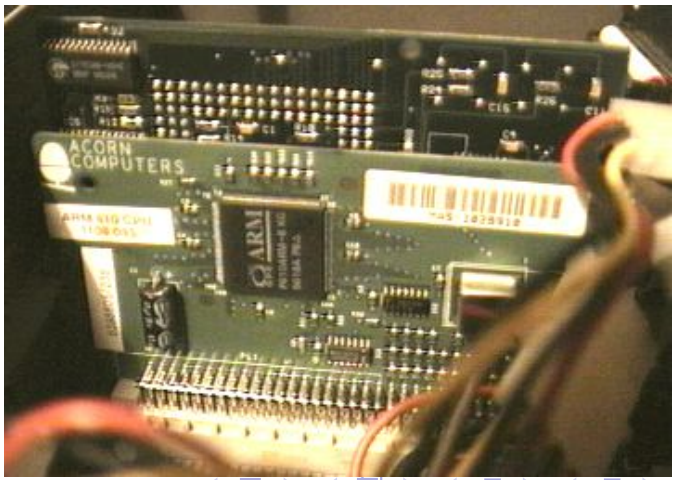
ARM-assembler cursus

Dick Tanis

2006

```
1580 :    MOVNE   R0,#7
1590 :    LDRNE   R2,[R12,#4]
1600 :    SWINE   "XOS_Module"
1610 :    LDMFD   R13!,{pc}
1620 :
1630 : .SWIhandler
1640 :    MOV     R10,R12
1650 :    LDR     R12,[R12,#0]
1660 :    CMP     R11,#10
1670 :    ADDLT   pc,pc,R11,LSL #2
1680 :    B       swi_unknown
1690 :    B       swi_resetdict
```



Inhoud

Inleiding cursus

Doel cursus:

- Werking van de ARM en architectuur
- Instructies + condities
- Data-verwerking en stacks
- ARM-assembler in basic
- Maken van utilities
- Maken van modules ?

Getalstelsels

Getallen zijn een som van machten: $n_0 k^0 + n_1 k^1 + n_2 k^2 + \ldots$,
Integers k en $n = 0, \ldots, k-1$
Belangrijkste getalstelsels:

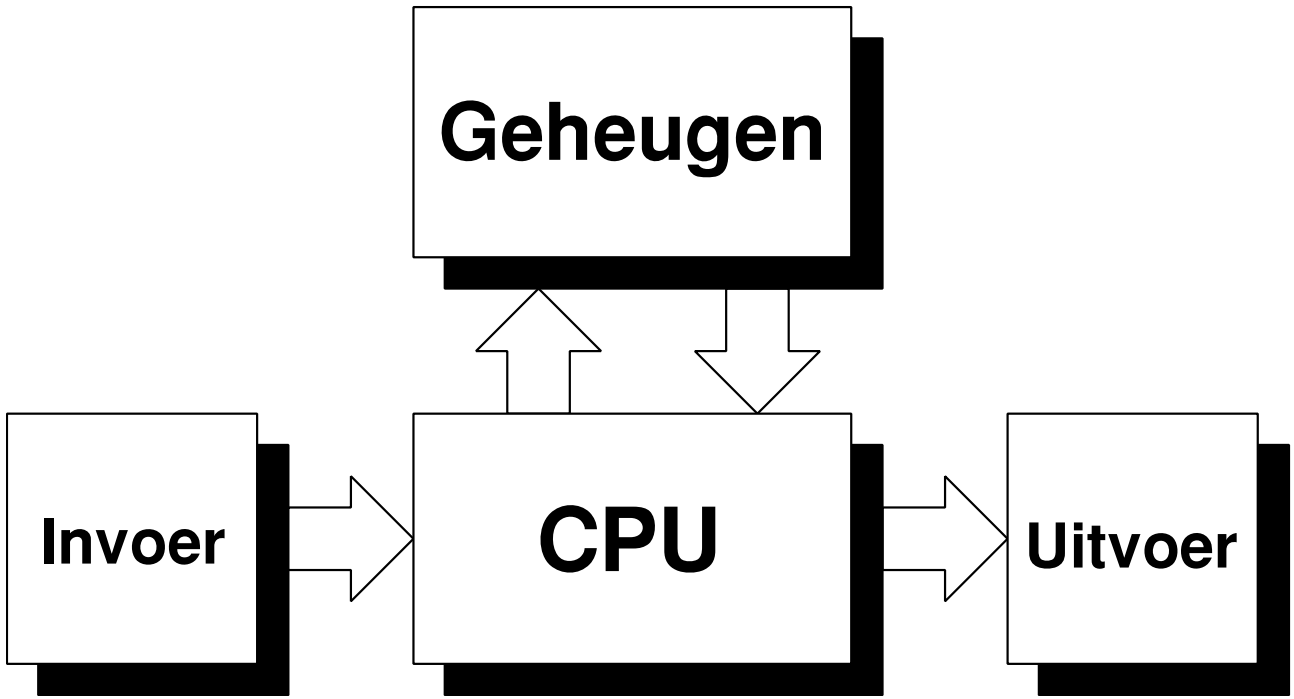
- Decimaal, $k = 10$: 3, 48, 345, 1454, 32678
- Binair, $k = 2$: %1, %10, %101, %1011, %10100
- Hexadecimaal, $k = 16$: &A, &1D, &E10, &1F4A, &3EFB

Reden hexadecimaal:

$$\&F = \%1111,\ \&FF = \%1111\ 1111$$

$$\&A = \%1010,\ \&AF = \%1010\ 1111$$

Typisch computer systeem



- Invoer/uitvoer
- Geheugen (in bytes onder RISC OS)
- Communicatie-bussen

Communicatie-bussen

Verbinding tussen CPU en andere andere delen in de computer

Data-bus: overbrengen data van en naar geheugen

- parallel
- 32 bits breed op de ARM, oftewel 1 word = 4 bytes
- bidirectioneel

Adres-bus: specificiëren van geheugen-locatie

- ARM 2 en 3, alleen 26 bits
- ARM 6 en hoger, 26 of 32 bits
- XScale alleen 32 bit

Word/byte locaties in geheugen

Benaderen van data in geheugen als 8 bits (byte) of 32 bits (word)

	Bit 31 0				
Locatie 0	byte 3	byte 2	byte 1	byte 0	word 0
Locatie 4	byte 7	byte 6	byte 5	byte 4	word 1
Locatie 8	byte 11	byte 10	byte 9	byte 8	word 2
Locatie 12	byte 15	byte 14	byte 13	byte 12	word 3

Ophalen van words in het geheugen kan alleen met word-aligned (deelbaar door 4) adressen

Instructies/processor cyclus

Instructies zijn binaire nummers en bestaan uit bitgroepen.

- Groep voor operaties: optellen, aftrekken etc
- Groep voor adressen in geheugen
- Groep voor registers
- Groep voor speciale operaties: shiften etc

Cyclus RISC processor in fases:

1. Ophalen instructie (Fetch)
2. Decoderen instructie (Decode)
3. Uitvoeren van instructie (Exucute)
4. Toegang tot cache (Access)
5. Wegschrijven naar register/geheugen (Writeback)

Pipelining

In ARM 2-7, 3 fases onafhankelijk en overlapbaar

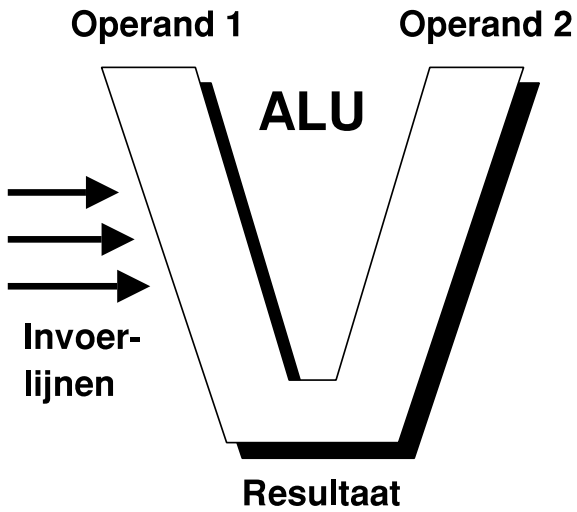
Cyclus 1	ophalen instructie 1	<leeg>	<leeg>
Cyclus 2	ophalen instructie 2	decoderen instructie 1	<leeg>
Cyclus 3	ophalen instructie 3	decoderen instructie 2	uitvoeren instructie 1
Cyclus 4	ophalen instructie 4	decoderen instructie 3	uitvoeren instructie 2

Voordeel: Effectief 1 instructie uitgevoerd per cyclus
Nadeel: Pipeline moet soms geleegd worden (branch-instructies)

Arithmetic Logic Unit (ALU)

Instructies:

- data-instructies: verplaatsen van registers, ARM $\leftrightarrow$ geheugen
- rekenkundige instructies, ALU: optellen, aftrekken, vermenigvuldigen, OR, AND etc



Barrel shifter

Eerst gaat operand 2 via de barrel shifter naar de ALU. De barrel shifter verschuift data in operands.

Shift van 3 naar links

Data voor verschuiving: %01101001100110011110010110101111
Data na verschuiving : %01001100110011110010110101111000

Barrel shifter heeft 3 elementen nodig:

1. Orginele operand die verschoven moet worden
2. Het aantal verschuivingen 0-32
3. Type verschuiving: links, rechts etc

Processor registers

Register:

Hardware element in de CPU

32 bits groot

Register $\leftrightarrow$ geheugen

Toegang tot register is sneller dan tot geheugen

R0 tot en met R15

Speciale registers:

- R15: program counter (26/32 bit)/status register (26 bits)
- R14: link register (voor subroutines)
- CPSR: huidige status register (32 bit), ARM 6 en hoger
- SPSR: status register vorige modus (32 bit), ARM 6 en hoger

26 bit programm counter en status register

Register 15:

- status register, bit 31-28
- interrupt-vlaggen, bit 27 en 26
- geheugenadres voor de volgende instructie, bit 25-2
- modus-vlaggen, bit 1 en 0

Bit:	31	30	29	28	27	26	25 2	1	0
R15:	N	Z	C	V	I	F	...Programm counter...	S1	S0

Status vlaggen

De status vlaggen betekenen:

N	Negative	Is 1 bij een negatief resultaat van een berekening
Z	Zero	Is 1 als uitkomst van een berekening nul is
C	Carry	Bit 32 voor 64 bit berekeningen
V	Overflow	Is 1 als resultaat van een berekening niet in 32 bits past of bij een foutmelding (RISC OS)
I	IRQ	Geen interrupts dan 1
F	FIQ	Geen snelle interrupts dan 1

Modus vlaggen

S1 en S0 leveren de volgende processor modi op:

S1	**S0**	**Modus**		**Functie**
0	0	USR	Gebruiker modus	Hierin draaien de applicaties
0	1	FIQ	Snelle interrupt modus	Afhandelen van externe devices
1	0	IRQ	Interrupt modus	Afhandelen van externe devices
1	1	SVC	Supervisor modus	OS-routines draaien in deze modus, SWI's

Registers in diverse modi

Register	Processor registers in modus:			
Naam	**User**	**FIRQ**	**IRQ**	**SVC**
R0	R0	R0	R0	R0
R1	R1	R1	R1	R1
R2	R2	R2	R2	R2
R3	R3	R3	R3	R3
R4	R4	R4	R4	R4
R5	R5	R5	R5	R5
R6	R6	R6	R6	R6
R7	R7	R7	R7	R7
R8	R8	R8_FIRQ	R8	R8
R9	R9	R9_FIRQ	R9	R9
R10	R10	R10_FIRQ	R10	R10
R11	R11	R11_FIRQ	R11	R11
R12	R12	R12_FIRQ	R12	R12
R13	R13	R13_FIRQ	R13_IRQ	R13_SVC
R14	R14	R14_FIRQ	R14_IRQ	R14_SVC
R15	R15	R15	R15	R15

Algemeen

Vanaf ARM 6 volledig 32 bits PC. Dit betekent:

- PSR heeft een apart register: CPSR
- PSR kan tegelijkertijd met PC worden bewaard bij springen naar andere modus. Elke belangrijke modus heeft nu een eigen SPSR en bevat de oude PSR van de vorige modus.
- Nieuwe instructies voor toegang tot CPSR en SPSR

Door 32 bits PSR zijn er meer processor modi oa: Abort modus en Undefined instruction modus

Processor modi

ARM ingesteld als 32 bits processor levert 10 processor modi:

- User modus: normaal voor applicaties
  of User26 modus: 26 bits versie
- FIQ modus: snelle afhandeling van hardware
  of FIQ26 modus: 26 bits versie
- IRQ modus: voor de standaard interupts
  of IRQ26 modus: 26 bits versie
- SVC modus: beschermde modus voor OS
  of SVC26 modus: 26 bits versie
- Abort (ABT) modus: springt in deze modus bij ophalen data of instructie op ongeldig adres.
- Undefined (UND) modus: springt in deze modus bij een ongedefinieerde instructie.

Registers in diverse modi

User26	**SVC26**	**IRQ26**	**FIQ26**
R0	R0	R0	R0
⋮	⋮	⋮	⋮
R7	R7	R7	R7
R8	R8	R8	R8_FIRQ
R9	R9	R9	R9_FIRQ
R10	R10	R10	R10_FIRQ
R11	R11	R11	R11_FIRQ
R12	R12	R12	R12_FIRQ
R13	R13_SVC	R13_IRQ	R13_FIRQ
R14	R14_SVC	R14_IRQ	R14_FIRQ
..... R15 (PC/PSR)			

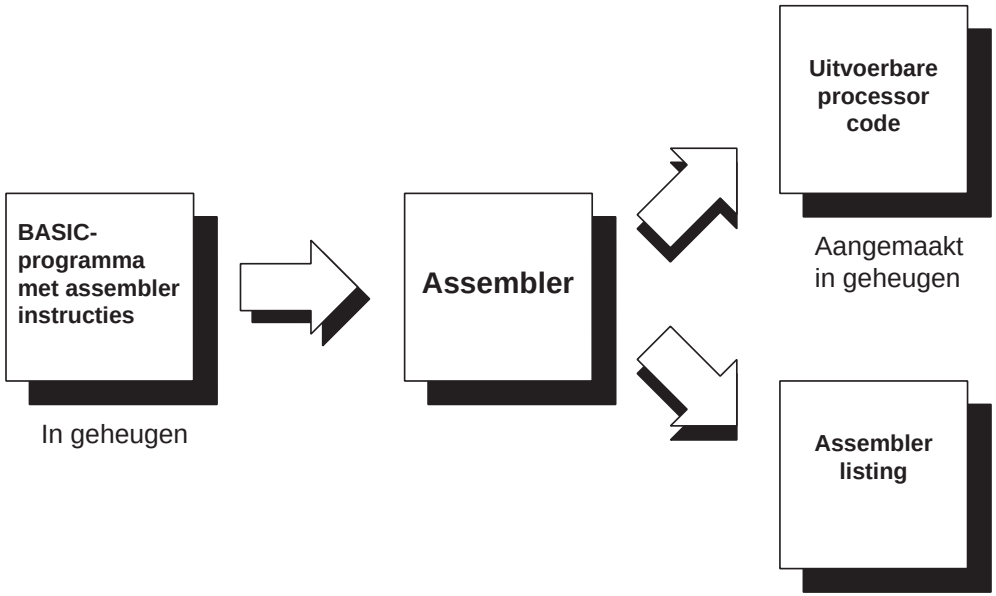
User	**SVC**	**IRQ**	**ABT**	**UND**	**FIQ**
R0	R0	R0	R0	R0	R0
⋮	⋮	⋮	⋮	⋮	⋮
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIRQ
R9	R9	R9	R9	R9	R9_FIRQ
R10	R10	R10	R10	R10	R10_FIRQ
R11	R11	R11	R11	R11	R11_FIRQ
R12	R12	R12	R12	R12	R12_FIRQ
R13	R13_SVC	R13_IRQ	R13_ABT	R13_UND	R13_FIRQ
R14	R14_SVC	R14_IRQ	R14_ABT	R14_UND	R14_FIRQ
..... R15 (PC)					
..... CPSR					
	SPSR_SVC	SPSR_IRQ	SPSR_ABT	SPSR_UND	SPSR_FIRQ

CPSR en SPSR

31	30	29	28	...	7	6	-	4	3	2	1	0	
N	Z	C	V		I	F		M4	M3	M2	M1	M0	
								0	0	0	0	0	User26 modus
								0	0	0	0	1	FIQ26 modus
								0	0	0	1	0	IRQ26 modus
								0	0	0	1	1	SVC26 modus
								1	0	0	0	0	User modus
								1	0	0	0	1	FIQ modus
								1	0	0	1	0	IRQ modus
								1	0	0	1	1	SVC modus
								1	0	1	1	1	ABT modus
								1	1	0	1	1	UND modus

De Assembler

Assembler geïntegreerd in BASIC.



Binair	**Hexadecimaal**	**Assembler**
%11100001101000000001000000000010	&E1A01002	MOV **R1,R2**
%11100000100000110000000000000101	&E0831005	ADD **R1,R3,R5**
%11100001010100010000000000000101	&E1510005	CMP **R1,R5**

ARM Assembler-instructies

Assembler-instructies bestaan uit 3 delen:

- type instructie: MOV, ADD SUB etc
- operands waarop de instructie werkt
- conditie

Assembler-instructies

```
MOV  0,3
MOV  R0,R3
MOV  positie,teller
MOV  PC,r14
```

Simpel BASIC Assembler-programma

```
REM Onze eerste BASIC Assembler-programma
DIM code% 31
P%=code%
[
.hallo                      ;label van string
 EQUS   "Hallo allemaal"    ;string in geheugen
 EQUB   0                   ;afsluitende 0
 ALIGN                      ;P% weer word aligned
.programma                  ;label Assembler-programma
 ADR    R0,hallo            ;in R0 komt adres van
                            ;onze string
 SWI    "OS_Write0"         ;print string
 MOV    PC,R14              ;stop Assembler-programma
                            ;en terug naar BASIC
]
CALL programma
END
```

BASIC in Assembler

BASIC-operaties kunnen in Assembler-instructies gebruikt worden

Voorbeelden

```
MOV R0,#67
MOV R0,#ASC("C")
MOV R0,fred%
ADD R0,R0,X%*2+5
ADD R0,R0,#%10100101
MOV R0,#&F000
MOV R1,#(start% DIV 256)
MOV R2,#INT(SIN(DEG(60))*100)
```

Opmerking: BASIC-operaties worden uitgevoerd tijdens assembleren. Dus niet bij uitvoeren processor code!

Data transfer van en naar BASIC

Van BASIC naar machine-code:

Register 0 tot en met 7 kan gevuld worden via de variabelen `A%` tot en met `H%`

Van machine-code naar BASIC:

Inhoud van register 0 kan met
`<var> = USR(<adres/label Assembler-programma>)`
teruggegeven worden in variabele `<var>`

Condities

Instructie condities:

- Voorwaarde waaronder een instructie uitgevoerd mag worden
- 4 bits van de instructie: 16 condities
- Condities hangen af van de statusvlaggen: NZCV
- Statusvlaggen hangen af van vorige instructies

Voorbeeld

Conditie code %1011 betekent:

dat:	N vlag	=	SET	en	V vlag	=	CLEAR
of:	N vlag	=	CLEAR	en	V vlag	=	SET
of:	Z vlag	=	SET				

Voer instructie uit als vorige vergelijking: operand 1 $\leq$ operand 2

De conditie codes

In Assembler, conditie van instructie bestaat uit 2 letters

Assembler conditie codes

`EQ`:	Gelijk	`NE`:	Ongelijk
`VS`:	Overflow set	`VC`:	Overflow clear
`AL`:	Altijd	`NV`:	Nooit
`HI`:	Hoger	`LS`:	Lager of gelijk
`PL`:	Positief	`MI`:	Negatief
`CS`:	Carry set	`CC`:	Carry clear
`GE`:	Groter dan of gelijk	`LT`:	Lager dan
`GT`:	Groter dan	`LE`:	Lager dan of gelijk

`SUBPL R0,R1,R2` wordt uitgevoerd als resultaat van een vorige instructie positief was

Conditie EQ/NE

EQ: Gelijk
Conditie: Z vlag = set

Voorbeeld

`CMP`	`R5,R10`	Vergelijk de inhoud van register 5 met 10
`ADDEQ`	`R5,R5,#2`	Als inhoud van R5 en R10 gelijk tel dan 2 op bij de inhoud in R5

NE: Ongelijk
Conditie: Z vlag = clear

Voorbeeld

`CMP`	`R2,R0`	Vergelijk inhoud R2 met R0
`SUBNE`	`R2,R2,R0`	Als 2 en 0 niet gelijk dan trek de inhoud van elkaar af en bewaar resultaat in register 2

Conditie VS/VC

VS: Overflow set
Conditie: V vlag = set

Voorbeeld

`SWI "XOS_Byte"`	Voor de SYS call OS_Byte uit
`BVS error`	Spring naar error routine wanneer OS_Byte een foutmelding geeft

VC: Overflow clear
Conditie: V vlag = clear

Conditie PL/MI

MI: Negatief
Conditie: N vlag = set

Voorbeeld

`SUBS`	`R0,R0,#5`	Trek 5 af van de inhoud in R0
`ADDMI`	`R0,R0,#5`	Als vorig resultaat 'n negatief getal is tel dan er weer 5 bij op

PL: Positief
Conditie: N vlag = clear

Conditie: CS/CC

CS: Carry set of HS: hoger of zelfde (unsigned)
Conditie: C vlag = set

Voorbeeld

`ADDS`	`R1,R1,#1024`	Voeg 1024 toe aan de inhoud van R1
`ADDCS`	`R2,R2,#1`	Als carry gezet tel dan 1 op bij inhoud R2

CC: Carry clear of LO: lager (unsigned)
Conditie: C vlag = clear

Voorbeeld

`CMP`	`R1,#ASC("A")`	Vergelijk inhoud R1 met ascii-waarde van karakter A
`MOVCC`	`R1,R1,#0`	Als lagere ascii-waarde dan wordt inhoud R1 0

Conditie: AL/NV

AL: Altijd
Conditie: Altijd

Voorbeeld

```
ANDAL R0,R1,R2   Voer altijd R0 = R1 AND R2 uit
ADD   R1,R1,#2   Tel altijd 2 op bij R1 (AL is standaard conditie)
```

NV: Nooit
Conditie: Nooit

Instructie wordt met deze conditie nooit uitgevoerd. Maak hier geen gebruik van in je code!

Conditie: HI/LS

HI: Hoger (unsigned)
Conditie: C vlag = set en Z vlag = clear

Voorbeeld

```
CMP   R11,R6   Vergelijk R11 en R6
MOVHI R11,#0   Als R11 > R6 dan R11=0
```

LS: Lager of zelfde (unsigned)
Conditie: C vlag = clear of Z vlag = set

Voorbeeld

```
CMP   R4,R2      Vergelijk R4 met R2
ADDLS R4,R4,#1   Als R4 ≤ R2 dan verhoog R4 met 1
```

Conditie: GE/LT

GE: Groter dan of gelijk (signed)

Conditie: N vlag = set en V vlag = set

of: N vlag = clear en V vlag = clear

of: Z vlag = set

Voorbeeld

```
CMP    R5,R2       Vergelijk R11 en R6
SUBGE  R5,R5,#2    Als R5 ≥ R2 dan trek 2 af van R5
```

LT: Lager dan (signed)

Conditie: N vlag = set en V vlag = clear

of: N vlag = clear en V vlag = set

Conditie: GT/LE

GT: Groter dan (signed)

Conditie: N vlag = set en V vlag = set
of: N vlag = clear en V vlag = clear
en: Z vlag = clear

Voorbeeld

```
CMP    R8,R9              Vergelijk R8 en R6
SWIGT  256+ASC(">")       Als R8 > R9 dan print het > karakter
```

LE: Lager dan of gelijk (signed)

Conditie: N vlag = set en V vlag = clear
of: N vlag = clear en V vlag = set
of: Z vlag = set

Statusvlaggen aanpassen

Met S geef je aan dat het resultaat van de instructie de statusvlaggen mag aanpassen:

- Voordeel: Resultaat van 1 instructie kan je gebruiken om conditioneel meerdere instructies uit te voeren
- Vergelijkingsinstructies zetten altijd de vlaggen: CMP, CMN, TEQ, TST, (SWI)

Voorbeelden

```
.lus                label lus
SUBS   R1,R1,#1     verlaag teller met 1 en zet statusvlaggen
SUB    R5,R2,#5     R5=R2-5, zet geen statusvlaggen
BNE    lus          als teller ≠ 0 dan nog een keer de lus

CMP    R0,#32       vergelijk R0 met ascii karakter spatie
SUBCC  R1,R1,#1     als lager dan 32 verlaag teller met 1
MOVCC  R0,#0        als lager dan 32 R0 = afsluitende 0
```

Overzicht instructies

De dataverwerking instructies

ADD	ADC	SUB	SBC
RSB	RSC	MUL	MLA
MOV	MVN	CMP	CPN
AND	ORR	EOR	
BIC	TST	TEQ	

Formaat Assembler-instructie:

< Type Assembler-instructie > <Doel register>,
<Operand 1>, <Operand 2>

Operands

Operand 1: Register 0-15
Operand 2:

1. Een register: ADD R1, R2, **R3**
2. Een directe constante: AND R0, R4, **#%101**
3. Een verschoven register operand:
   - SUB R0, R1, **R2, LSL #3**
   - ADD R0, R1, **R2, LSL R10**
   - EOR R0, R1, **R2, ASR #1**

Bereik directe constantes

Instructie bevat 12 bits voor de directe constante:

- 8 bits voor de nummerieke constante
- 4 bits voor positie in operand

Bit 31	Bit 0	Positie
.....	76543210	0
10.....	765432	1
3210.....	7654	2
543210.....	76	3
76543210.....		4
..76543210.....		5
....76543210.....		6
......76543210.....		7
........76543210.....		8
..........76543210.....		9
............76543210.....		10
..............76543210.....		11
................76543210.....		12
..................76543210.....		13
....................76543210....		14
......................76543210..		15

LSL: logische shift naar links

Syntax: LSL #n of LSL Rx

Voorbeeld LSL #1

Voor:	x	←	b31 b30 b29 . . . b2 b1 b0	←	0
Na:	b31		b30 b29 b28 . . . b1 b0 0		
	Carry		Data word		

ASL werkt hetzelfde als **LSL**

LSR: logische shift naar rechts

Syntax: LSR #n of LSR Rx

Voorbeeld LSR #1

Voor:	0	$\rightarrow$	b31 b30 b29 ... b2 b1 b0	$\rightarrow$	x
Na:			0 b31 b30 ... b3 b2 b1		b0
			Data word		Carry

ASR: Rekenkundige shift naar rechts

Syntax: ASR #n of ASR Rx

Voorbeeld ASR #1

Voor:	b31	$\rightarrow$	b31 b30 b29 ... b2 b1 b0	$\rightarrow$	x
Na:			b31 b31 b30 ... b3 b2 b1		b0
			Data word		Carry

ROR: Roteer naar rechts

Syntax: ROR #n of LSR Rx

Voorbeeld ROR #1

Voor:	→	b31 b30 b29 . . . b2 b1 b0	→	x
Na:		b0 b31 b30 . . . b3 b2 b1		b0
		Data word		Carry

RRX: Roteer rechts met carry

Syntax: RRX

Voorbeeld RRX

Voor:	→	b31 b30 b29 . . . b2 b1 b0	→	x
Na:		x b31 b30 . . . b3 b2 b1		b0
		Data word		Carry

Instructies

ADD: Optellen
Syntax: ADD (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één + operand twee
Vlaggen: N,Z,C,V

ADC: Optellen met carry
Syntax: ADC (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één + operand twee + carry
Vlaggen: N,Z,C,V

64 bit optellen

32 bit hoog	32 bit laag
01101010101000111001110011001100	10110101000110001100011100011110
01010111000110100100011001100011	10101010101010011110011010011001
11000001101111011110001100110000(1)	01011111110000101010110110110111

Instructies

SUB: Aftrekken
Syntax: SUB (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één - operand twee
Vlaggen: N,Z,C,V

SBC: Aftrekken met carry
Syntax: SBC (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één - operand twee - NOT(carry)
Vlaggen: N,Z,C,V

Als 'geleend' moet worden op bit 31 (0 - 1): carry = clear (0)
Als niet 'geleend' moet worden op bit 31: carry = set (1)

Instructies

RSB: Omgekeerd aftrekken
Syntax: RSB (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand twee - operand één
Vlaggen: N,Z,C,V

SBC: Omgekeerd Aftrekken met carry
Syntax: SBC (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand twee - operand één NOT(carry)
Vlaggen: N,Z,C,V

Instructies

MOV: Verplaats data
Syntax: MOV (S) <doel reg>, <operand2>
Operatie: resultaat = operand twee
Vlaggen: N,Z,C

MVN: Verplaats geïnverteerde data
Syntax: MVN (S) <doel reg>, <operand2>
Operatie: resultaat = NOT operand twee
Vlaggen: N,Z,C

Voorbeelden

```
MVN R0,#0          zet -1 in R0
MVN R1,#9          zet -10 in R1
MVN R6,R7,LSR #1   R6 = NOT(R7/2)
```

Instructies

CMP: Vergelijk
Syntax: CMP <operand1>, <operand2>
Operatie: resultaat operand één - operand twee zet vlaggen
Vlaggen: N,Z,C,V

MVN: Vergelijk negatief
Syntax: CMN <operand1>, <operand2>
Operatie: resultaat operand één - (-operand twee) zet vlaggen
Vlaggen: N,Z,C,V

Voorbeelden

`CMN R5,R7`	vergelijk R5 met -R7
`CMN R1,#9`	vergelijk R6 met -9
`CMN R3,R0,LSL #1`	vergelijk R3, met -R0*2

Instructies

AND: logische AND
Syntax: AND (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één AND operand twee
Vlaggen: N,Z,C

ORR: logische ORR
Syntax: ORR (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één OR operand twee
Vlaggen: N,Z,C

Voorbeelden

`AND`	`R5,R5,#%1111`	R5 = R5 AND %1111 (bits 31-4 worden 0)
`ORR`	`R7,R7,#%1100`	zet bit 2 en 3 in R7
`ORRS`	`R5,R5,#2`	R5 = R5 OR 2 (resultaat zet vlaggen)

Instructies

EOR: logische exclusieve OR
Syntax: EOR (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één EOR operand twee
Vlaggen: N,Z,C

BIC: Clear bits
Syntax: BIC (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één AND(NOT(operand twee))
Vlaggen: N,Z,C

Voorbeelden

`EOR R7,R7,#%11`	zet bit 0 en 1 op 0 in R7
`BIC R5,R5,#%11`	zet bits 0 en 1 op 0 in R5
`BIC R6,R6,R6`	zet bit in R6 volgens R6 (R6=0)

Instructies

TST: Test bits
Syntax: TST <operand1>, <operand2>
Operatie: resultaat operand één AND operand twee zet vlaggen
Vlaggen: N,Z,C

TEQ: Test gelijkheid
Syntax: TEQ <operand1>, <operand2>
Operatie: resultaat = operand één EOR (NOT(operand twee))
Vlaggen: N,Z,C

Voorbeelden

`TST`	`R1,#%1000`	controleer of bit 3 is gezet
`CMPNE`	`R1,R2`	als bit 3 gezet dan vergelijk R1 met R2
`TEQEQ`	`R1,R3`	als bit 3 niet gezet of R1 is gelijk aan R2 dan vergelijk R1 met R3

Instructies

MUL: Vermenigvuldigen
Syntax: MUL (S) <doel reg>, <operand1>, <operand2>
Operatie: resultaat = operand één * operand twee
Vlaggen: N,Z en C is onbekend

Restricties:

- Doel register, operand 1 en operand 2 zijn registers
- Operand 2 geen directe constante of verschoven register
- Doel register en operand 1 niet hetzelfde register

MLA: Vermenigvuldig met optellen
Syntax: MLA (S) <doel reg>, <operand1>, <operand2>, som
Operatie: resultaat = (operand één * operand twee) + som
Vlaggen: N,Z en C is onbekend

Register 15 in instructies

R15 als verschillende operands in 26 bits mode:

- Als operand 1: alleen bits 2-25, programm counter beschikbaar
- Als operand 2: alle 32 bits, status-, interruptvlaggen, mode bits en programm counter
- Als doel register:
  - standaard: alleen bits 2-25, programm counter wordt gewijzigd
  - met S optie: bit 26-31, interrupt- en statusvlaggen worden ook gewijzigd

In 32 bits modus bevat R15 alleen de PC en is volledig beschikbaar/beschrijfbaar. Dus `MOVS PC,R14` niet gebruiken!

Pipeline en R15

Door pipelining bevat R15 niet het adres van de uitgevoerde instructie: Oftewel `MOV R15,PC` laat de processor 2 instructies (8 bytes) vooruit springen

Voorbeeld

```
&1000   ORRS   R15,PC,#1<<31    zet N-vlag en springt
&1004   SWIMI  256+7            als N-vlag gezet dan biep
&1008   MOV    PC,R14           eindigt routine
```

Wijzigen statusvlaggen in 26 bit modus

Via TEQP kunnen we de PSR wijzigen zonder de PC te wijzigen. Het resultaat van de EOR-operatie wordt weggeschreven in de PSR van R15.

Voorbeelden

`TEQP R15,#%11`	schakelt processor naar SVC-modus
`TEQP R15,#1<<31`	N-vlag aan, alle andere vlaggen uit

Zetten V-vlag, andere vlaggen ongewijzigd

`MOV R0,R15`	R0 is PC+PSR
`ORR R0,R0,#1<<28`	V-vlag wordt gezet in bewaarde PSR
`TEQP R0,#0`	resultaat EOR wordt opgeslagen in de PSR

Wijzigen status vlaggen in 32 bits modus

Via de instructies MSR en MRS kunnen de CPSR en SPSR gewijzigd worden

Voorbeelden

`MSR CPSR_csxf,R0`	kopieer R0 naar CPSR
`MSR SPSR_csxf,R0`	kopieer R0 naar SPSR
`MSR CPSR_f,R0`	kopieer statusvlaggen van R0 naar CPSR
`MSR CPSR_f,#1<<28`	zet alleen V-vlag van de statusvlaggen
`MRS R0,CPSR`	kopieer CPSR naar R0
`MRS R0,SPSR`	kopieer SPSR naar R0

In USR modus kan alleen de statusvlaggen van de PSR gewijzigd worden

Tussen geheugen en registers

Om data van en naar geheugen te kopiëren:

- Voor 1 register:
  - LDR: data vanuit geheugen in een register
  - STR: data vanuit register naar het geheugen
- Voor meerdere registers:
  - LDM: laden van data uit het geheugen in meerdere registers
  - STM: bewaren van meerdere registers in het geheugen

Indirecte adressering: Inhoud van register bepaalt geheugen-locatie

Pre-indexed adressering

Syntax:

LDR/STR <doel reg>, [base{,<offset>}]!

Base: register met startadres voor geheugenblok

Offset: positie in geheugenblok

- Een register
- Een constante
- Een verschoven register

Voorbeelden

`LDR R0,[R1,R2]`	Laad R0 van adres R1+R2
`STR R0,[R1,#-4]`	Bewaar R0 op adres R1-4
`LDR R2,[base,index LSL#2]`	Laad R2 van base+index*4
`LDR R3,[base,#4]!`	Laad R3 van base+4, base=base+4

Post-indexed adressering

Syntax:

LDR <doel reg>, [base],<offset>

Base: register met adres van geheugenblok

Offset: toevoeging aan base adres

- Een register
- Een constante
- Een verschoven register

Voorbeelden

`LDR R0,[R1],R2`	Laad R0 van adres in R1, R1=R1+R2
`STR R5,[R1],#4`	Bewaar R5 op adres in R1, R1=R1+4
`LDR R2,[R5],R2 LSL#2`	Laad R2 van adres in R5, R5=R5+R2*4
`STR R3,[R4],#-20`	Bewaar R3 op adres in R4, R4=R4-20

PC relatief en byte adressering

BASIC Assembler heeft een speciale instructie voor relatieve (directe) adressering:

LDR <doel reg>,adres

Voorbeelden

`LDR R0,&200`	Laad R0 van adres &200
`STR R5,tabel`	Bewaar R5 op adres waar label 'tabel' staat

Voor overdracht van bytes voegen we de B optie toe

Voorbeelden

`LDRB R0,[R1,R2 LSL#6]`	Laad byte van R1+R2*64
`STRB R0,[R5],#1`	Bewaar byte op R5, R5=R5+1

Meerdere registers

Syntax:

STM/LDM<opties> <base> {!}, <register lijst>

Opties

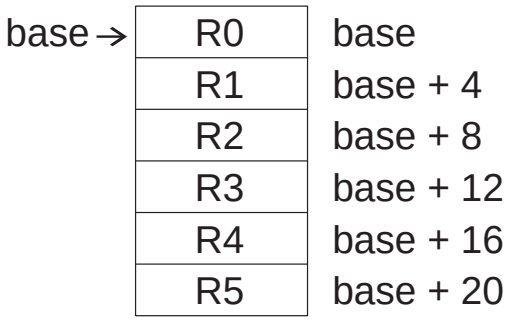
- I: verhogen (met 1 word) van adres na bewaren/laden van elk register
- D: verlagen (met 1 word) van adres na bewaren/laden van elk register
- A: aanpassen base adres na laden/wegschrijven register
- B: aanpassen base adres voor laden/wegschrijven register

Base: register met startadres

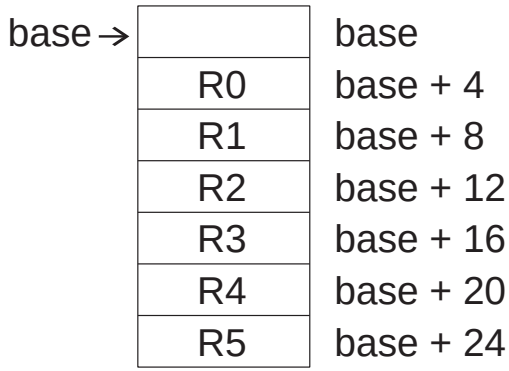
Registerlijst: R1,R4,R6, R1-R8 of R0,R1-R4,R7

STM voorbeelden

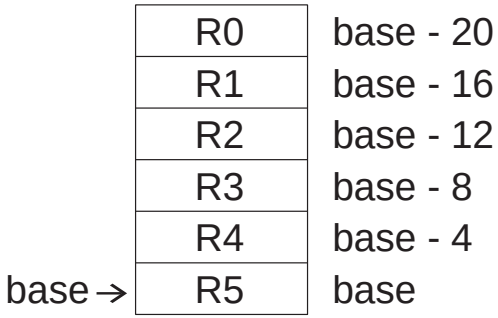
STMIA base,{R0-R5}



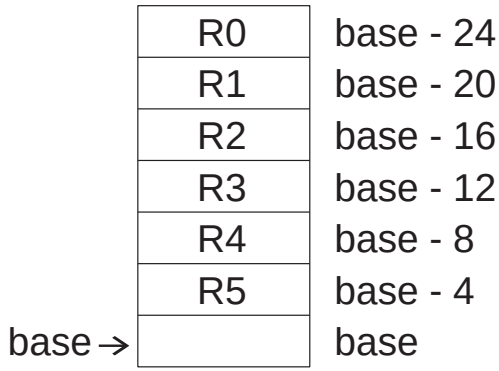
STMIB base,{R0-R5}



STMDA base,{R0-R5}

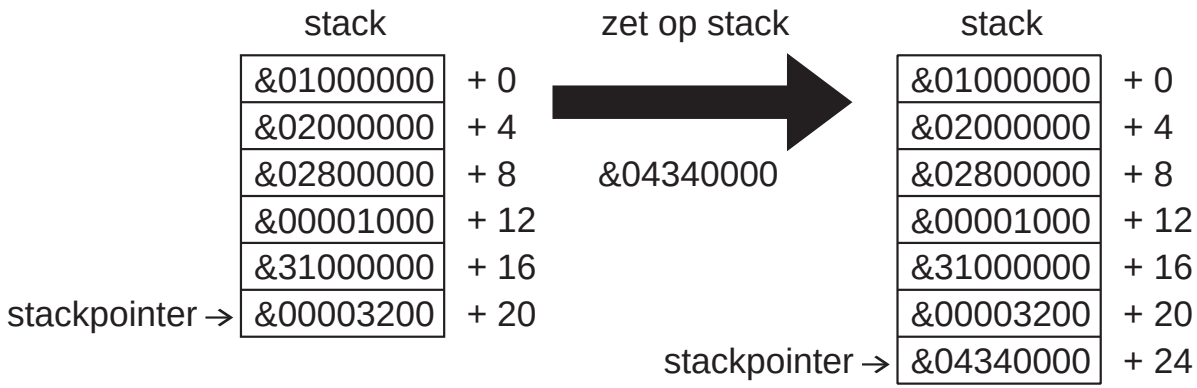


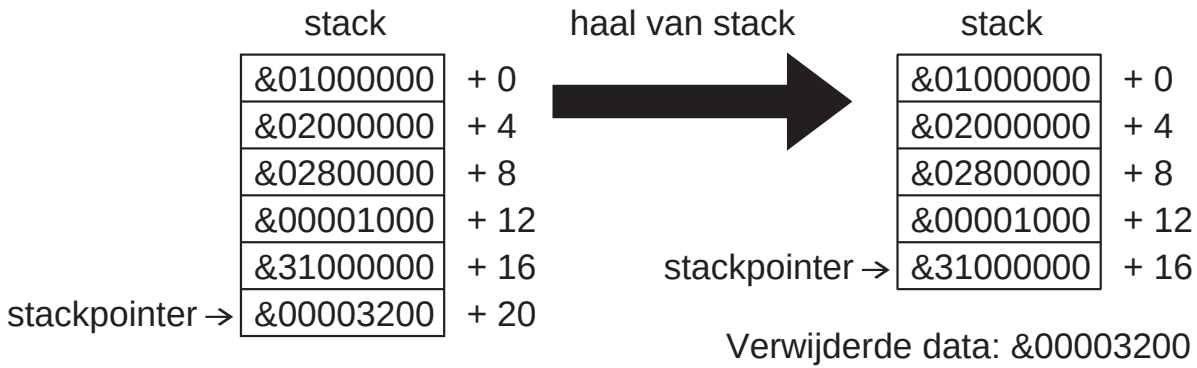
STMDB base,{R0-R5}



Computer stacks

Ophalen en wegzetten van data:



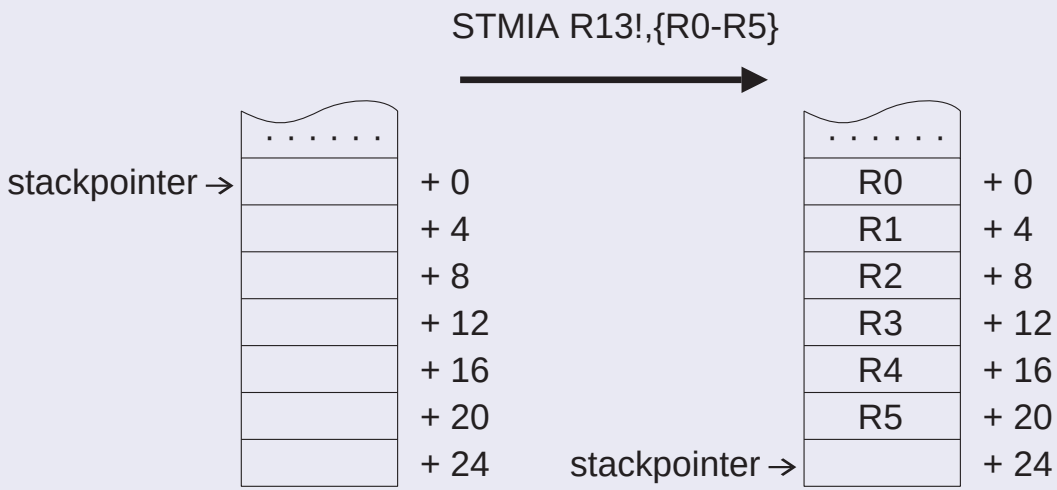


Type stacks

Er zijn 4 type computer stacks:

- FA: vol oplopende stack
- FD: vol aflopende stack
- EA: lege oplopende stack
- ED: lege aflopende stack

Lege oplopende (EA) stack



Stacks met LDM en STM

Registers op stack zetten:

STMDB (stack pointer)!,{register lijst}

Registers van stack halen:

LDMIA (stack pointer)!,{register lijst}

Voorbeelden

`STMFD R13!,{R1,R2,R4}`	Zet R1,R2 en R4 op FD stack
`LDMFD R13!,{R1,R2,R4}`	Laad R1,R2 en R4 van FD stack
`STMIA R13!,{R1-R7}`	Bewaar R1 tm R7 op EA stack
`LDMDA R13!,{R2-R4}`	Laad R2 tm R4 van FA stack
`STMDA R13!,{R0,R14}`	Bewaar R0 en R14 op ED stack

OPT instellingen

De OPT-parameters stellen de BASIC Assembler in, instellingen zijn:

- Bit 0: Print listing tijdens assembleren
- Bit 1: Rapporteer fouten, bijv. niet gevonden labels
- Bit 2: Offset assembly, als code op andere plek in het geheugen moet draaien dan waar het geassembleerd wordt
- Bit 3: Controle assemblage-geheugen, foutmelding als grens van het gealloceerde geheugen is bereikt
- Bit 4: Gebruik van 'nieuwe' instructies (MRS, MSR) mogelijk

OPT instellingen

	Nieuwe instr.	Geheugen-controle	Offset assembly	Rapport. fouten	Print listing
OPT 0	nee	nee	nee	nee	nee
OPT 1	nee	nee	nee	nee	ja
OPT 2	nee	nee	nee	ja	nee
OPT 3	nee	nee	nee	ja	ja
OPT 4	nee	nee	ja	nee	nee
OPT 5	nee	nee	ja	nee	ja
OPT 6	nee	nee	ja	ja	nee
OPT 7	nee	nee	ja	ja	ja
OPT 8	*nee*	*ja*	*nee*	*nee*	*nee*
OPT 9	*nee*	*ja*	*nee*	*nee*	*ja*
OPT 10	*nee*	*ja*	*nee*	*ja*	*nee*
OPT 11	*nee*	*ja*	*nee*	*ja*	*ja*
OPT 12	*nee*	*ja*	*ja*	*nee*	*nee*
OPT 13	*nee*	*ja*	*ja*	*nee*	*ja*
OPT 14	*nee*	*ja*	*ja*	*ja*	*nee*
OPT 15	*nee*	*ja*	*ja*	*ja*	*ja*
OPT 16	ja	nee	nee	nee	nee
⋮	⋮	⋮	⋮	⋮	⋮

Fout-controle

Dit geeft normaal een fout bij het assembleren:

```
TEQ R0,#1
BNE einde
SWI 256+7 ;beep
.einde
MOV PC,R14
```

Door 2 maal te assembleren kan het wel:

```
FOR pass% = 0 TO 2 STEP 2
P% = code%
[ OPT pass%
 TEQ R0,#1
 BNE einde
 SWI 256+7 ;beep
 .einde
 MOV PC,R14
]
NEXT pass%
```

Geheugen-controle en Offset Assembly

```
size% = 255
DIM code size%
FOR pass% = 8 TO 10 STEP : REM Geen listing en geheugen-controle
P% = code% : L% = P% + size%
[ OPT pass%
 ;assembler code
]
NEXT pass%
```

Offset assembly: code uitvoeren op ander plaats in geheugen dan waar het geassembleerd wordt.

```
DIM code%
FOR pass% = 5 TO 7 STEP 2 : REM Print listing en gebruik offset assembly
P% = 0 : O% = code%
[ OPT pass%
; assembler code
]
NEXT pass%
```

Data in assembler

Assembler instructie	Alternatief	Functie
EQUB	DCB, =var%	reserveert een byte
EQUW	DCW	reserveert een 'halve word' (2 bytes)
EQUD	DCD	reserveert een word (4 bytes)
EQUS	=string$	reserveert een string

ALIGN maakt P% weer word-aligned

Spring met link-adres

Syntax:

BL <adres>

Na sprong bevat R14 adres van instructie na BL (plus de PSR bij 26 bit)

Voorbeeld

```
.hoofd_programma
        :
BL subroutine
        :
einde_programma
.subroutine
        :
MOV PC,R14
```

Bewaren link register en vlaggen

Bewaren link register:

```
STMDB R13!,{R14} ;aan begin van programma/routine
LDMIA R13!,{PC}  ;aan het eind van programma/routine
```

Herstellen link register met vlaggen:

26 bit	32 bit
`STMDB R13!,{R14}`	`STMDB R13!,{R14}`
`.........`	`MRS R14,CPSR`
`.........`	`STMDB R13!,{R14}`
`.........`	`.........`
`.........`	`LDMIA R13!,{R14}`
`.........`	`MSR CPSR_f,R14`
`LDMIA R13!,{PC}^`	`LDMIA R13!,{PC}`

Bewaren link register en vlaggen

26/32 bit neutrale code in hoofd- en sub-routines (ROS 3.1 of hoger)
Deze voorbeeld-routine behoudt ook inhoud van R2.

```
STMDB     R13!,{R2,R14}     ;bewaar originele registers op stack
MRS       R14,CPSR          ;CPSR in R14, NOP-instructie onder ARM2/3
TEQ       R0,R0             ;alleen in USR-modus om Z-vlag te zetten
TEQ       PC,PC             ;EQ als 32 bit NE als 26 bit
STREQ     R14,[R13,#-4]! of STMEQDB R13!,{R14} ;bewaar CPSR op stack
.....................       ;de code van de routine
TEQ       R0,R0             ;alleen in USR-modus
TEQ       PC,PC
LDMNEIA   R13!,{R2,PC}^     ;einde in 26 bit modus, behoud registers
                            ;en zet originele PSR terug
LDR       R14,[R13],#4 of LDMIA R13!,{R14} ;haal CPSR op van stack
MSR       CPSR_f,R14        ;zet vlaggen terug in CPSR
LDMIA     R13!,{R2,PC}      ;einde routine en herstel inhoud registers
```

Utility eigenschappen

Een utility (&FFC) heeft de volgende eigenschappen:

- Code draait in RMA en moet plaats-onafhankelijk zijn
- Bij aanroepen code:
  - R0 = pointer naar aanroep van commando
  - R1 = pointer naar commando vlaggen
  - R12 = pointer naar werkruimte
  - R13 = pointer naar einde werkruimte
  - R14 = terugkeer-adres
- Werkruimte 1024 bytes
- Code draait in USR-modus, interrupts aan
- Gebruik SWI's met X optie, zelf fouten afhandelen
- Verlaat code met MOV PC,R14

Utilities in BASIC Assembler

```
REM Dit programma maakt een utility aan
copyright$ = "Dit is een utility"
file$      = "Utility"
size%      = 4095
PROCassemble
SYS"OS_File",10,file$,&FFC,,code%,endcode%
END
DEFPROCassemble
 DIM code% size%
 FOR pass% = 8 TO 10 STEP 2
 P% = code% : L% = P% + size%
 [OPT pass%
  ;Utility code
  ..............
  MOV  PC,R14
  EQUS copyright$
 ]
 NEXT pass%
 endcode%=P%
ENDPROC
```

Afhandelen van fouten

Afhandeling van foutmelding van SWI's:

- gebruik altijd de X-versie SWI
- bij een foutmelding van SWI:
  - is V-vlag gezet
  - R0 is pointer naar een foutblok:
    - R0+0 = foutnummer
    - R0+4 = pointer naar foutmelding, max 255 kars
- Gebruik na SWI: BVS fout_routine, MOVVS PC,R14 of LDMVS R13!,{PC} of iets dergelijks, in ieder geval moet je je code stoppen met V-vlag gezet en R0 pointer naar foutblok

Fatale foutmeldingen in eigen code:

- Maak eigen foutblok
- Zet V-vlag en eindig code met R0 pointer naar foutblok

Afhandelen van fouten

Voorbeeld fout-afhandeling van een SWI:

`STMDB R13!,{R0,R14}`	Zet registers op stack
`.....`	Verdere code in routine
`SWI "XOS_File"`	Voor SWI OS_File uit
`STRVS R0,[R13,#0]`	Bij fout zet R0 op stack inplaats van originele R0
`LDMVSIA R13!,{R0,PC}`	Bij fout eindig code en laat OS fout genereren
`.....`	Verdere code in routine
`LDMIA R13!,{R0,PC}`	Eindig routine normaal en zet registers terug

Vermenigvuldigen

```
MOV R0,R0,ASL #n      ;vermenigvuldig met 2^n
RSB R0,R0,R0,ASL #n   ;R0=R0*(2^n-1)
ADD R0,R0,R0,ASL #n   ;R0=R0*(2^n+1)
```

Voorbeelden

```
;vermenigvuldig met 10
MOV R0,R0,ASL #1      ;R0 maal 2
ADD R0,R0,R0, ASL #2 ;R0 maal 5
;vermenigvuldig met 31
RSB R0,R0,R0,ASL #5
```

Delen

```
; getal/deler = quotient met rest
; r_teken = teken rest
; q_teken = teken quotient
ANDS    r_teken,getal,#1<<31        ;zet teken negatief als getal negatief
RSBMI   getal,getal,#0              ;als negatief, maak getal positief
EOR     q_teken,r_teken,deler       ;bepaal teken van quotient
CMP     deler,#0                    ;controleer teken van deler
RSBMI   deler,deler,#0              ;als deler negatief, maak positief
MOV     rest,#0                     ;initialiseer rest
MOV     quotient,#0                 ;initialiseer quotient
MOV     posteller,#1<<31            ;initialiseer positieteller
.delen_loop                         ;verwerk alle alle 32 bits
MOVS    getal,getal,ASL#1           ;shift 1 plaats links
ADC     rest,rest,rest              ;rest maal 2 + bit 31 (carry)
CMP     rest,deler                  ;als rest >= deler
SUBGE   rest,rest,deler             ;dan rest=rest-deler
ORRGE   quotient,quotient,posteller ;dan voeg positiebit toe aan quotient
MOVS    posteller,posteller,LSR#1   ;zet posteller 1 bit naar links
BNE     delen_loop                  ;niet alle bits verwerkt dan volgende bit
CMP     q_teken,#0                  ;check teken quotient
RSBMI   quotient,quotient,#0        ;als negatief dan maak quotient negatief
CMP     r_teken,#0                  ;check teken rest
RSBMI   rest,rest,#0                ;als negatief dan maak negatief
```

Meerdere if constructies

Case When Otherwise structuren in Assembler:

```
 CMP   R0, #(eindtabel-tabel)/4 ;R0=routine nummer
 ADDCC PC, PC, R0, LSL#2          ;R0 niet groter dan aantal
                                  ;opties in tabel dan spring
                                  ;naar juiste procedure
 B      ongeldig                  ;als var groter,
                                  ;dan ongeldig

.tabel
 B      nul
 B      een
 B      twee
.eindtabel
```

Vlaggen aan en uitzetten

Vlaggen wijzigen met 26/32 neutrale code:

V-vlag

Aanzetten	Uitzetten
`CMP R0,#1<<31`	`CMP R0,#1<<31`
`CMNVC R0,#1<<31`	`CMNVS R0,#1<<31`

N-vlag

Aanzetten	Uitzetten
`MVN R0,#0`	`MOV R0,#1`
`TEQ R0,#0`	`TEQ R0,#0`

Z-vlag

Aanzetten	Uitzetten
`TEQ R0,R0`	`TEQ PC,#0`

C-vlag

Aanzetten	Uitzetten
`CMP PC,#0`	`ADDS R0,R0,#0`

Diversen

Laden van signed bytes:

```
LDRB R0, <adres>    ;laad byte
MOV  R0,R0,LSL #24 ;schuif op tot 31..24
MOV  R0,R0,ASR #24 ;zet terug, signed
```

Wisselen inhoud van 2 registers:

```
EOR R0,R0,R1
EOR R1,R0,R1
EOR R0,R0,R1
```

Intervallen

Voorbeeld:Zit getal in interval 1-10 of 100-200

```
CMP     R0,#1       ;vergelijk getal met ondergrens
RSBGES  R1,R0,#10   ;als groter of gelijk dan
                    ;bovengrens-getal, alles binnen
                    ;de grenzen 1-10 geeft een
                    ;positief resultaat, zet vlaggen
CMPLT   R0,#100     ;als getal>10 dan vergelijk met
                    ;ondergrens 100
RSBGES  R1,R0,#200  ;als groter of gelijk dan
                    ;bovengrens-getal, zet vlaggen
MVNGE   R3,#0       ;R3=-1 als getal in 1-10 of
                    ;100-200 ligt
```

Debuggen

Handige debuggingtools:

- Reporter (`http://www.avisoft.force9.co.uk/Reporter.htm`):
  - SWI Report_Registers, Report_Regs
  - SWI Report_Text0, Report_TestS
- ARMAlyser (`http://www.armclub.org.uk/free/`):

Wat is een module

Een module (&FFA) is een stuk software met de volgende eigenschappen:

- Uitbreiding op het OS: SCSIFS, ABCLibrary, Chimemod, Smartmenu
- Code draait in RMA of ROM en moet plaats-onafhankelijk zijn
- Private word (werkruimte) elke submodule heeft z'n eigen private word
- Module header, hiermee communiceert RISC OS met module:
  - Initialisatie, starten en beëindigen van module
  - Modulenaam, helpstring en versie
  - Startpunt (entrypoint) voor commando's inclusief help
  - Startpunt voor SWI's inclusief tabel voor SWI-namen
  - Startpunt voor afhandelen service calls

Werkruimte

Elke module 1 word private werkruimte.
Doel: Opslaan pointer naar werkelijke werkruimte (geheugenblok)

- Gebruik bij initialisatie:
  1. Alloceer geheugenblok tijdens initialisatie (OS_Module 6)
  2. Bewaar pointer in private word met STR R2,[R12]
- Bij elke aanroep van module door OS:
  1. R12 wijst naar private word
  2. Ophalen pointer naar eigen werkruimte met LDR R12,[R12]
- Bij beëindigen (finalise) van module:
  1. Geen code nodig om werkruimte vrij te geven RISC OS doet dit al standaard

Module header formaat

Offset	Type	Inhoud word
&00	offset naar code	Starten van module (taal of applicatie)
&04	offset naar code	Initialisatie-code
&08	offset naar code	Finalise-code
&0C	offset naar code	Afhandelen van service calls
&10	offset naar string	Module titel
&14	offset naar string	Helpstring
&18	offset naar tabel	Tabel commando-namen en help
&1C	nummer	Basisnummer van SWI (optioneel)
&20	offset naar code	SWI-code (optioneel)
&24	offset naar tabel	SWI decodeer-tabel (optioneel)
&28	offset naar code	SWI decodeer-code (optioneel)
&2C	offset naar string	Bestandsnaam Messages-bestand (optioneel)
&30	offset naar word	Module-vlaggen (optioneel)

Starten module

Bij aanroepen

R0 = pointer naar commando string, inclusief modulenaam
R12 = pointer naar private word

Bij verlaten

Stopt niet tenzij er een fout plaatsvindt

Processor modus

USR-modus

Gebruik

Dit startpunt in module wordt gebruikt om een omgeving (BASIC) of wimptaak te starten

Initialisatie

Bij aanroepen

R10 = pointer naar vlaggen, via *RMReInit of OS_Module
R11 = I/O basis of nummer submodule
R12 = pointer naar private word, inhoud = 0
R13 = SVC-stackpointer (FD)

Bij verlaten

R7 - R11 en R13, modus en interrupts onveranderd

Processor modus

SVC-modus

Gebruik

Alloceer werkruimte/DA's, registreer filters/vectors
Bij foutmelding, zet V, R0 -> foutblok en eindig routine
Stop met MOV PC,R14 of LDMFD R13!,{PC}

Finalise

Bij aanroepen

R11 = nummer of submodule
R12 = pointer naar private word
R13 = SVC-stackpointer (FD)

Bij verlaten

R7 - R11 en R13, modus, interrupts onveranderd

Processor modus

SVC-modus

Gebruik

Geef buffers/DA's vrij
Zet V met R0 -> foutblok als module niet gestopt mag worden
Stop met MOV PC,R14 of LDMFD R13!,{PC}

Afhandelen service calls

Bij aanroepen

R1 = nummer service call
R12 = pointer naar private word
R13 = SVC-stackpointer (FD)

Bij verlaten

R1 = 0 als service call door module wordt afgehandeld
R0, R2 - R8 parameter registers
R12 mag worden gewijzigd

Processor modus

SVC-modus of IRQ-modus

Gebruik

Handel ongeïnteresseerde services calls snel af
Stop met MOV PC,R14 of LDMFD R13!,{PC}

Afhandelen service call

Vanaf RISC OS 4 gebruik additionele service-tabel:

- svc-routine onveranderd behalve:

Adres	Inhoud
svc - 4	Offset naar service-tabel
svc	MOV R0,R0 (magische instructie)
svc + 4...	Vervolg svc-routine

- service-tabel:

Adres	Inhoud
tabel + 0	Bit 0 aan dan R1->index in tabel
	Bit 0 uit dan R1 = nummer service
tabel + 4	Offset svc-routine
tabel + 8	1e geïnteresseerde servicenummer
tabel + 12	2e geïnteresseerde servicenummer
tabel + X	Afsluitende 0

Titel en helpstring

Titelstring

Alfanummerieke string afgesloten met 0

Mag geen spaties of control codes bevatten

Helpstring

Alfanummerieke string met afgesloten 0

Geen control codes behalve tab

Vorm: module_naam Tab[Tab] v.vv (DD MMM YYY)

Tabel commando's en help

Tabel bevat sleutelwoorden van commando's en/of pointers naar helptekst en commando code. Tabel is als volgt opgebouwd:

Sleutelwoord commando
ALIGN voor word aligned
Offset naar code gerekend vanaf begin module, of 0 geen code
Informatie word
Offset naar syntax melding of 0 standaard bericht
Offset naar help tekst of 0 geen helptekst

Code offset commando's

Bij aanroepen

R0 = pointer naar commando vlaggen
R1 = aantal parameters gevonden door OSCLI
R12 = pointer naar private word
R13 = SVC-stackpointer (FD)
R14 = terugkeer-adres

Bij verlaten

R0 = bij fout pointer naar foutblok, V gezet

R7 - R11 en R13 onveranderd

Processor modus

SVC-modus

Informatie word commando's

Informatie word bevat informatie over aantal vlaggen en commando-type:

Byte	**Inhoud**
0	Minimum aantal parameters (0-255)
1	OS_GSTrans selectie voor de 1e 8 parameters
2	Maximum aantal parameters (0-255)
3	Vlaggen

Byte 1 Bit gezet dan parameter via GSTrans, bit 0 parameter 1, bit 1 parameter 2 etc

Bit 31 = 1 commando is een FS-commando zoals *bye, *eject etc. OSCLI vindt dit commando pas als er geen normale variant bestaat voor dit commando

Informatie word commando's

Bit 30 = 1

Commando is configuratie-optie (*Status of *Configure) en R0 is als volgt:

R0 = 0 *Configure is zonder parameters aangeroepen, module drukt syntax af (OS_Write0)

R0 = 1 *Status is aangeroepen, module drukt ingestelde waarde af

Anders *Configure is aangeroepen met parameters R0 -> commando vlaggen

Commando vlaggen incorrect of bij fout, zet V en R0 als volgt:

R0 = 0 Foute configuratie-optie

R0 = 1 Geen nummerieke parameter meegegeven

R0 = 2 Parameter is te lang

R0 = 3 Teveel parameters

R0 > 3 R0 = pointer naar foutblok

Informatie word commando's

Bit 29 = 1

Offset naar helptekst wijst naar code inplaats van string. De code wordt als volgt aangeroepen:

R0 = pointer naar een buffer
R1 = bufferlengte
R1 - R6 en R12 mogen veranderd zijn

Als aan het eind R0 <> 0 dan R0 = pointer naar string met afsluitende 0. OS drukt vervolgens deze string af.

Basisnummer SWI

Met het basisnummer identificeert RISC OS SWI met module. Het is geen pointer maar een vaste waarde. Neem als bijvoorbeeld hourglass:

&406C0	+0	Hourglass_On
	+1	Hourglass_Off
	+2	Hourglass_Smash
	+3	Hourglass_Start
	+4	Hourglass_Percentage
	+5	Hourglass_LEDs
	+6	Hourglass_Colours

Het basisnummer is &406C0. Module kan in totaal 64 SWI's hebben.

SWI-code

Bij aanroepen

R0 - R8 registers met eventuele opties

R11 = interne SWI-nummer (0-63)

R12 = pointer naar private word

R13 = SVC-stackpointer (FD)

R14 = terugkeer-adres + eventuele vlaggen (26 bit)

Bij verlaten

R0 - R8 teruggegeven waarden, bij fout R0 -> foutblok, V gezet

R10 - R12 mogen veranderd zijn

Gebruik MOV PC,R14 (32 bit), MOVS PC,R14 (26 bit)

Processor modus

SVC-modus

SWI-code voorbeeld

```
.SWIstart
 LDR    R12,[R12]                          ;R12 -> werkruimte
 CMP    R11,#(switabel_eind - switabel)/4  ;vergelijk SWI-nummer met aantal in tabel
 ADDLO  PC,PC,R11,LSL#2                    ;in bereik dan spring naar juiste routine
 B      swi_onbekend                       ;niet in bereik dan kennen we hem ook niet
.switabel
 B      SWInr_0                            ;spring naar SWI routine 1
 B      SWInr_1
 ..............
 B      SWInr_n
.switabel_eind
.swi_onbekend
 STR    R14,[R13,#-4]!                     ;bewaar link op SVC-stack
 ADR    R0,errteken_onbekende_swi
 MOV    R1,#0                              ;gebruike interne messages
 MOV    R2,#0                              ;gebruik buffer messagetrans
 ADR    R4,modulenaam                      ;vervang %0 voor modulenaam
 SWI    "XMessagesTrans_ErrorLookup"       ;R0 -> foutblok
 CMPVC  R1,#1<<31                          ;set V als V ongezet
 LDR    PC,[R13],#4                        ;haal link van stack en stop routine
.errteken_onbekende_swi
 EQUD   &1E6                               ;foutnummer badswi van OS
 EQUD   "BadSWI"                           ;bericht-teken
 EQUB   0
 ALIGN
```

SWI decodeer-tabel

OS gebruikt SWI decodeer-tabel voor conversies tussen nummer en SWI-naam.
Formaat decodeer-tabel:

SWI-groep prefix
Naam van SWI 0
...
Naam van SWI n
Eindig met 0 byte

Decodeer-tabel hourglass met 2 SWI's

```
EQUS "Hourglass"
EQUB 0
EQUS "On"
EQUB 0
EQUS "Off"
EQUB 0
EQUB 0
```

Modules in BASIC Assembler

```
REM Dit programma maakt een module aan
name$ = "Module" : version$ = "0.01 (4 november 2006)" : helpmod$ = name$+CHR$9+version$
size% = 4095 : PROCassemble : SYS"OS_File",10,name$,&FFA,,code%,endcode% : END
DEFPROCassemble
 DIM code% size% : FOR pass% = 12 TO 14 STEP 2 :REM offset assembly
 P% = 0 : O% = code% : L% = O% + size%
 [OPT pass%
  EQUD 0 ;geen startcode
  EQUD initialise
  EQUD 0 ;geen finalise
  EQUD service_call_handler
  EQUD title
  EQUD helptitle
  EQUD 0 ; geen commando's
 .title
  EQUS name$
  EQUB 0
 .helptitle
  EQUS helpmod$
  EQUB 0
  ALIGN
 .initialise
  ..............
 .service_call_handler
  ..............
 ]
 NEXT pass% : endcode%=O% : ENDPROC
```