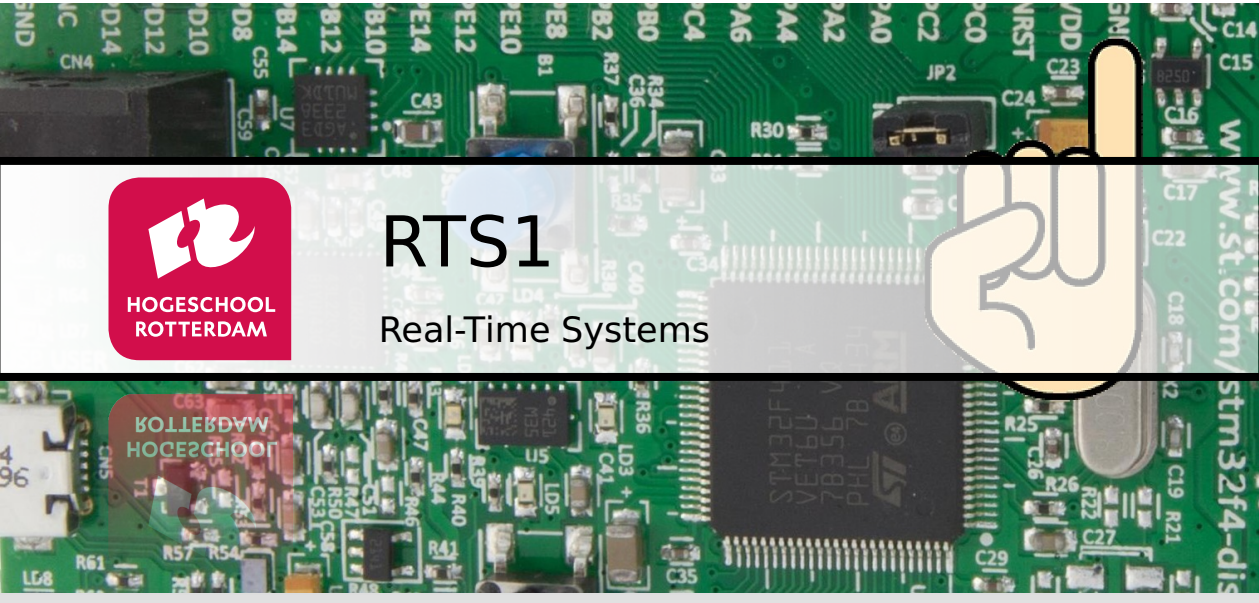


De LEGv7 architectuur en de Pinky instructieset

Versie 1.1c

J.Z.M. Broeders

Versiehistorie

Datum	Versie	Omschrijving	Auteur
04-09-2022	1.1c[1]	`B.N` instructie toegevoegd en diverse fouten in tabellen verbeterd.	BroJZ
03-07-2022	1.0	Eerste versie.	BroJZ

[1] Toelichting versiecodering *A.Bc*: A = grote aanpassing, B = kleine aanpassing, c = taal- of wiskundige correcties.

Inhoudsopgave

1

Inleiding

Om volledig te begrijpen hoe een computerprogramma op computerhardware draait, is niet alleen kennis vereist van de computerarchitectuur, maar ook van de machinetaalinstructies die op deze architectuur uitgevoerd worden.

Machinetaalinstructies bestaan, zoals alle informatie in een computergeheugen, uit enen en nullen. Een leesbare vorm van deze machinetaalinstructies worden assembly instructies genoemd. In de praktijk gebruik je een compiler om een programma voor een embedded systeem geschreven in bijvoorbeeld C om te zetten naar machinecode. Met behulp van een assembler kun je een programma geschreven in assembly instructies omzetten naar machinecode. Met een linker kun je programmadelen geschreven in bijvoorbeeld C combineren met programmadelen die geschreven zijn in assembly code.

Door het gebruik van assembly code heb je volledige controle over wat de processor doet. Je kunt bijvoorbeeld bepaalde instructies uitvoeren die in een hogere programmeertaal onmogelijk zijn, zoals het rechtstreeks benaderen van de registers in de processor. Hiermee kun je bijvoorbeeld de 'context switch' van een operating system implementeren. Ook kun je bijvoorbeeld na een berekening eenvoudig bekijken of er een overflow is opgetreden, iets dat vanuit bijvoorbeeld de programmeertaal C niet (eenvoudig) kan.

Omdat we de concepten van computerarchitectuur uitleggen aan de hand van de ARM Cortex M4 processor [7] zullen we ook assemblerprogramma's schrijven voor deze processor. De ARM architectuur[2] is een zeer populaire processorarchitectuur die door zeer veel processor- en

[2] https://en.wikipedia.org/wiki/ARM_architecture_family

microcontrollerfabrikanten wordt toegepast[3]. De recente door ARM ontworpen processoren kunnen ruwweg worden ingedeeld in drie groepen [13]:

- ARM Cortex-**A**. Zogenoemde Application processoren. Dit zijn processoren die bedoeld zijn voor prestatie-intensieve systemen zoals telefoons, tablets, laptops en pc's. Deze processoren bevatten meerdere cores en zijn uitgerust met cache geheugens. De Cyclone V SoC 5CSEMA5F31C6[4] van Intel die zich op het DE1-SoC bord[5] van Terasic bevindt, die je bij de cursus HWP01 gebruikt, bevat bijvoorbeeld een dual core ARM Cortex-A9 processor. In de cursus CSC1 (in kwartaal twee van de minor) ga je hierop een Linux OS draaien.
- ARM Cortex-**R**. Zogenoemde Real-Time processoren. Deze processoren zijn bedoeld voor prestatie-intensieve real-time systemen zoals drive controllers, routers, modems, printers, media players, ABS systemen en motor management systemen.
- ARM Cortex-**M**. Zogenoemde microcontroller processor cores. De processorkernen zijn bedoeld voor embedded applicaties. Onder andere Renesas, Texas Instruments, NXP Semiconductors, Microchip Technology en STMicroelectronics brengen microcontrollers op de markt die voorzien zijn van een Cortex-M processorkern.

De Cortex-A en -R processorkernen ondersteunen twee verschillende instructiesets: de ARM en de zogenoemde Thumb instructieset. De ARM instructieset bestaat uit machinetaalinstructies die allemaal 32 bits groot zijn. De Thumb instructieset combineert 32-bits met 16-bits instructies. Programma's gecodeerd in de ARM instructieset leveren meer performance maar programma's gecodeerd in de Thumb instructieset nemen minder geheugen in beslag. De Cortex-M processorkern ondersteunt alleen de Thumb instructieset.

De Cortex M serie bestaat uit verschillende subgroepen[6]:

- De Cortex-M0 en Cortex-M0+ processorkernen zijn geoptimaliseerd voor chips met de laagste prijs en het laagste energiegebruik. Ze implementeren de ARMv6-M architectuur[1].
- De Cortex-M1 processorkern is gebaseerd op de Cortex-M0 en geoptimaliseerd om op een FPGA (als softcore) te gebruiken.
- De Cortex-M3 processorkern implementeert de ARMv7-M architectuur [2].

[3] https://en.wikipedia.org/wiki/List_of_products_using_ARM_processors

[4] https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v.html

[5] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836

[6] https://developer.arm.com/ip-products/processors/cortex-m/
https://en.wikipedia.org/wiki/ARM_Cortex-M

- De Cortex-M4 processorkern is in principe een Cortex-M3 uitgebreid met DSP hardware en DSP machinetaalinstructies. De Cortex-M4F variant bevat ook een FPU (Floating Point Unit) die 32-bits floating point berekeningen kan uitvoeren. In de programmeertaal C worden 32-bits floating point variabelen aangegeven met het type **`float`**.
- De Cortex-M7 processorkern is een high performance versie van de Cortex-M4. Er is ook een Cortex-M7F die voorzien is van een FPU die optioneel ook 64-bits floating point berekeningen kan uitvoeren. In de programmeertaal C worden 64-bits floating point variabelen aangegeven met het type **`double`**.
- De Cortex-M23 processorkern is een modernere versie van de Cortex-M0+ en implementeert de ARMv8-M architectuur [3]
- De Cortex-M33, Cortex-M55 en Cortex-M85 processorkernen zijn modernere versies van de Cortex-M3, -M4 en -M7 en implementeren de ARMv8-M architectuur.

In deze cursus gebruik je het STM32F411E-DISCO ontwikkelbord[7] dat voorzien is van een STM32F411VET6 microcontroller[11, 12]. Deze microcontroller bevat een Cortex-M4F processorkern. Elektrotechniek studenten van Hogeschool Rotterdam hebben in hun opleiding al gebruik gemaakt van een CC3220S-LAUNCHXL ontwikkelbord[8] dat voorzien is van een CC3220S microcontroller[4, 5]. Deze microcontroller bevat een Cortex-M4 processorkern (zonder FPU).

De ARM Cortex-M4 gebruikt de ARMv7 Thumb instructieset [2] die bestaat uit 16-bits en 32-bits instructies. Deze instructieset is vrij groot [6] en bevat ongeveer 280 instructies. Daarom hebben we een kleine subset gedefinieerd die we LEGv7 Pinky noemen die slechts uit drieëndertig 16-bits instructies bestaat. De LEGv7 architectuur en Pinky instructieset worden in dit document gespecificeerd. Omdat de LEGv7 architectuur een subset is van de ARMv7 architectuur en de Pinky instructieset een subset is van de Thumb instructieset, kunnen programma's geschreven in LEGv7 Pinky draaien op elke Cortex-M4 processor. Je kunt dus het STM32F411E-DISCO bord gebruiken om jouw Pinky assemblertaalprogramma's te testen.

[7] https://www.st.com/en/evaluation-tools/32f411ediscovery.html

[8] https://www.ti.com/tool/CC3220S-LAUNCHXL

2

De LEGv7 architectuur

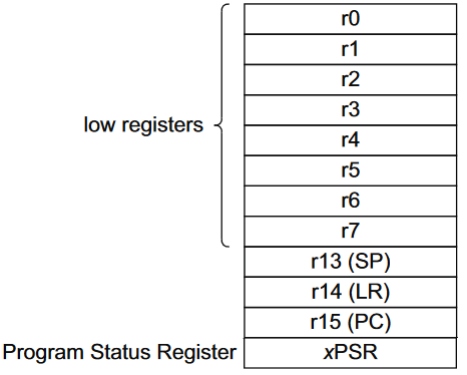
De LEGv7 architectuur is een subset van de ARMv7 architectuur[2, 7, 10]. De LEGv7 is een zogenoemde Harvard architectuur[9]. Dat wil zeggen dat de processor twee aparte bussen naar het geheugen heeft en gelijktijdig instructies en data kan ophalen. Beide bussen zijn 32 bits breed. Omdat de instructies in de Pinky instructieset 16 bits breed zijn, kan de processor twee instructies in één buscycle ophalen.

De LEGv7 architectuur is een 32-bits architectuur. Dat wil zeggen dat de ALU (Arithmetic and Logic Unit) 32 bits breed is. De ALU kan in één klokcycle twee 32-bits getallen bewerken (bijvoorbeeld optellen). De data- en de adresbussen zijn ook 32 bits breed. De processor kan dus $2^{32} = 4\,\text{GB}$ geheugenplaatsen adresseren. Elke geheugenplaats bevat één byte (8 bits). Een 32-bits integer neemt dus vier geheugenplaatsen in beslag, maar kan, als het uitgelijnd (Engels: aligned) is op een adres dat deelbaar is door vier, in één buscycle opgehaald worden.

De LEGv7 architectuur is een zogenoemde RISC (Reduced Instruction Set Computer) architectuur ook wel een load-store architectuur genoemd. Dat wil zeggen dat de processor geen bewerkingen kan uitvoeren op data die zich in het geheugen bevindt. Deze data moet altijd eerst worden geladen (Engels: load) in een register en kan dan pas bewerkt worden. Het resultaat kan vervolgens (indien gewenst) weer opgeslagen (Engels: store) worden in het geheugen.

[9] https://en.wikipedia.org/wiki/Harvard_architecture

De registers van de LEGv7 architectuur zijn gegeven in figuur 2.1. Alle registers zijn 32 bits breed. De ARMv7 architectuur heeft naast de low registers ook nog high registers (r8 tot en met r12) [7, pagina 2-3].



Figuur 2.1: De registers van de LEGv7 architectuur.

De registers R0[10] tot en met R7 zijn general purpose registers die gebruikt worden om data tijdelijk op te slaan.

De stack pointer (SP = R13) wijst naar een specifiek deel van het RAM geheugen dat gereserveerd is voor de zogenoemde stack. De processor gebruikt deze stack om bepaalde registers op te slaan als er een interrupt optreedt. Als programmeur kun je de stack gebruiken om data tijdelijk op te slaan. Omdat alle registers 32 bits breed zijn, moet de SP altijd uitgelijnd (Engels: aligned) zijn op een adres dat deelbaar is door vier. Bit 0 en bit 1 van de SP zullen dus altijd 0 zijn.

Wat we in Python en C een functie noemen wordt in assemblertaal een subroutine genoemd. Het Link register (LR = R14) wordt gebruikt om het terugkeeradres op te slaan als een subroutine wordt aangeroepen.

De Program Counter (PC = R15) houdt bij waar de processor met het uitvoeren van instructies uit het geheugen is gebleven. Omdat een instructie uit 16 bits bestaat en altijd uitgelijnd (Engels: aligned) moet zijn op een adres dat deelbaar is door twee, zal bit 0 van de PC altijd 0 zijn.

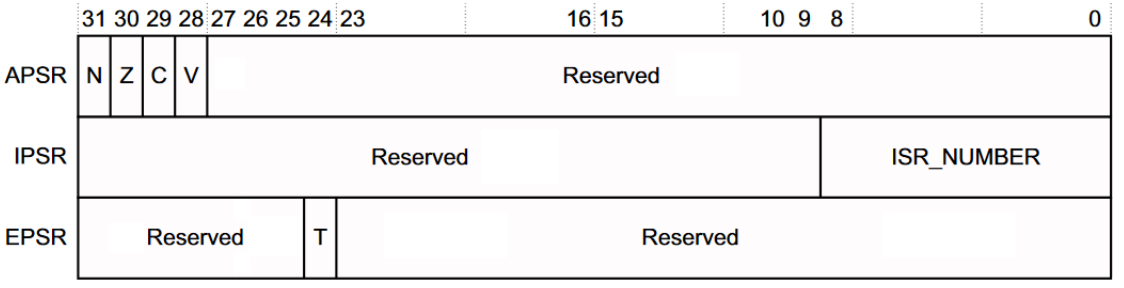
Het Program Status Register xPSR kan niet rechtstreeks gelezen of beschreven worden met behulp van Pinky instructies. De bits in dit register worden door verschillende Pinky instructies

[10] De assemblertaal van de LEGv7 architectuur is hoofdletterongevoelig (Engels: case insensitive) dus register R0 is hetzelfde als r0.

(indirect) aangepast of gebruikt. Dit register bestaat uit drie elkaar overlappende registers, zie figuur 2.2:

- Application Program Status Register (APSR);
- Interrupt Program Status Register (IPSR);
- Execution Program Status Register (EPSR).

Het xPSR register van de ARMv7 architectuur bevat nog een aantal bits die we niet in de LEGv7 architectuur hebben opgenomen [7, pagina 2-4].



Figuur 2.2: Het Program Status register van de LEGv7 architectuur.

Het ASPR register bevat de volgende bits, ook wel condition flags genoemd:

- Negative (N) flag; Geeft bij bepaalde instructies aan dat het resultaat negatief is (bit 31 = 1).
- Zero (Z) flag; Geeft bij bepaalde instructies aan dat het resultaat nul is (bit 31 tot en met bit 0 zijn allemaal 0).
- Carry (C) flag; Geeft bij bepaalde instructies aan dat er een carry (of een borrow) is opgetreden (het *unsigned* resultaat past niet in 32 bits).
- Overflow (V) flag; Geeft bij bepaalde instructies aan dat er een overflow is opgetreden (het *signed* resultaat past niet in 32 bits).

Deze flags kunnen door bepaalde instructies gebruikt worden om bijvoorbeeld te bepalen of er gesprongen moet worden in het programma.

Het IPSR register bevat het nummer van de interrupt service routine (ISR) die op dit moment wordt uitgevoerd [7, pagina 2-6]. Als de processor geen ISR uitvoert, de processor bevindt zich dan in de zogenoemde *Thread mode*, dan is het ISR_NUMMER nul. Als de processor wel

een ISR uitvoert, de processor bevindt zich dan in de zogenoemde *Handler mode*, dan geeft het `ISR_NUMMER` aan welke interrupt of exception[11] wordt uitgevoerd.

Het `EPSR` register bevat het `T` bit dat aangeeft of de processor ARM (`T=0`) dan wel Thumb instructies (`T=1`), of in ons geval Pinky instructies, aan het uitvoeren is. In een Cortex-M processor moet dit bit altijd één zijn omdat deze de ARM instructieset niet ondersteund. Als je de `PC` laad met een nieuwe waarde, dan kun je op een slimme manier kiezen of de processor daar ARM of Thumb instructies moet gaan ophalen. De instructies zijn namelijk altijd 16 bits of 32 bits breed. Dat wil zeggen dat bit 0 van de `PC` altijd met een 0 geladen moet worden. Bij het laden van de `PC` wordt bit 0 niet in de `PC` gezet maar in het `T` bit in het `EPSR` register. Bij de Cortex-A en Cortex-R kan op deze manier overgeschakeld (Engels: exchange) worden naar de gewenste instructieset. Omdat de Cortex-M alleen maar Thumb instructies ondersteund, dient de `PC` bij een Cortex-M altijd met een oneven adres geladen worden (zodat het `T` bit 1 gemaakt wordt).

[11] Interrupts die intern in de processor zelf worden opgewerkt, bijvoorbeeld omdat er een interne fout is opgetreden, worden door Arm exceptions genoemd.

3

De Pinky instructieset

De Pinky instructieset is een subset van de Thumb instructieset [2, 7, 10]. Programma's geschreven in LEGv7 Pinky kunnen geassembleerd worden in de STM32Cube IDE[12] met de daarin aanwezige GNU assembler[13] en uitgevoerd worden op een STM32 microcontroller. Alle instructies in deze instructieset eindigen op **.N**[14] dit zorgt ervoor dat de assembler een foutmelding geeft als de instructie niet in 16 bits past. De instructie **CMP.N** en alle instructies die eindigen op **S.N** werken de flags in het CPSR register bij. De overige instructies werken deze flags niet bij.

Een instructie bestaat uit een operator en maximaal drie zogenoemde operands. De operator geeft de bewerking weer die wordt uitgevoerd en de operands geven de data weer waarop de bewerking wordt uitgevoerd. Instructies in machinecode bestaan uit een aantal nullen en enen en zijn dus lastig leesbaar. Wij maken daarom gebruik van zogenoemde assembly code. Een instructie in assembly code kan door een zogenoemde assembler, één op één, vertaald worden in machinecode. Elke processorfamilie heeft zijn eigen machinetaal en dus ook zijn eigen assemblertaal. Een regel assembly code uit een assemblertaalprogramma voor de Cortex-M4 is als volgt opgebouwd:

```
label: mnemonic dst, src1, src2 // commentaar
```

De verschillende delen worden hieronder verklaard:

[12] https://www.st.com/en/development-tools/stm32cubeide.html

[13] https://sourceware.org/binutils/docs-2.38/as.pdf

[14] **N** staat voor *Narrow* en geeft aan dat de instructie uit 16 bits bestaat. In de ARMv8 Thumb instructieset zijn ook *Wide* (32-bits) instructies opgenomen die eindigen op **.W**.

- `label`. Een label is optioneel en kan gebruikt worden om naar een bepaalde instructie te refereren.
- `mnemonic`. De mnemonic[15] is een afkorting waarmee wordt aangegeven wat de instructie doet. Bijvoorbeeld de instructie **`ADD.N`** voert een optelling (ADDition) uit en de instructie **`LDR.N`** laadt een general purpose register met een waarde uit het geheugen (LoaD Register).
- `dst`, `src1`, `src2`. De destination (bestemming) en source (bron) operands geven aan op welke data de instructie wordt uitgevoerd en waar het resultaat wordt weggeschreven. De verschillende manieren waarop de destination en source operands kunnen worden gedefinieerd worden addressing modes genoemd. Sommige instructies hebben geen drie, maar slechts twee, één of geen operands.

  De instructies in de Pinky instructieset gebruiken vijf verschillende addressing modes:

  - Register mode. In dit geval is de operand een register.
    Bijvoorbeeld: de instructie **`SUB.N`** `R2, R0, R1` voert de volgende bewerking uit: `R2`=`R0`−`R1`.
  - Immediate mode. In dit geval is de operand een constant getal. Deze addressing mode kan (uiteraard) alleen als source operand gebruikt worden. Bijvoorbeeld: de instructie **`SUB.N`** `R2, R0, #1` voert de volgende bewerking uit: `R2`=`R0`−1.
  - Memory mode. In dit geval is de operand een geheugenadres waarop een load of store wordt uitgevoerd. In dit geval geeft register `src1` het basisadres en `src2` de offset.
    Bijvoorbeeld: de instructie **`LDR.N`** `R3, [R4, #16]` telt 16 (de offset) op bij de inhoud van register `R4` (het basisadres). Dit adres wordt vervolgens gebruikt om een 32-bits getal in te lezen vanuit het geheugen. Dit getal wordt tenslotte opgeslagen op in register `R3`. De constante offset moet deelbaar zijn door vier. In plaats van een constante offset kan ook een variabele offset gebruikt worden die is opgeslagen in een register.
    Bijvoorbeeld: **`LDR.N`** `R3, [R4, R5]`, de inhoud van `R5` wordt nu als offset gebruikt.
  - Literal mode. Deze mode kan alleen gebruikt worden bij een load instructie. Met behulp van deze adressing mode kan een 32-bits getal of adres in een register geladen worden.

[15] Het Engelse woord mnemonic betekent geheugensteuntje.

Bijvoorbeeld: `LDR.N R6, =0x12345678`. Hoe dit mogelijk is met een instructie die zelf maar 16 bits groot is, wordt uitgelegd in paragraaf 3.8.

- Register-list mode. Deze addressing mode wordt gebruikt door de `PUSH.N` en `POP.N` instructies waarmee een aantal registers op de stack gezet of er vanaf gehaald kunnen worden.
  Bijvoorbeeld: `POP.N {R0-R4,R6}` laadt achtereenvolgens `R0` tot en met `R4` en `R6` vanaf de stack.

- `// commentaar`. Alle tekst na een dubbele slash wordt gezien als commentaar.

Assembly code is overigens niet hoofdlettergevoelig. De instructie `SUB.N R2, R0, R1` kun je dus ook schrijven als `sub.n r2, r0, r1`.

De Pinky instructieset bestaat uit 33 verschillende instructies die qua functionaliteit in te delen zijn in 11 groepen. Een volledige lijst van alle Pinky instructies, op alfabetische volgorde, vind je in hoofdstuk A.

3.1 De rekenkundige instructies

Met behulp van deze instructies kunnen rekenkundige bewerkingen worden uitgevoerd. Zie figuur 3.1. Deze instructies werken de status flags N, Z, C en V bij en worden in één klokcycle uitgevoerd. <imm3> is een 3-bits unsigned offset en <imm8> is een 8-bits unsigned offset.

Instructie	Betekenis	N	Z	C	V	Cycles
ADDS.N <Rdn>,#<imm8>	<Rdn> ← <Rdn> + <imm8>	↕	↕	↕	↕	1
ADDS.N <Rd>,<Rn>,#<imm3>	<Rd> ← <Rn> + <imm3>	↕	↕	↕	↕	1
ADDS.N <Rd>,<Rn>,<Rm>	<Rd> ← <Rn> + <Rm>	↕	↕	↕	↕	1
SUBS.N <Rdn>,#<imm8>	<Rdn> ← <Rdn> - <imm8>	↕	↕	↕	↕	1
SUBS.N <Rd>,<Rn>,#<imm3>	<Rd> ← <Rn> - <imm3>	↕	↕	↕	↕	1
SUBS.N <Rd>,<Rn>,<Rm>	<Rd> ← <Rn> - <Rm>	↕	↕	↕	↕	1

Figuur 3.1: Rekenkundige instructies.

3.2 De logische instructies

Met behulp van deze instructies kunnen bitwise logische bewerkingen worden uitgevoerd. Zie figuur 3.2. Deze instructies werken de status flags N en Z bij, maar de C en V flag behouden hun waarde.

Instructie	Betekenis	N	Z	C	V	Cycles
ANDS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> AND <Rm>	↕	↕	-	-	1
EORS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> EOR <Rm>	↕	↕	-	-	1
ORRS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> OR <Rm>	↕	↕	-	-	1

Figuur 3.2: Logische (bitwise) instructies.

3.3 De schuif-instructies

Met behulp van deze zogenoemde 'Logic Shift Left/Right' instructies kan de inhoud van een register naar links of rechts geschoven worden. <imm5> is een 5-bits unsigned waarde die bepaalt over hoeveel posities de inhoud verschoven wordt. Het laatste bit dat uit het register wordt geschoven komt in de C-flag terecht. Zie figuur 3.3.

Instructie	Betekenis	N	Z	C	V	Cycles
LSLS.N <Rd>,<Rm>,#<imm5>	<Rd> ← <Rm> << <imm5>	↕	↕	↕	-	1
LSRS.N <Rd>,<Rm>,#<imm5>	<Rd> ← <Rm> >> <imm5>	↕	↕	↕	-	1

Figuur 3.3: Schuif-instructies.

3.4 De kopieer-instructies

Met behulp van deze zogenoemde 'Move' instructies kan de inhoud van een register of een 8-bits unsigned constante gekopieerd worden naar het destination register. Zie figuur 3.4.

Instructie	Betekenis	N	Z	C	V	Cycles
MOVS.N <Rd>,#<imm8>	<Rd> ← <imm8>	↕	↕	-	-	1
MOVS.N <Rd>,<Rm>	<Rd> ← <Rm>	↕	↕	-	-	1

Figuur 3.4: Kopieer-instructies.

3.5 De vergelijken met nul en spring voorwaarts instructies

Met behulp van deze 'Compare Branch on (Not) Zero' instructies kan *voorwaarts* gesprongen worden in het programma, als de inhoud van een register (niet) nul is. Zie figuur 3.5. De instructie waarnaartoe gesprongen wordt, wordt aangegeven met een label, zie pagina 13.

De executietijd van deze instructies is 1 als er niet wordt gesprongen en 1+P als er wel wordt gesprongen. Waarbij P kan variëren tussen 1 en 3.[16]

Instructie	Betekenis	N	Z	C	V	Cycles
CBNZ.N <Rn>,<label>	if (<Rn> != 0) PC ← label	-	-	-	-	1(+P)
CBZ.N <Rn>,<label>	if (<Rn> == 0) PC ← label	-	-	-	-	1(+P)

Figuur 3.5: Spring als de inhoud van een register gelijk (of ongelijk) is aan nul.

3.6 De spring-instructie

Met deze 'Branch' kan *voorwaarts* of *achterwaarts* worden gesprongen in het programma. Zie figuur 3.6. De instructie waarnaartoe gesprongen wordt, wordt aangegeven met een label, zie pagina 13. De executietijd van deze instructies is 1+P, waarbij P kan variëren tussen 1 en 3.[17]

Instructie	Betekenis	N	Z	C	V	Cycles
B.N <label>	PC ← <label>	-	-	-	-	1+P

Figuur 3.6: Spring-instructie.

3.7 De vergelijken en springen als . . . instructies

Met behulp van deze 'Compare' instructies kan de inhoud van een register worden vergeleken met een 8-bits unsigned constante of met de inhoud van een ander register. Bij de **CMP.N** instructie wordt het resultaat niet opgeslagen maar verder is deze instructie gelijk aan de **SUBS.N** instructie. De **CMP.N** instructie werkt wel de vlagen in het APSR bij.[18] Vervolgens kan met een 'Branch on condition <c>' *voorwaarts* of *achterwaarts* worden gesprongen in het programma, als een bepaalde conditie <c> waar is. Zie figuur 3.7. De instructie waarnaartoe gesprongen wordt, wordt aangegeven met een label, zie pagina 13. De executietijd van deze

[16] Het aantal clockcycles dat nodig is om de pipeline opnieuw te vullen [8, pagina 3-4]. In de praktijk blijkt, bij het uitvoeren van deze instructies op de STM32F411VET6 microcontroller, P altijd 1 te zijn.

[17] Zie voetnoot 16.

[18] Deze instructie zou dus eigenlijk CMPS.N moeten heten om aan te geven dat de flags in het APSR bijgewerkt worden.

instructies is 1 als er niet wordt gesprongen en 1+P als er wel wordt gesprongen. Waarbij P kan variëren tussen 1 en 3.[19]

Instructie	Betekenis	N	Z	C	V	Cycles
B<c>.N <label>	if (<c>) PC ← <label>	-	-	-	-	1(+P)
CMP.N <Rn>,#<imm8>	<Rn> - <imm8>	↕	↕	↕	↕	1
CMP.N <Rn>,<Rm>	<Rn> - <Rm>	↕	↕	↕	↕	1

Figuur 3.7: Vergelijken en springen als een bepaalde conditie waar is.

De verschillende condities <c> waarop gesprongen kan worden zijn weergegeven in figuur 3.8. De `B<c>.N` kan op elke plaats in een programma gebruikt worden, maar als deze instructie na een `CMP.N` instructie wordt gebruikt, dan hebben bepaalde condities een specifieke betekenis. Dit is weergegeven in figuur 3.9. Let erop dat er *verschillende* condities zijn voor het vergelijken van *signed* en *unsigned* getallen.

cond	<c>	betekenis	conditie	betekenis na CMP Rn,op2
0000	EQ	Equal	Z == 1	Rn == op2
0001	NE	Not Equal	Z == 0	Rn != op2
0010	CS / LO	Carry Set / LOwer	C == 1	Rn < op2, unsigned
0011	CC / HS	Carry Clear / Higher or Same	C == 0	Rn >= op2, unsigned
0100	MI	MInus	N == 1	
0101	PL	Plus	N == 0	
0110	VS	Overflow Set	V == 1	
0111	VC	Overflow Clear	V == 0	
1000	HI	Higher	C == 1 and Z == 0	Rn > op2, unsigned
1001	LS	Lower or Same	C == 0 or Z == 1	Rn <= op2, unsigned
1010	GE	Greater than or Equal	N == V	Rn >= op2, signed
1011	LT	Less Than	N != V	Rn < op2, signed
1100	GT	Greater Than	Z == 0 and N == V	Rn > op2, signed
1101	LE	Less than or Equal	Z == 1 or N != V	Rn <= op2, signed

Figuur 3.8: Verschillende condities waarop gesprongen kan worden.

[19] Zie voetnoot 16.

<c>	betekenis	betekenis na CMP Rn,op2	
		unsigned	signed
EQ	Equal	Rn == op2	
NE	Not Equal	Rn != op2	
CS / LO	Carry Set / LOwer	Rn < op2	
CC / HS	Carry Clear / Higher or Same	Rn >= op2	
HI	Higher	Rn > op2	
LS	Lower or Same	Rn <= op2	
GE	Greater than or Equal		Rn >= op2
LT	Less Than		Rn < op2
GT	Greater Than		Rn > op2
LE	Less than or Equal		Rn <= op2

Figuur 3.9: Specifieke betekenis van de condities <c> van de `B<c>.N` instructie na een `CMP.N` instructie.

3.8 Load en store instructies

Met behulp van deze instructies kan informatie ingelezen worden vanuit, en weggeschreven worden naar, het geheugen. Deze 'STore Register' instructies maken gebruik van de memory addressing mode, zie pagina 13. De 'LoaD Register' instructies maken ook gebruik van deze addressing mode of van de literal addressing mode, zie pagina 13. Zie figuur 3.10. In deze tabel wordt bij het aantal cycles de variabele Q gebruikt. Hierbij geldt dat Q één is, als de voorafgaande instructie een `LDR.N` instructie is en als de huidige `LDR.N` of `STR.N` instructie het register dat in de voorafgaande instructie wordt geladen, niet gebruikt in de adresberekening [8, pagina 3-12]. In alle andere gevallen geldt dat Q nul is.

Instructie	Betekenis	N	Z	C	V	Cycles
LDR.N <Rt>,=<32-bits literal>	<Rt> ← <32-bits literal>	-	-	-	-	2-Q
LDR.N <Rt>, [<Rn>,#<imm7>]	<Rt> ← [<Rn> + <imm7>]	-	-	-	-	2-Q
LDR.N <Rt>,[<Rn>,<Rm>]	<Rt> ← [<Rn> + <Rm>]	-	-	-	-	2-Q
STR.N <Rt>, [<Rn>,#<imm7>]	<Rt> → [<Rn> + <imm7>]	-	-	-	-	1
STR.N <Rt>,[<Rn>,<Rm>]	<Rt> → [<Rn> + <Rm>]	-	-	-	-	2-Q

Figuur 3.10: Load en store instructies.

Met behulp van de literal addressing mode kun je een 32-bits getal of adres in een register laden. Bijvoorbeeld: `LDR.N R6, =0x12345678`. Deze instructie wordt op een bijzondere manier gecodeerd als een 16-bits machinecode instructie door de assembler. De constante

32-bits waarde (in het voorbeeld `0x12345678`) wordt een 'literal' genoemd en wordt door de assembler een stukje verderop in het geheugen geplaatst in een zogenoemde 'literal section'. Vervolgens wordt de instructie omgezet naar een load instructie die memory addressing gebruikt waarbij de programcounter (`PC`) als basisregister wordt gebruikt. De offset wordt door de assembler berekend. Dus de instructie:

```
LDR.N   R6, =0x1234567
```

Wordt door de assembler omgezet in:

```
LDR.N   R6, [PC, #offset]
//...
.word   0x1234567
```

De assembler directive **`.word`** plaatst een 32-bits constant getal in het geheugen. Omdat de offset beperkt is tot een 10-bits unsigned getal, zijn soms meerdere literal sections nodig in een programma.

3.9 De doe niets instructie

Met behulp van deze zogenoemde 'No OPeration' instructie kan de processor één klokcycle wachten. Zie figuur 3.11.

Instructie	Betekenis	N	Z	C	V	Cycles
NOP.N	-	-	-	-	-	1

Figuur 3.11: No operation instructie.

3.10 Spring naar subroutine en return instructies

Wat we in Python en C een functie noemen wordt in assemblertaal een subroutine genoemd. Met behulp van de zogenoemde 'Branch Link en eXchange' instructie kan naar een subroutine worden gesprongen. Het terugkeeradres wordt opgeslagen in het link register (`LR`). Let erop dat het adres in register `Rm` *oneven* moet zijn, zodat het T-bit in het EPSR één blijft en de processor verdergaat met het uitvoeren van Thumb (of in ons geval Pinky) instructies, zie pagina 11. Als het adres in register `Rm` even is, dan zal de processor een exception genereren omdat de Cortex-M de ARM instructieset niet ondersteunt en dus niet kan overschakelen (Engels: exchange) naar deze instructieset.

Aan het einde van een subroutine kan met de 'Branch eXchange' teruggekeerd worden (Engels: return) naar de plaats waar de subroutine is aangeroepen. Dit terugkeeradres is automatisch in het link register opgeslagen door de 'Branch Link en eXchange' instructie waarmee de subroutine is aangeroepen. Bit 0 is door deze instructie ook automatisch 1 gemaakt, zodat de processor na de return verdergaat met het uitvoeren van Thumb (of in ons geval Pinky) instructies. Zie figuur 3.12.

Het uitvoeren van deze instructies duurt 1+P klokcycles, waarbij P kan variëren tussen 1 en 3.[20]

Instructie	Betekenis	N	Z	C	V	Cycles
BLX.N <Rm>	LR = addr_of_inst + 3, PC ← Rm	-	-	-	-	1+P
BX.N LR	PC ← LR - 1, Tbit = 1	-	-	-	-	1+P

Figuur 3.12: Spring naar subroutine en return instructies.

3.11 Stack instructies

De stack pointer (SP) wijst naar een specifiek deel van het RAM geheugen dat gereserveerd is voor de zogenoemde stack. Bij de LEGv7 architectuur wijst de SP naar de *laatst gevulde* plaats en groeit de stack naar *beneden* (richting adres 0). De processor gebruikt deze stack om bepaalde registers op te slaan als er een interrupt optreedt. Als programmeur kun je de stack gebruiken om data tijdelijk op te slaan. Omdat alle registers 32 bits breed zijn, moet de SP altijd uitgelijnd (Engels: aligned) zijn op een adres dat deelbaar is door vier. Er zijn verschillende instructies die gebruikt kunnen worden om de stack te bewerken. Zie figuur 3.13.

Met de **SUB.N** instructie kun je (vrije) ruimte op de stack creëren door de stack pointer te verlagen. Dit kun je bijvoorbeeld doen in het begin van een subroutine om ruimte te maken waarin je lokale variabelen kunt opslaan (die niet in de registers passen). Met de **ADD.N** instructie kun je de met **SUB.N** gecreëerde ruimte weer terugwinnen door de stack pointer te verhogen. Dit kun je bijvoorbeeld doen aan het einde van een subroutine om de op de stack opgeslagen lokale variabelen weer vrij te geven.

Met de **LDR.N** instructie kun je door de memory addressing mode, zie pagina 13, te gebruiken, met de SP als basisadres, data vanaf de stack in een register laden. Met de **STR.N** instructie

[20] Zie voetnoot 16.

Instructie	Betekenis	N	Z	C	V	Cycles
ADD.N SP,SP,#<imm9>	SP ← SP + <imm9>	-	-	-	-	1
LDR.N <Rt>,[SP,#<imm10>]	<Rt> ← [SP + <imm10>]	-	-	-	-	2
POP.N {<register_list>}	Pop from Stack <register_list>	-	-	-	-	1+N(+P)
PUSH.N {<register_list>}	Push to Stack <register_list>	-	-	-	-	1+N
STR.N <Rt>,[SP,#<imm10>]	<Rt> → [SP + <imm10>]	-	-	-	-	1
SUB.N SP,SP,#<imm9>	SP ← SP - <imm9>	-	-	-	-	1

Figuur 3.13: Stack instructies.

kun je door de memory addressing mode te gebruiken, met de SP als basisadres, data vanuit een register op de stack opslaan.

Met de **PUSH.N** instructie kunnen één of meerdere registers op de stack worden geplaatst. Daarbij wordt gebruik gemaakt van de register-list addressing mode, zie pagina 14. Je kunt een selectie van de general purpose registers en het LR pushen. De registers worden altijd in een vaste volgorde gepusht: LR, R7, . . . , R0 onafhankelijk van de volgorde waarin ze in de register list zijn opgenomen. De executietijd is 1+N klokcycles, waarbij N het aantal registers is dat gepusht moet worden. Met de **POP.N** instructie kunnen één of meerdere registers van de stack worden gehaald. Daarbij wordt gebruik gemaakt van de register-list addressing mode. Je kunt een selectie van de general purpose registers en de PC poppen. De registers worden altijd in een vaste volgorde gepopt: R0, R1, . . . , R7 en PC onafhankelijk van de volgorde waarin ze in de register list zijn opgenomen. De executietijd is 1+N(+P) klokcycles, waarbij N het aantal registers is dat gepopt moet worden. De +P moet alleen bij de executietijd worden opgeteld als de PC wordt gepopt, waarbij P kan variëren tussen 1 en 3.[21] Dat je het adres dat zich in het LR bevindt kan pushen en later in de PC kan poppen is erg handig bij het implementeren van een subroutine die zelf ook weer een of meerdere subroutines aanroept.

[21] Zie voetnoot 16.

4

Arm Application Binary Interface

Arm heeft een zogenoemde ABI (Application Binary Interface) gedefinieerd [9]. Machinecode die aan deze definitie voldoet kan met elkaar gecombineerd worden, ook als deze machinecode door verschillende tools (compilers of assemblers) gegenereerd is. In deze standaard is onder andere gedefinieerd hoe parameters aan functies[22] moeten worden doorgegeven en hoe de returnwaarde van een functie weer teruggegeven moet worden[23]. Ook wordt aangegeven welke registers aangepast mogen worden in een functie en welke registers aan het einde van de functie dezelfde waarde moeten hebben als aan het begin van de functie. De Arm ABI specificeert het volgende:

- registers `R0` tot en met `R3` kunnen gebruikt worden om parameters door te geven aan een functie en mogen door de functie aangepast worden;
  - Als je een functie aanroept, moet je er dus vanuit gaan dat `R0` tot en met `R3` gewijzigd zijn.
  - Je mag, in assembly code, voor deze registers `R0` tot en met `R3` ook respectievelijk de namen `A1`[24] tot en met `A4` gebruiken.
- `R0` en `R1` kunnen gebruikt worden om een returnwaarde terug te geven;
  - Als de returnwaarde in 32 bits past dan wordt `R0` gebuikt

[22] In de Arm documentatie wordt, wat in C en Python een functie genoemd wordt, een *procedure* genoemd.

[23] https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aapcs32.pdf

[24] A staat voor Argument. Het is wel vreemd dat de nummering bij 1 (en niet bij 0) begint.

- Als een 64 bits resultaat moet worden teruggegeven dan worden R1 (most significant deel) en R0 (least significant deel) gebruikt.

- registers R4 tot en met R7 kunnen gebruikt worden om variabelen op te slaan die behouden worden als je functies aanroept.
  - Als je een functie aanroept, kun je er dus op rekenen dat R4 tot en met R7 ongewijzigd zijn.
  - Als je deze registers in een functie gebruikt, dan moet je er voor zorgen dat de oorspronkelijke waarde voor het einde van de functie weer in het register wordt geplaatst. Een goede plaats om zo'n register tijdelijk op te slaan, is de stack.
  - Je mag, in assembly code, voor deze registers R4 tot en met R7 ook respectievelijk de namen V1[25] tot en met V4 gebruiken.

Als je een programma geheel in assembler schrijft, dan hoef je jezelf niet te conformeren aan deze ABI. Maar als je jouw assembly code aan wilt kunnen roepen vanuit een hogere programmeertaal (bijvoorbeeld C) en vice versa of als je functies uit een library wilt aanroepen, dan moet je jezelf wel conformeren aan deze ABI.

[25] V staat voor Variabele. Het is wel vreemd dat de nummering bij 1 (en niet bij 0) begint.

5

Instructie encoding

Om goed te begrijpen hoe de processor in staat is om Pinky assembly code instructies uit te voeren, moet je begrijpen hoe deze instructies in machinecode vertaald worden. Dit wordt ook wel instructie encoding genoemd. Werk dat we normaal gesproken aan de assembler overlaten.

De Pinky instructieset bestaat uit 33 verschillende instructies die qua codering in te delen zijn in 10 groepen. Een volledige lijst met de codering van alle Pinky instructies, op alfabetische volgorde, vind je in hoofdstuk B. Voor het coderen van de `B<c>.N` instructie kun je hoofdstuk D gebruiken.

5.1 Instructies zonder addressing mode

Bij deze instructies wordt geen explicite addressing mode gebruikt. De vertaling van assembly naar machinecode is eenvoudig. Zie figuur 5.1.

instructies zonder operand	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BX.N LR	0	1	0	0	0	1	1	1	0	1	0	1	1	0	0	0
NOP.N	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Figuur 5.1: Instructies zonder addressing mode.

5.2 Instructies met register addressing mode

Bij deze instructies worden drie, twee of één operand(s) gebruikt die allemaal de register addressing mode, zie pagina 13 gebruiken. Elk register wordt gecodeerd met drie bits die binair het nummer van het betreffende register weergeven. Zie figuur 5.2.

register instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		
LDR.N <Rt>,[<Rn>,<Rm>]	0	1	0	1	1	0	0	Rm			Rn			Rt		
STR.N <Rt>,[<Rn>,<Rm>]	0	1	0	1	0	0	0	Rm			Rn			Rt		
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	Rm			Rdn		
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	Rm			Rdn		
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	Rm			Rdn		
CMP.N <Rn>,<Rm>	0	1	0	0	0	0	1	0	1	0	Rm			Rn		
BLX.N <Rm>	0	1	0	0	0	0	1	1	1	1	Rm			0	0	0

Figuur 5.2: Instructies met register addressing mode.

5.3 Instructies met 3-bits immediate addressing mode

Bij deze instructies wordt een 3-bits immediate operand gebruikt, zie pagina 13, en twee operands die de register addressing mode gebruiken. Zie figuur 5.3. De immediate operand is een 3-bits unsigned waarde (0 t/m 7).

imm3 instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,#<imm3>	0	0	0	1	1	1	0	imm3			Rn			Rd		
SUBS.N <Rd>,<Rn>,#<imm3>	0	0	0	1	1	1	1	imm3			Rn			Rd		

Figuur 5.3: Instructies met 3-bits immediate addressing mode.

5.4 Instructies met 5-bits immediate addressing mode

Bij deze instructies wordt een 5-bits immediate operand gebruikt en twee operands die de register addressing mode gebruiken. Zie figuur 5.4. De immediate operand is een 5-bits unsigned waarde (0 t/m 31).

De `MOVS.N <Rd>, <Rm>` instructie lijkt een vreemde eend in de bijt, maar zoals je in figuur 5.4 kunt zien is de machinecode van de instructie:

imm5 instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSLS.N <Rd>,<Rm>,#<imm5>	0	0	0	0	0	imm5					Rm			Rd		
MOVS.N <Rd>,<Rm>	0	0	0	0	0	0	0	0	0	0	Rm			Rd		
LSRS.N <Rd>,<Rm>,#<imm5>	0	0	0	0	1	imm5					Rm			Rd		
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt		
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt		

Figuur 5.4: Instructies met 5-bits immediate addressing mode.

```
MOVS.N  <Rd>, <Rm>
```

gelijk aan de machinecode van de instructie:

```
LSLS.N  <Rd>, <Rm>, #0
```

De `MOVS.N <Rd>, <Rm>` is dus eigenlijk helemaal geen aparte instructie, maar een schuifinstructie, zie paragraaf 3.3, die de inhoud van `<Rm>` nul plaatsen verschuift naar links en in `<Rd>` plaatst. Zo'n instructie wordt ook wel een pseudo-instructie genoemd.

Wat opvalt bij de codering van de load en store instructies, zie paragraaf 3.8 is dat de operand in assembly code 7 bits is (0 t/m 127), maar omdat de offset altijd deelbaar door vier moet zijn[26], hoeven de laagste twee bits niet opgeslagen te worden. De waarde van `imm5` wordt als volgt berekend: `imm5 = <imm7> / 4` waarbij `<imm7> modulo 4` nul moet zijn.

5.5 Conditionele instructies met 6-bits immediate addressing mode

Bij de 'Compare Branch on (Not) Zero' instructies wordt voorwaarts gesprongen in het programma als de inhoud van een register (niet) nul is, zie paragraaf 3.5. Het adres waarnaar eventueel gesprongen wordt, wordt in assembly code met behulp van een label opgegeven. In machinecode moet echter een 6-bits unsigned offset (0 t/m 63) worden opgegeven die aangeeft hoeveel *instructies* er vooruit moet worden gesprongen. Dit wordt gedaan door deze offset maal twee[27] bij de `PC` op te tellen. Deze offset bestaat uit twee delen: bit 9 van de machinecode is het MSB[28] en bit 7 t/m bit 3 van de opcode zijn de overige bits. Zie figuur 5.5. De offset wordt als volgt berekend:

[26] Een binair getal is deelbaar door vier als bit 0 en bit 1 beide nul zijn.

[27] Maal twee omdat elke instructie 16 bits = 2 bytes groot is.

[28] Most Significant Bit

`imm6 = i:imm5 = (<label> - (address_of_instruction + 4))/2`

imm6 conditionele instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CBZ.N <Rn>,<label>	1	0	1	1	0	0	i	1	imm5					Rn		
CBNZ.N <Rn>,<label>	1	0	1	1	1	0	i	1	imm5					Rn		

Figuur 5.5: Conditionele instructies met 6-bits immediate addressing mode.

5.6 Stack instructies met 7-bits immediate addressing mode

Er zijn twee assemblertaalinstructies waarmee je een 9-bits unsigned constante bij of van de stackpointer (SP) kunt optellen of aftrekken, zie paragraaf 3.11. De SP moet, zoals al eerder vermeldt, altijd uitgelijnd (Engels: aligned) zijn op een adres dat deelbaar is door vier. De constante die bij of van de SP wordt opgeteld of afgetrokken moet dus ook deelbaar zijn door vier[29]. De laagste twee bits van deze constante hoeven dus in de machinecode niet opgeslagen te worden. Zie figuur 5.4. De waarde van `imm7` kan als volgt berekend worden: `imm7 = <imm9> / 4` waarbij `<imm9> modulo 4` nul moet zijn.

imm7 stack instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD.N SP,SP,#<imm9>	1	0	1	1	0	0	0	0	0	imm7						
SUB.N SP,SP,#<imm9>	1	0	1	1	0	0	0	0	1	imm7						

Figuur 5.6: Stack instructies met 7-bits immediate addressing mode.

5.7 Instructies met 8-bits immediate addressing mode

Bij deze instructies wordt een 8-bits immediate operand gebruikt en een operand die de register addressing mode gebruikt. Zie figuur 5.7. De immediate operand is een 8-bits unsigned waarde (0 t/m 255).

Bij de assemblertaalinstructie **`LDR.N`** `<Rt>, =<32-bits literal>` wordt de literal addressing mode, zie pagina 13, gebruikt met een 32-bits constante (het mag ook een label zijn). Deze instructie wordt op een bijzondere manier gecodeerd als een 16-bits machinecode instructie door de assembler. De constante 32-bits waarde (de opgegeven constante of het adres waar het label naar verwijst) wordt door de assembler een stukje verderop in het geheugen geplaatst in een zogenoemde literal section. Vervolgens wordt de instructie omgezet naar

[29] Een binair getal is deelbaar door vier als bit 0 en bit 1 beide nul zijn.

imm8 instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVS.N <Rd>,#<imm8>	0	0	1	0	0	Rdn			imm8							
ADDS.N <Rdn>,#<imm8>	0	0	1	1	0	Rdn			imm8							
SUBS.N <Rdn>,#<imm8>	0	0	1	1	1	Rdn			imm8							
CMP.N <Rn>,#<imm8>	0	0	1	0	1	Rdn			imm8							
LDR.N <Rt>,=<label>	0	1	0	0	1	Rt			imm8							
LDR.N <Rt>,[SP,#<imm10>]	1	0	0	1	1	Rt			imm8							
STR.N <Rt>,[SP,#<imm10>]	1	0	0	1	0	Rt			imm8							

Figuur 5.7: Instructies met 8-bits immediate addressing mode.

een load instructie die memory addressing gebruikt waarbij de programcounter (PC) als basisregister wordt gebruikt. Dus de instructie:

```
LDR.N   R6, =label
```

Wordt door de assembler omgezet in:

```
LDR.N   R6, [PC, #offset]
\\...
.word   label
```

De assembler directive `.word` plaats een 32-bits constant getal in het geheugen. De offset wordt als volgt berekend:

```
imm8 = (address_of_literal - (address_of_instruction + 4))/4
```

Deze waarde wordt keer vier vermenigvuldigd (waardoor een <imm10> offset ontstaat) en deze waarde wordt opgeteld bij de waarde in de PC om het adres te bepalen waarvandaan de 32-bits waarde wordt ingelezen. Omdat de offset (uitgedrukt in bytes) beperkt is tot een 10-bits unsigned getal, zijn soms meerdere literal sections nodig in een programma.

Bij de `LDR.N` en `STR.N` assemblertaalinstructies die 32-bits data van of op de stack laden of opslaan geef je een 10-bits unsigned constante op die bij de stackpointer (SP) wordt opgeteld, zie paragraaf 3.11. De SP moet, zoals al eerder vermeldt, altijd uitgelijnd zijn op een adres dat deelbaar is door vier. De constante die bij of van de SP wordt opgeteld moet dus ook deelbaar zijn door vier. De laagste twee bits van deze constante hoeven dus in de machinecode niet opgeslagen te worden. De waarde van `imm8` kan bij deze instructies als volgt berekend worden: `imm8 = <imm10> / 4` waarbij `<imm10> modulo 4` nul moet zijn.

5.8 De B.N instructie

Het adres waarnaar gesprongen wordt, wordt in assembly code met behulp van een label opgegeven. In machinecode moet echter een 11-bits two's complement offset (-1024 t/m +1023) worden opgegeven die aangeeft hoeveel *instructies* er vooruit of achteruit moet worden gesprongen. Zie figuur 5.8. Dit wordt gedaan door deze offset maal twee[30] bij de PC op te tellen. De offset wordt als volgt berekend:

```
imm11_two's_complement = (<label> - (address_of_instruction + 4))/2
```

imm11 instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B.N <label>	1	1	1	0	0	imm11_two's_complement										

Figuur 5.8: De B.N instructie.

5.9 De B<c>.N instructie

Het adres waarnaar gesprongen wordt, als de conditie <c> waar is wordt in assembly code met behulp van een label opgegeven. In machinecode moet echter een 8-bits two's complement offset (-128 t/m +127) worden opgegeven die aangeeft hoeveel *instructies* er vooruit of achteruit moet worden gesprongen. Zie figuur 5.9. Dit wordt gedaan door deze offset maal twee[31] bij de PC op te tellen. De offset wordt als volgt berekend:

```
imm8_two's_complement = (<label> - (address_of_instruction + 4))/2
```

imm8 conditionele instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B<c>.N <label>	1	1	0	1	cond				imm8_two's_complement							

Figuur 5.9: De B<c>.N instructie.

De conditie <c> wordt gecodeerd in vier bits, zie hoofdstuk D.

5.10 Stack instructies met register-list addressing mode

De PUSH.N en POP.N instructies, zie pagina 21, maken gebruik van de register-list addressing mode, zie pagina 14. In de machinecode van deze instructies geven bit 0 tot en met bit 7

[30] Maal twee omdat elke instructie 16 bits = 2 bytes groot is.

[31] Zie voetnoot 30.

aan of respectievelijk de general purpose registers R0 tot en met R7 in de register-list zijn opgenomen. Een één geeft aan dat dit register in de lijst is opgenomen, een nul geeft aan dat dit niet zo is. Zie figuur 5.10. Als bij de machinecode van de **PUSH.N** instructie bit 8 hoog is, dan wordt het LR ook op de stack gezet. De registers worden altijd in een vaste volgorde gepusht: LR, R7, . . . , R0.

Als bij de machinecode van de **POP.N** instructie bit 8 hoog is, dan wordt de PC ook van de stack gehaald. De registers worden altijd in een vaste volgorde gepopt: R0, R1, . . . , R7 en PC.

register_list stack instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUSH.N {<register_list>}	1	0	1	1	0	1	0	LR	register_list							
POP.N {<register_list>}	1	0	1	1	1	1	0	PC	register_list							

Figuur 5.10: Stack instructies met register-list addressing mode.

6

Instructie decoding

Om te begrijpen hoe de processor de machinecodeinstructies decodeert, is het leerzaam om dit ook eens zelf te doen. Werk dat we normaal gesproken aan een disassembler overlaten.

Een volledige lijst met de decodering van alle Pinky instructies, op volgorde van operator, vind je in hoofdstuk C. Voor het decoderen van de `B<c>.N` instructie kun je hoofdstuk D gebruiken.

Bibliografie

[1] *Arm v6-M Architecture – Reference Manual*. Arm, 2010, DDI 0419C. URL: https://documentation-service.arm.com/static/5f8feef5f86e16515cdbf7e4 (geciteerd op p. 6).

[2] *Arm v7-M Architecture – Reference Manual*. Arm, 2021, DDI 0403E.e. URL: https://documentation-service.arm.com/static/606dc36485368c4c2b1bf62f (geciteerd op pp. 6, 7, 8, 12).

[3] *Arm v8-M Architecture – Reference Manual*. Arm, 2022, DDI 0553B.s. URL: https://documentation-service.arm.com/static/62501530caabfd7b3c13e9d2 (geciteerd op p. 7).

[4] *CC3220 SimpleLink Wi-Fi and Internet of Things*. Technical Reference Manual. Texas Instruments, 2017, SWRU465. URL: https://www.ti.com/lit/ug/swru465/swru465.pdf (geciteerd op p. 7).

[5] *CC3220R, CC3220S, and CC3220SF SimpleLink Wi-Fi Single-Chip Wireless MCU Solutions*. Datasheet. Texas Instruments, 2016, SWAS035C. URL: https://www.ti.com/lit/ds/symlink/cc3220s.pdf (geciteerd op p. 7).

[6] *Cortex-M4 Datasheet*. Arm, 2020. URL: https://www.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm%20Cortex-M4%20Processor%20Datasheet.pdf (geciteerd op p. 7).

[7] *Cortex-M4 Devices – Generic User Guide*. Arm, 2011, DUI 0553. URL: https://documentation-service.arm.com/static/5f2ac76d60a93e65927bbdc5 (geciteerd op pp. 5, 8, 9, 10, 12).

[8] *Cortex-M4 Technical Reference Manual*. Arm, 2010, DUI 0439B. URL: https://documentation-service.arm.com/static/5f19da2a20b7cf4bc524d99a (geciteerd op pp. 16, 18).

[9] *Procedure Call Standard for the Arm Architecture*. Arm, 2022, AAPCS32-1. URL: https://github.com/ARM-software/abi-aa/releases (geciteerd op p. 22).

[10] *STM32 Cortex-M4 MCUs and MPUs programming manual*. STMicroelectronics, 2020, PM0214. URL: https://www.st.com/resource/en/programming_manual/pm0214-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf (geciteerd op pp. 8, 12).

[11] *STM32F411xC STM32F411xE Arm Cortex-M4 32b MCU+FPU, 125 DMIPS, 512KB Flash, 128KB RAM, USB OTG FS, 11 TIMs, 1 ADC, 13 comm. interfaces*. Datasheet. STMicroelectronics, 2017, DocID026289. URL: https://www.st.com/resource/en/application_note/dm00354244-stm32-microcontroller-debug-toolbox-stmicroelectronics.pdf (geciteerd op p. 7).

[12] *STM32F411xC/E advanced Arm-based 32-bit MCUs*. Reference Manual. STMicroelectronics, 2018, RM0383. URL: https://www.st.com/resource/en/reference_manual/rm0383-stm32f411xce-advanced-armbased-32bit-mcus-stmicroelectronics.pdf (geciteerd op p. 7).

[13] *Which ARM Cortex Core Is Right for Your Application: A, R or M?* White Paper. URL: https://www.silabs.com/documents/public/white-papers/Which-ARM-Cortex-Core-Is-Right-for-Your-Application.pdf (geciteerd op p. 6).

A

De Pinky instructieset

Instructie	Betekenis	N	Z	C	V	Cycles	Opmerking
ADD.N SP,SP,#<imm9>	SP ← SP + <imm9>	-	-	-	-	1	<imm9> mod 4 must be 0
ADDS.N <Rdn>,#<imm8>	<Rdn> ← <Rdn> + <imm8>	↕	↕	↕	↕	1	
ADDS.N <Rd>,<Rn>,#<imm3>	<Rd> ← <Rn> + <imm3>	↕	↕	↕	↕	1	
ADDS.N <Rd>,<Rn>,<Rm>	<Rd> ← <Rn> + <Rm>	↕	↕	↕	↕	1	
ANDS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> AND <Rm>	↕	↕	-	-	1	
B.N <label>	PC ← <label>	-	-	-	-	1+P	
B<c>.N <label>	if (<c>) PC ← <label>	-	-	-	-	1(+P)	+P if branch is taken
BLX.N <Rm>	LR = addr_of_inst + 3, PC ← Rm	-	-	-	-	1+P	
BX.N LR	PC ← LR - 1, Tbit = 1	-	-	-	-	1+P	
CBNZ.N <Rn>,<label>	if (<Rn> != 0) PC ← label	-	-	-	-	1(+P)	Only forward branch allowed, +P if branch is taken
CBZ.N <Rn>,<label>	if (<Rn> == 0) PC ← label	-	-	-	-	1(+P)	Only forward branch allowed, +P if branch is taken
CMP.N <Rn>,#<imm8>	<Rn> - <imm8>	↕	↕	↕	↕	1	
CMP.N <Rn>,<Rm>	<Rn> - <Rm>	↕	↕	↕	↕	1	
EORS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> EOR <Rm>	↕	↕	-	-	1	
LDR.N <Rt>,=<32-bits literal>	<Rt> ← <32-bits literal>	-	-	-	-	2-Q	Load literal from memory
LDR.N <Rt>, [<Rn>,#<imm7>]	<Rt> ← [<Rn> + <imm7>]	-	-	-	-	2-Q	<imm7> mod 4 must be 0
LDR.N <Rt>,[<Rn>,<Rm>]	<Rt> ← [<Rn> + <Rm>]	-	-	-	-	2-Q	
LDR.N <Rt>,[SP,#<imm10>]	<Rt> ← [SP + <imm10>]	-	-	-	-	2-Q	Load from Stack, <imm10> mod 4 must be 0
LSLS.N <Rd>,<Rm>,#<imm5>	<Rd> ← <Rm> << <imm5>	↕	↕	↕	-	1	Logical Shift Left through Carry
LSRS.N <Rd>,<Rm>,#<imm5>	<Rd> ← <Rm> >> <imm5>	↕	↕	↕	-	1	Logical Shift Right through Carry
MOVS.N <Rd>,#<imm8>	<Rd> ← <imm8>	↕	↕	-	-	1	
MOVS.N <Rd>,<Rm>	<Rd> ← <Rm>	↕	↕	-	-	1	
NOP.N	-	-	-	-	-	1	
ORRS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> OR <Rm>	↕	↕	-	-	1	
POP.N {<register_list>}	Pop from Stack <register_list>	-	-	-	-	1+N(+P)	Pop order: R0, R1, R2, R3, R4, R5, R6, R7, (PC)
PUSH.N {<register_list>}	Push to Stack <register_list>	-	-	-	-	1+N	Push order: LR, R7, R6, R5, R4, R3, R2, R1, R0
STR.N <Rt>, [<Rn>,#<imm7>]	<Rt> → [<Rn> + <imm7>]	-	-	-	-	1	<imm7> mod 4 must be 0
STR.N <Rt>,[<Rn>,<Rm>]	<Rt> → [<Rn> + <Rm>]	-	-	-	-	2-Q	
STR.N <Rt>,[SP,#<imm10>]	<Rt> → [SP + <imm10>]	-	-	-	-	1	<imm10> mod 4 must be 0
SUB.N SP,SP,#<imm9>	SP ← SP - <imm9>	-	-	-	-	1	<imm9> mod 4 must be 0
SUBS.N <Rdn>,#<imm8>	<Rdn> ← <Rdn> - <imm8>	↕	↕	↕	↕	1	
SUBS.N <Rd>,<Rn>,#<imm3>	<Rd> ← <Rn> - <imm3>	↕	↕	↕	↕	1	
SUBS.N <Rd>,<Rn>,<Rm>	<Rd> ← <Rn> - <Rm>	↕	↕	↕	↕	1	

Legenda:

N	Number of registers to pop or push
P	The number of cycles required for a pipeline refill. This ranges from 1 to 3 depending on the alignment and width of the target instruction, and whether the processor manages to speculate the address early.
Q	Q = 1 if the previous instructions is a LDR.N <Rt>, ... and the current LDR or STR instruction does not use <Rt> to calculate the adress, otherwise Q = 0.
↕	flag is affected by instruction
-	flag is not affected by instruction

B

Coderen van Pinky instructies

Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Opmerking
ADD.N SP,SP,#<imm9>	1	0	1	1	0	0	0	0	0	imm7							imm7 = <imm9> / 4, <imm9> module 4 must be 0
ADDS.N <Rdn>,#<imm8>	0	0	1	1	0	Rdn			imm8								
ADDS.N <Rd>,<Rn>,#<imm3>	0	0	0	1	1	1	0	imm3			Rn			Rd			
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd			
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	Rm			Rdn			
B.N <label>	1	1	1	0	0	imm11_two's_complement											imm11_two's_complement = (<label> - (addr_of_instr + 4))/2
B<c>.N <label>	1	1	0	1	cond				imm8_two's_complement								imm8_two's_complement = (<label> - (addr_of_instr + 4))/2
BLX.N <Rm>	0	1	0	0	0	1	1	1	1	Rm				0	0	0	
BX.N LR	0	1	0	0	0	1	1	1	0	1	0	1	1	0	0	0	
CBNZ.N <Rn>,<label>	1	0	1	1	1	0	i	1	imm5					Rn			i:imm5 = (<label> - (addr_of_instr + 4))/2
CBZ.N <Rn>,<label>	1	0	1	1	0	0	i	1	imm5					Rn			i:imm5 = (<label> - (addr_of_instr + 4))/2
CMP.N <Rn>,#<imm8>	0	0	1	0	1	Rdn			imm8								
CMP.N <Rn>,<Rm>	0	1	0	0	0	0	1	0	1	0	Rm			Rn			
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	Rm			Rdn			
LDR.N <Rt>,=<32 bits literal>	0	1	0	0	1	Rt			imm8								imm8 = (addr_of_litaral - (addr_of_instr + 4))/4
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt			imm5 = <imm7> / 4, <imm7> module 4 must be 0
LDR.N <Rt>,[<Rn>,<Rm>]	0	1	0	1	1	0	0	Rm			Rn			Rt			
LDR.N <Rt>,[SP,#<imm8>]	1	0	0	1	1	Rt			imm8								imm8 = <imm10> / 4, <imm10> module 4 must be 0
LSLS.N <Rd>,<Rm>,#<imm5>	0	0	0	0	0	imm5					Rm			Rd			
LSRS.N <Rd>,<Rm>,#<imm5>	0	0	0	0	1	imm5					Rm			Rd			
MOVS.N <Rd>,#<imm8>	0	0	1	0	0	Rdn			imm8								
MOVS.N <Rd>,<Rm>	0	0	0	0	0	0	0	0	0	0	Rm			Rd			
NOP.N	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	Rm			Rdn			
POP.N {<register_list>}	1	0	1	1	1	1	0	P	register_list								
PUSH.N {<register_list>}	1	0	1	1	0	1	0	M	register_list								
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt			imm5 = <imm7> / 4, <imm7> module 4 must be 0
STR.N <Rt>,[<Rn>,<Rm>]	0	1	0	1	0	0	0	Rm			Rn			Rt			
STR.N <Rt>,[SP,#<imm8>]	1	0	0	1	0	Rt			imm8								imm8 = <imm10> / 4, <imm10> module 4 must be 0
SUB.N SP,SP,#<imm9>	1	0	1	1	0	0	0	0	1	imm7							imm7 = <imm9> / 4, <imm9> module 4 must be 0
SUBS.N <Rdn>,#<imm8>	0	0	1	1	1	Rdn			imm8								
SUBS.N <Rd>,<Rn>,#<imm3>	0	0	0	1	1	1	1	imm3			Rn			Rd			
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd			

C

Decoderen van Pinky instructies

Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Opmerking
MOVS.N <Rd>,<Rm>	0	0	0	0	0	0	0	0	0	0	Rm			Rd			
LSLS.N <Rd>,<Rm>,#<imm5>	0	0	0	0	0	imm5					Rm			Rd			
LSRS.N <Rd>,<Rm>,#<imm5>	0	0	0	0	1	imm5					Rm			Rd			
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd			
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd			
ADDS.N <Rd>,<Rn>,#<imm3>	0	0	0	1	1	1	0	imm3			Rn			Rd			
SUBS.N <Rd>,<Rn>,#<imm3>	0	0	0	1	1	1	1	imm3			Rn			Rd			
MOVS.N <Rd>,#<imm8>	0	0	1	0	0	Rdn		imm8									
CMP.N <Rn>,#<imm8>	0	0	1	0	1	Rdn		imm8									
ADDS.N <Rdn>,#<imm8>	0	0	1	1	0	Rdn		imm8									
SUBS.N <Rdn>,#<imm8>	0	0	1	1	1	Rdn		imm8									
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	Rm			Rdn			
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	Rm			Rdn			
CMP.N <Rn>,<Rm>	0	1	0	0	0	0	1	0	1	0	Rm			Rn			
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	Rm			Rdn			
BLX.N <Rm>	0	1	0	0	0	0	1	1	1	1	Rm			0	0	0	
BX.N LR	0	1	0	0	0	1	1	1	0	1	0	1	1	0	0	0	
LDR.N <Rt>,=<32 bits literal>	0	1	0	0	1	Rt		imm8									<32 bits literal> = [(imm8 * 4) + addr_of_instr + 4]
STR.N <Rt>,[<Rn>,<Rm>]	0	1	0	1	0	0	0	Rm			Rn			Rt			
LDR.N <Rt>,[<Rn>,<Rm>]	0	1	0	1	1	0	0	Rm			Rn			Rt			
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt			<imm7> = imm5 * 4
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt			<imm7> = imm5 * 4
LDR.N <Rt>,[SP,#<imm8>]	1	0	0	1	1	Rt		imm8									<imm10> = imm8 * 4
STR.N <Rt>,[SP,#<imm8>]	1	0	0	1	0	Rt		imm8									<imm10> = imm8 * 4
CBZ.N <Rn>,<label>	1	0	1	1	0	0	i	1	imm5					Rn			<label> = (i:imm5 * 2) + addr_of_instr + 4
PUSH.N {<register_list>}	1	0	1	1	0	1	0	M	register_list								
CBNZ.N <Rn>,<label>	1	0	1	1	1	0	i	1	imm5					Rn			<label> = (i:imm5 * 2) + addr_of_instr + 4
POP.N {<register_list>}	1	0	1	1	1	1	0	P	register_list								
ADD.N SP,SP,#<imm9>	1	0	1	1	0	0	0	0	0	imm7							<imm9> = imm7 * 4
SUB.N SP,SP,#<imm9>	1	0	1	1	0	0	0	0	1	imm7							<imm9> = imm7 * 4
NOP.N	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	
B<c>.N <label>	1	1	0	1	cond			imm8_two's_complement									<label> = (imm8_two's_complement * 2) + addr_of_instr + 4
B.N <label>	1	1	1	0	0	imm11_two's_complement											<label> = (imm11_two's_complement * 2) + addr_of_instr + 4

D

B<c>.N condities

cond	<c>	betekenis	conditie	betekenis na CMP Rn,op2
0000	EQ	Equal	Z == 1	Rn == op2
0001	NE	Not Equal	Z == 0	Rn != op2
0010	CS / LO	Carry Set / LOwer	C == 1	Rn < op2, unsigned
0011	CC / HS	Carry Clear / Higher or Same	C == 0	Rn >= op2, unsigned
0100	MI	MInus	N == 1	
0101	PL	Plus	N == 0	
0110	VS	Overflow Set	V == 1	
0111	VC	Overflow Clear	V == 0	
1000	HI	Higher	C == 1 and Z == 0	Rn > op2, unsigned
1001	LS	Lower or Same	C == 0 or Z == 1	Rn <= op2, unsigned
1010	GE	Greater than or Equal	N == V	Rn >= op2, signed
1011	LT	Less Than	N != V	Rn < op2, signed
1100	GT	Greater Than	Z == 0 and N == V	Rn > op2, signed
1101	LE	Less than or Equal	Z == 1 or N != V	Rn <= op2, signed