la mémoire vers les registres. Utilisé pour restaurer de manière contiguë le contenu de plusieurs registres préalablement stocké en mémoire.	LDM{<cond>}<addr_mode> <Rn>{!}, <registers>

```
dest :    .long  101,102,103,104,105,106
          ldr    r9,=dest
          ldmia  r9!,{r1-r6}
```

Modes d’adressage :

- IA → increment after
- IB → increment before
- DA → decrement after
- DB → decrement before

! → Mise à jour du registre <Rn> après opération

STR Instruction

- STR (store) transfert le contenu d'un registre général du CPU vers une cellule mémoire à une adresse donnée

```
STR{<cond>} <Rd>, <addressing_mode>
```

- Utilisation
  - copie d'une donnée d'un registre général vers la mémoire

```
str r1,[r4]        // copie de la donnée contenue dans le
                   // registre R1 vers l'adresse contenue
                   // dans le registre R4
```

  - stockage du PC (R15) comme adresse relative en mémoire

```
str pc,[r3]        // copie l'adresse de PC, vers la position
                   // mémoire contenue dans le registre R3
```

”store”, une autre instruction...

STM	Transferts multiples des registres vers la mémoire. Utilisée pour sauvegarder de manière contiguë le contenu de plusieurs registres dans la mémoire.	STM{<cond>}<addr_mode> <Rn>{!}, <registers>

```
dest :  .space  16*4
        ldr     r1,=1
        ldr     r2,=2
        ldr     r3,=3
        ldr     r4,=4
        ldr     r9,=dest
        stmia   r9!,{r1-r4}
```

Modes d’adressage :

- IA → increment after
- IB → increment before
- DA → decrement after
- DB → decrement before

! → Mise à jour du registre <Rn> après opération

Instructions arithmétiques et logiques

- Les instructions arithmétiques et logiques sont groupées en plusieurs familles distinctes
  - Les instructions arithmétiques (addition, soustraction et multiplication)
  - Les instructions logiques (ET, OU, OU exclusif, test)
  - Les instructions de comparaison
- Format de ces opérations

```
<opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

  - `<opcode>:` opération (mov, add, sub,…)
  - `{<cond>}` : condition d'exécution, optionnel
  - `{S}` : mise à jour des fanions après exécution, optionnel
  - `<Rd>:` 1ère opérande -> registre de destination
  - `<Rn>:` 2ème opérande
  - `<shifter_operand>:` 3ème opérande

Opérations arithmétiques

ADD	Addition de deux données (Rd = Rn + shifter_operand)	ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
ADC	Addition de deux données en tenant compte du carry	ADC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
SUB	Soustraction de deux données (Rd = Rn - shifter_operand)	SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
SBC	Soustraction de deux données en tenant compte du carry	SBC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
RSB	Soustraction inverse de deux données (Rd = shifter_operand - Rn)	RSB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
RSC	Soustraction inverse de deux données en tenant compte du carry	RSC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
        ldr     r0, =3583
        ldr     r1, =7620
        add     r2, r0, r1
```

→ R2 = R0 + R1 = 3583 + 7620

Addition avec Carry

- Une addition avec carry permet de réaliser une addition sur 64 bits

R1	0x00000000	R0	0xffffffff
R3	0x00000000	R2	0x00000001
R5	0x00000001	R4	0x00000000

```
        ldr     r0,=0xffffffff
        ldr     r1,=0x0
        ldr     r2,=0x1
        ldr     r3,=0x0
        adds    r4,r0,r2
        adc     r5,r1,r3
```

Soustraction et soustraction inverse

- L'opération de soustraction a un sens :
  Rd = Rn – shitfer_operand

```
ldr     r0, =50
ldr     r1, =24
sub     r2, r0, r1
```

→R2 = R0 – R1 = 50 – 24 = 26

- L'opération de soustraction inverse change le sens des opérandes :
  Rd = shifter_operand – Rn

```
ldr     r0, =50
rsb     r2, r0, #24
```

→R2 = 24 – R0 = 24 – 50 = -26

- La soustraction inverse permet d'obtenir le complément à 2 d'une valeur contenue dans un registre

```
rsb     r2, #0
```

→R2 = 0 – R2 = -R2

Opérations de multiplication

MUL	Multiplication 32 bits x 32 bits → 32 bits	MUL <Rd>, <Rm>, <Rs> →Rd = Rm * Rs
MLA	Multiplication avec un accumulateur 32 bits x 32 bits → 32 bits + acc	MLA <Rd>, <Rm>, <Rs>, <Rn> → Rd = Rm * Rs + Rn
SMULL	Multiplication en valeurs signées 32 bits signés x 32 bits signés → 64 bits signés	SMULL <RdLo>, <RdHi>, <Rm>, <Rs> → RdLo = lower (Rm * Rs) → bits 31-0 → RdHi = upper (Rm * Rs) → bits 63-32
UMULL	Multiplication en valeurs non-signés 32 bits x 32 bits → 64 bits	UMULL <RdLo>, <RdHi>, <Rm>, <Rs> → RdLo = lower (Rm * Rs) → bits 31-0 → RdHi = upper (Rm * Rs) → bits 63-32

- Hormis ces quatre instructions principales, le processeur implémente également d'autres opérations de multiplication

Opérations logiques / booléennes

AND	Réalise une opération ET logique bit à bit entre deux opérandes	AND{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
EOR	Réalise une opération OU-Exclusif logique bit à bit entre deux opérandes	EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
ORR	Réalise une opération OU logique bit à bit entre deux opérandes	ORR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
BIC	Réalise un ET logique bit à bit entre une valeur et le complément à 1 de la seconde	BIC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
        ldr     r0, =0x11
        ldr     r1, =0x22
        ldr     r2, =0x44
        ldr     r3, =0x88
        and     r0, r0, #0x0f
        orr     r1, r1, #0x88
        eor     r2, r2, #0xf0
        bic     r3, r3, #0xf0
```

CMP Instruction

- CMP (compare) compare deux données. La première donnée vient d'un registre. La deuxième peut être une valeur immédiate, une donnée contenue dans un registre ou le résultat d'un décalage avant la comparaison. CMP met à jour les condition flags à partir du résultat de la soustraction de la deuxième donnée à la première.

```
CMP{<cond>} <Rn>, <shifter_operand>
        → <Rn> - <shifter_operand>
```

- Utilisation
  - comparaison d'un registre avec une valeur immédiate

```
cmp r1, #35        // compare la donnée contenue dans le
                   // registre R1 avec la valeur 35
```

  - comparaison de deux données contenues dans deux registres

```
cmp r1, r3         // compare les données contenues dans les
                   // registres R1 et R3
```

CMN Instruction

- CMN (Compare Negative) compare une donnée avec le complément à 2 d'une deuxième donnée. La première donnée vient d'un registre. La seconde peut être une valeur immédiate, une donnée contenue dans un registre ou le résultat d'un décalage préalable. CMN met à jour les condition flags à partir du résultat de l'addition des deux données.

```
CMN{<cond>} <Rn>, <shifter_operand>
    → <Rn> - (0 - <shifter_operand>)
        → <Rn> + <shifter_operand>
```

- Utilisation
  - compare avec le complément à 2 d'une donnée

```
cmn r1, #35        // compare la donnée contenue dans le
                   // registre R1 avec la valeur -35
```

Autres instructions de comparaison...

TST	Compare deux données et met à jour les flags sur la base d'une opération AND logique	TST{<cond>}{S} <Rn>, <shifter_operand>
TEQ	Compare deux données et met à jour les flags sur la base d'une opération EOR (ou exclusif)	TEQ{<cond>}{S} <Rn>, <shifter_operand>

- TST peut être utilisé afin de déterminer si au moins un bit contenu dans une série est à un.
- Exemple

```
        ldr     r0, =0x53
        ldr     r1, =0x04
        tst     r0, r1
```

Opérations de branchement

B	Branchement par offset, valeur immédiate de 24 bits signés → PC = PC + offset « 2 + 8 → ±32MB	B <offset_24>	loop : … … b loop
BX	Branchement direct par registre (p. ex. pour le retour de sous-routines) → PC = Rm & 0xffff'fffe	BX <Rm>	routine : … … bx lr
BL	Branchement par offset et sauvetage de l'adresse de la prochaine instruction à exécuter (adresse de retour), pour appel de sous-routines → LR = PC + 4 → PC = PC + offset « 2 + 8 → ±32MB	BL <offset_24>	… … … bl routine
BLX	Branchement et sauvetage de la prochaine adresse à exécuter, pour appel indirect de sous-routines → LR = PC + 4 → PC = Rm & 0xffff'fffe	BLX <Rm>	ldr r1,=routine … … blx r1

- Chaque branchement peut s'exécuter de manière conditionnelle

Branchements conditionnels

- Pour effectuer un branchement conditionnel, il suffit d'ajouter les 2 caractères de la condition à l'opération de branchement

`B<cc> label` ou `BLX<cc> R1`

Mnemonic		Description
AL		Always (AL normally omitted)
EQ	(Z==1)	Equal
NE	(Z==0)	Not equal
CS	(C==1)	Carry Set
CC	(C==0)	Carry Clear
MI	(N==1)	Negative (minus)
PL	(N==0)	Positive or zero (plus)
VS	(V==1)	Overflow
VC	(V==0)	No overflow

Mnemonic		Description
HI	>	Unsigned higher
HS	>=	Unsigned higher or same
LS	<=	Unsigned lower or same
LO	<	Unsigned lower

Mnemonic		Description
GT	>	Signed greater than
GE	>=	Signed greater than or equal
LE	<=	Signed less than or equal
LT	<	Signed less than

Exemple de boucle

```c
// C-code
#define CONSTANT 10

int sum = CONSTANT;

for (int i=0; i<CONSTANT; i++) {
        sum += i;
}
```

```
// Assembler-code
        ldr     r3, =10
        ldr     r2, =0
        b       test
loop :  add     r3, r3, r2
        add     r2,#1
test :  cmp     r2,#10
        blo     loop
```