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

HTA-FR – Kurs Telekommunikation

Verfasser:
D. Gachet / HTA-FR - Telekommunikation

Embedded systems 1

Befehlssatz des ARM-Prozessors

Klasse T-2 // 2018-2019

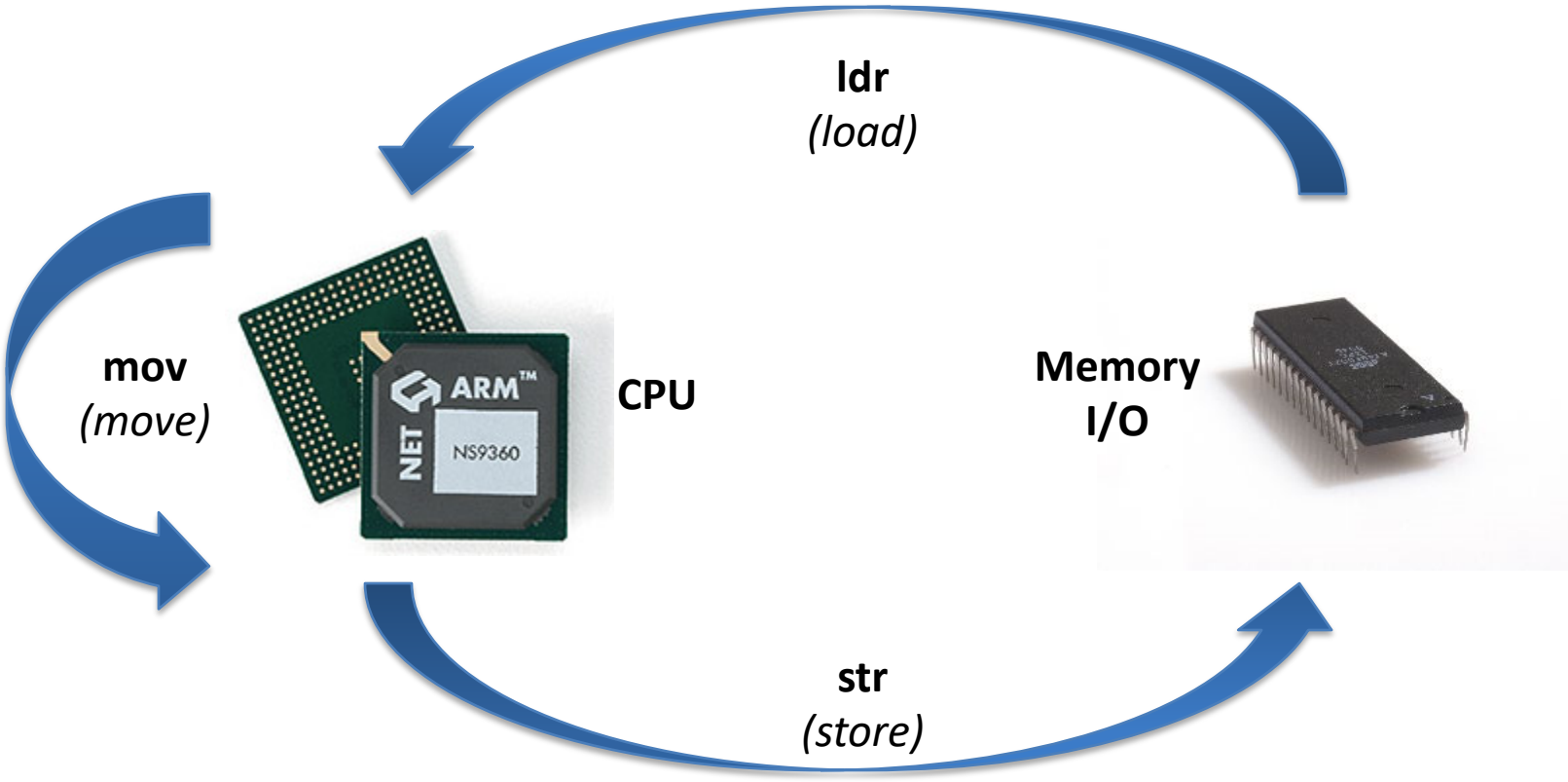
Inhalt

- **Transferbefehle**
  Load and Store instruction
- **Befehle für arithmetische und logische Operationen**
  Arthmetical and logical instructions
- **Schleifen- und Verzweigungsbefehle**
  Loops and branches instructions

Transferbefehle

Es sind 3 Befehlsfamilien vorhanden:

- ***Load* für die Übertragungen von Werten aus dem Speicher zur CPU (Rx-Register)**
- ***Store* für die Übertragungen der Werte von der CPU (Rx-Register) in den Speicher**
- ***Move* für die Übertragungen der Werte innerhalb der CPU (zwischen den Rx-Registern)**



MOV-Befehle

MOV (move) legt einen Wert im Zielregister ab. Der Wert kann ein unmittelbarer Wert oder ein Wert aus einem Register sein. Er kann vor dem Speichern verschoben (shifted) werden.

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

Verwendung

- **Kopieren eines Wertes von einem Register in ein anderes**

```
mov r1, r4          // kopiert den im Register R4 gespeicherten Wert
                    // in das Register R1
```

- **Initialisierung einer Konstante in einem Register**

```
mov r1, #250        // kopiert den Wert 250 in das Register R1
```

- **Verschieben oder Rotieren eines Registers**

```
mov r1, r1, lsl #5  // verschiebt den im Register R1 gespeicherten Wert
                    // um 5 Bit nach links
```

- **Rücksprung aus der Subroutine durch Kopieren der Rücksprungadresse (LR oder R14) in den Programmzähler (R15 oder PC)**

```
mov pc, lr          // Rücksprung aus der Subroutine, wenn die Adresse
                    // zuvor im Register LR (link register)
                    // gespeichert worden ist
```

Weitere Move-Befehle

MVN	kopiert das 1er-Komplement des Wertes (binäre Inversion des Wertes)	MVN{<cond>}{S} <Rd>, <shifter_operand>
MRS	kopiert den Wert aus einem der Statusregister in ein allgemeines Register	MRS{<cond>} <Rd>, CPSR MRS{<cond>} <Rd>, SPSR
MSR	kopiert die Felder (c, x, s, f) in eines der Statusregister (CPSR *current* oder 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 (Bit 0..7)
x = extended (Bit 8..15)
s = status (Bit 16..23)
f = flags (Bit 24..31)

Beispiel (MSR / MRS)

Befehle, die den Prozessor in den User-Modus (Bit5 – Bit 0) des CPSR (0b10000) zwingen

```
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-Befehl

LDR (load) lädt einen Wert von einer Speicheradresse in ein internes CPU-Register

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

Verwendung

- **kopiert einen im Speicher abgelegten Wert in ein internes Register**

```
ldr   r1,[r4]     // kopiert den im Speicher unter der im Register R4
                  // enthaltenen Adresse Wert in das Register R1
```

- **kopiert eine Adresse in ein internes Register**

```
ldr   r1,=var     // kopiert den Wert der Adresse der Variablen
                  // var in das Register R1
```

- **Verwendung des PC (R15) als Verzweigung**

```
ldr   pc,=toto    // kopiert die Adresse von toto in den PC, was einen
                  // unbedingten Sprung des Programms an die Adresse
                  // toto bewirkt
```

Beispiel (LDR)

Befehle für die Verzweigung in eine Subroutine (Subroutine toto) und den Rücksprung unter Verwendung der Befehle ldr und mov

```
        mov  lr,pc              // sichert pc im lr (lr = pc + 8)
        ldr  pc,=toto           // Springt zur Adresse toto
        b    stop

toto:   lsr  r1,r1,#1
        mov  pc,lr              // realisiert den Rücksprung

stop:   b    stop
```

Weiterer Load-Befehl

LDM	Mehrfachtransfer aus dem Speicher in die Register. Verwendet für die zusammenhängende Wiederherstellung des Inhalts von mehreren Registern, der zuvor im Speicher abgelegt wurde.	LDM{<cond>}<addr_mode> <Rn>{!}, <registers>

```
@exemple:
dest:   .long 101,102,103,104,105,106

        ldr     r9,=dest
        ldmia   r9!,{r1-r6}
```

Adressierungsmodi:

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

! → Aktualisierung des Registers <Rn> nach der Operation

STR-Befehl

STR (store) überträgt den Wert eines internen CPU-Registers in eine Speicheradresse

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

Verwendung

- **kopiert einen Wert aus einem internen Register in den Speicher**

```
str  r1,[r4]   // kopiert den im Register R1 enthaltenen Wert
     // an die Speicheradresse, die im Register R4 enthalten ist
```

- **Speicherung des PC (R15) als relative Adresse im Speicher**

```
str  pc,[r3]   // kopiert die Adresse des PC an die
     // Speicherstelle, die im Register R3 enthalten ist
```

Weiterer Store-Befehl

STM	Mehrfachtransfer der Register in den Speicher. Verwendet für die zusammenhängende Sicherung des Inhalts von mehreren Registern im Speicher.	STM{<cond>}<addr_mode> <Rn>{!}, <registers>

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

Adressierungsmodi:

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

! → Aktualisierung des Registers <Rn> nach der Operation

Arithmetische und logische Befehle

Die arithmetischen und logischen Befehle sind in verschiedene Familien gruppiert

- **Die arithmetischen Befehle (Addition, Subtraktion und Multiplikation)**
- **Die logischen Befehle (UND, ODER, Exklusiv ODER, Test)**
- **Die Vergleichsbefehle**

Format dieser Operationen

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

- <opcode>: opération (add, sub, and, or, eor, …)
- <cond>: Ausführungsbedingung (optional)
- {S}: Aktualisierung der Flags nach der Ausführung (optional)
- <Rd>: 1. Operand ➔ Zielregister
- <Rn>: 2. Operand
- <shifter_operand>: 3. Operand

Arithmetische Operationen (ADD, ADC, SUB, SBC, RSB, RSC)

ADD	Addition von zwei Werten (Rd = Rn + shifter_operand)	ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
ADC	Addition von zwei Werten unter Berücksichtigung des Carry	ADC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
SUB	Subtraktion von zwei Werten (Rd = Rn - shifter_operand)	SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
SBC	Subtraktion von zwei Werten unter Berücksichtigung des Carry	SBC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
RSB	Inverse Subtraktion von zwei Werten (Rd = shifter_operand - Rn)	RSB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
RSC	Inverse Subtraktion von zwei Werten unter Berücksichtigung des Carry	RSC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
@Exemple:
     ldr   r0, =3583
     ldr   r1, =7620
     add   r2,r0,r1
```

➔ R2 = R0 + R1 = 3583 + 7620

Addition mit Carry

Eine Addition mit Carry ermöglicht die Realisierung einer Addition mit 64 Bit

R1	0x00000000	R0	0xffffffff
R3	0x00000000	R2	0x00000001
R5	0x00000001	R4	0x000000000

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

Subtraktion und inverse Subtraktion

Die Subtraktionsoperation in eine Richtung:

Rd = Rn – shitfer_operand

```
@Exemple:
     mov  r0, #50
     mov  r1, #24
     sub  r2, r0, r1
```

R2 = R0 – R1 = 50 – 24 = 26

Die Operation der inversen Subtraktion wechselt die Richtung der Operanden:

Rd = shifter_operand – Rn

```
@Exemple:
     mov  r0, #50
     rsb  r2, r0, #24
```

R2 = 24 – R0 = 24 – 50 = -26

Die inverse Subtraktion erlaubt, den dritten Operanden zu verschieben *(shifter).*

```
@Exemple:
     mov  r0, #25
     mov  r1, #24
     rsb  r2, r0, r1, lsl #1
```

R2 = (R1<<1) – R0
= (24<<1) – 25
= 48 – 25 = 23

Multiplikationsoperationen

MUL	Multiplikation 32 Bit x 32 Bit → 32 Bit	MUL <Rd>, <Rm>, <Rs> → Rd = Rm * Rs
MLA	Multipliziert mit einem Akkumulator 32 Bit x 32 Bit → 32 Bit + acc	MLA <Rd>, <Rm>, <Rs>, <Rn> → Rd = Rm * Rs + Rn
SMULL	Multipliziert Werte mit Vorzeichen 32 Bit mit Vorzeichen x 32 Bit mit Vorzeichen → 64 Bit mit Vorzeichen	SMULL <RdLo>, <RdHi>, <Rm>, <Rs> → RdLo = lower (Rm * Rs) → Bit 31-0 → RdHi = upper (Rm * Rs) → Bit 63-32
UMULL	Multipliziert Werte ohne Vorzeichen 32 Bit x 32 Bit → 64 Bit	UMULL <RdLo>, <RdHi>, <Rm>, <Rs> → RdLo = lower (Rm * Rs) → Bit 31-0 → RdHi = upper (Rm * Rs) → Bit 63-32

Ausser diesen Befehlen implementiert der Prozessor weitere Multiplikationsoperationen

AND	Führt eine logische bitweise UND-Operation zwischen zwei Operanden aus	AND{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
EOR	Führt eine logische bitweise Exklusiv-ODER-Operation zwischen zwei Operanden aus	EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
ORR	Führt eine logische bitweise ODER-Operation zwischen zwei Operanden aus	ORR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
BIC	Führt eine logische bitweise UND-Operation zwischen einem Wert und dem 1er-Komplement des zweiten Wertes aus	BIC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
@Exemple:
        mov    r0, #0x11
        mov    r1, #0x22
        mov    r2, #0x44
        mov    r3, #0x88
        and    r0, r0, #0x0f
        orr    r1, r1, #0x88
        eor    r2, r2, #0xf0
        bic    r3, r3, #0xf0
```

CMP-Befehl

CMP (compare) vergleicht zwei Werte. Der erste Wert stammt aus einem Register. Der zweite Wert kann ein unmittelbarer Wert, ein Wert aus einem Register oder das Ergebnis einer Verschiebung vor dem Vergleich sein. CMP aktualisiert die *condition flags* ausgehend vom Ergebnis der Subtraktion des zweiten Wertes vom ersten.

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

Verwendung

- **Vergleich eines Registers mit einem unmittelbaren Wert**

```
cmp r1, #35 // vergleicht den im Register R1 enthaltenen Wert
            // mit dem Wert 35
```

- **Vergleich zweier Werte, die in zwei Registern enthalten sind**

```
cmp r1, r3  // vergleicht die in den Registern
            // R1 und R3 enthaltenen Werte
```

CMN-Befehl

CMN (Compare Negative) vergleicht einen Wert mit dem 2er-Komplement eines zweiten Wertes. Der erste Wert stammt aus einem Register. Der zweite Wert kann ein unmittelbarer Wert, ein Wert aus einem Register oder das Ergebnis aus einer vorangegangenen *Shift*-Operation sein. CMN aktualisiert die *condition flags* ausgehend vom Ergebnis der Addition der beiden Werte.

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

Verwendung

- **Vergleichen mit dem 2er-Komplement eines Wertes**

```
cmn r1, #35 // vergleicht den im Register R1 enthaltenen Wert
            // mit dem Wert -35
```

- **Beispiel:**

```
mov r0, #25
mov r1, #-25
cmn r0, r1
```

Weitere Vergleichsbefehle

TST	Vergleicht zwei Werte und aktualisiert die Flags auf der Basis einer logischen UND-Operation	TST{<cond>} <Rn>, <shifter_operand>
TEQ	Vergleicht zwei Werte und aktualisiert die Flags auf der Basis einer logischen EOR-Operation (Exklusiv ODER)	TEQ{<cond>} <Rn>, <shifter_operand>

TST kann dazu verwendet werden zu bestimmen, ob mindestens ein Bit in einer Bitserie eine Eins ist.

```
@Exemple:
        mov     r0, #0x53
        mov     r1, #0x04
        tst     r0, r1
```

Verzweigungsoperationen

B	Verzweigung durch Offset, unmittelbarer Wert von 24 Bit mit Vorzeichen → *PC = PC + offset << 2 + 8* → *±32MB*	B <offset_24>	loop: ….. ….. ….. b loop
BX	Direkte Verzweigung durch Register (z. B. für den Rücksprung aus Subroutinen) → *PC = Rm & 0xffff'fffe* *Anmerkung: erlaubt den Übergang in den Thumb-Modus (16-Bit-Befehl, wird in diesem Kurs nicht behandelt)*	BX <Rm>	routine: ….. ….. ….. bx lr
BL	Verzweigung durch Offset und Sicherung der nächsten auszuführenden Adresse (Rücksprungadresse), für den Aufruf einer Subroutine → *LR = PC + 4* → *PC = PC + offset << 2 + 8* → *±32MB*	BL <offset_24>	….. ….. ….. bl routine
BLX	Verzweigung und Sicherung der nächsten auszuführenden Adresse, für den indirekten Aufruf einer Subroutine → *LR = PC + 4* → *PC = Rm & 0xffff'fffe*	BLX <Rm>	ldr r1,=routine ….. ….. blx r1

Jede Verzweigung kann bedingt ausgeführt werden.

Mnemonic	Beschreibung
B/BAL	Always (AL normally omitted)
BEQ	Equal
BNE	Not equal
BCS	Carry Set
BCC	Carry Clear
BMI	Negative (minus)
BPL	Positive or zero (plus)
BVS	Overflow
BVC	No overflow
BHI	Unsigned higher
BHS	Unsigned higher or same
BLS	Unsigned lower or same
BLO	Unsigned lower
BGT	Signed greater than
BGE	Signed greater than or equal
BLE	Signed less than or equal
BLT	Signed less than

Beispiel einer Schleife

```
#define CONSTANT 10

int i;
int sum = CONSTANT;

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

```
        mov     r3, #10
        mov     r2, #0
        b       test
loop:   add     r3, r3, r2
        add     r2,#1
test:   cmp     r2,#10
        blo     loop
```