ARM Cortex-A. Видове инструкции - част 1. Основни инструкции.

Автор: доц. д-р инж. Любомир Богданов

Съдържание

Режими на работа на ARM Cortex-A

Cortex-A имат два режима на работа:
*__привилигерован__ (privileged) – асемблерните инструкции могат да достъпват всички регистри на ядрото. Хендлерите на прекъсванията се изпълняват винаги в привилегирован режим.

*__непривилигерован__ (unprivileged) - асемблерните инструкции могат да достъпват само част от регистрите на ядрото. Основният код на програмата обикновено работи в непривилегирован режим.

Режими на работа на ARM Cortex-A

За процесори с TrustZone инструкции има още два вида:
***защитен** (secure)
***незащитен** (non-secure)

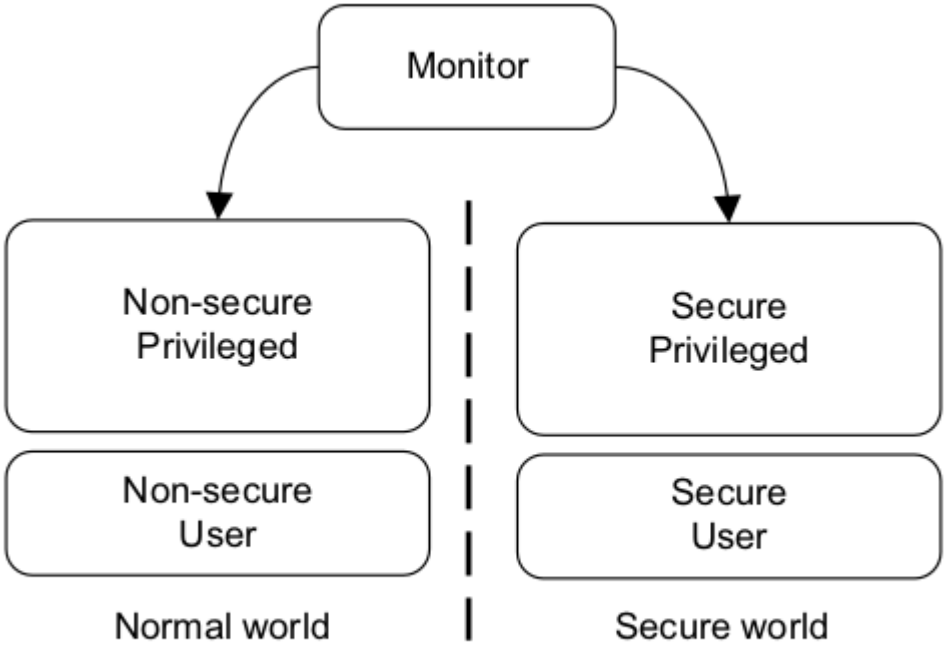


Figure 3-1 Secure and Non-secure worlds

Режими на работа на ARM Cortex-A

Зад понятието “secure world” стои **един бит** (наречен NS, от регистър SCR – Secure Configuration Register), който се добавя при всички достъпи до адресното поле и който **разделя това поле на две 32-битови полета с абсолютни адреси**, наречени: “Secure world” и “Normal world”.

Изпълнението на код от двете полета става чрез **мултиплексиране във времето** (time-slicing).

Преминаването от едното поле в другото става чрез **хендлер на прекъсване наречен Monitor**.

Режими на работа на ARM Cortex-A

За да се генерира прекъсване Monitor, трябва незащитен код да изпълни инструкцията:

*SMC – secure monitor call

или да се предизвикат прекъсвания (хардуерни изключения) IRQ, FIQ или External abort.

Асемблер на GNU и ARM

Използването на Асемблер за микропроцесори е оправдано, когато [1]:

* трябва да се напише високо-оптимизиран код;

* трябва да се разработи компилатор;

* трябва да се използват възможности на микропроцесора, които не са достъпни в езика C.

Асемблер на GNU и ARM

Примерни приложения [1]:

* в код за начално зареждане (boot code);

* в драйвери за модули в системата;

* при разработване на операционна система;

* по време на дебъгване – за да се направи връзката между един ред на код на C и съответстващите му инструкции.

Асемблер на GNU и ARM

ARM са **RISC** микропроцесори, което означава [1]:

* имат малък брой прости инструкции;
* всяка инструкция се изпълнява за един или няколко такта;
* има прости режими на адресация, където адресите идват от регистър на ядрото или от битови полета в самата инструкция (т.е. са внедрени в кода на инструкцията).

Асемблер на GNU и ARM

Основните класове инструкции на ARM могат да бъдат разделени на [1]:

* инструкции за **обработка на данни**;
* инструкции за **достъп до паметта** (load/store);
* инструкции за **преход;**
* инструкции **разни** (вж следващата лекция).

Забележка: ARM **не могат** да извършват операции **директно с регистри от паметта** [1].

Асемблер на GNU и ARM

Инструкциите на ARM са с **два** или **три операнда** и обикновено първият е приемника на резултата:

* инструкция операнд 1 [out], операнд 2 [in]

*инструкция операнд 1 [out], операнд 2 [in], операнд 3 [in]

Асемблер на GNU и ARM

Пример с различни архитектури (събиране на регистър от ядрото с числото 100):

x86: add eax, #100

68k: add #100, d0

msp430: add #100, r4

arm: add r0, r0, 100

Асемблер на GNU и ARM

Пример с различни архитектури (зареждане на регистър от ядрото със стойност от паметта на адрес, сочен от друг регистър на ядрото):

x86: mov eax, DWORD PTR [ebx]

68k: mov.l (a0), d0

msp430: add @r4, r5

arm: ldr r0, [r1]

Асемблер на GNU и ARM

Въпреки, че ARMv7 архитектурата е 32-битова, съществуват части от процесорите, които са по-големи (64 и повече бита) в [1]:

* модули за извличане на инструкции (IF)
* модули за достъп до паметта и I/O (MEM)

Асемблер на GNU и ARM

GNU асемблер (as) – програма, която конвертира сорс код на Асемблер в двоичен обектов файл [1], [a]. Този файл е входен за линкера. Асемблерът е част от GNU toolchain и се разработва от open-source общността. Безплатен.

Пример:

```
arm-none-eabi-as -g filename.s -o filename.o
arm-none-eabi-ld filename.o -o filename.elf
```

-g аргументът включва дебъгерна информация в изходния файл (отбелязан с -o аргумента)

Асемблер на GNU и ARM

Всеки файл на GNU асемблера има три основни полета:

етикет: инструкция @коментар
директива @коментар

Етикет :
*низ, идентифициращ адреса на настоящата инструкция.
*може да бъде използван за преходи;
*задължително започва с буква, но може да бъде последван от цифри;

Асемблер на GNU и ARM

Инструкция:
* инструкция от ARM Асемблера или
*асемблерна директива – команди, указващи операции, които трябва да бъдат извършени от асемблера, а не от микропроцесора (напр. поместване на кода на точно-определени адреси, подравняване на кода, деклариране на константи)

Коментар:
всичко след @ или / … */ ще бъде игнорирано и няма да бъде асемблирано

Асемблер на GNU и ARM

Асемблерни директиви – започват с точка. Най-често използваните са:

.align – изкуствено се добавят байтове със стойност 0x00 в сегмент за данни (променливите на програмата) или с NOP в сегмент за инструкции, така че следващият адрес да е кратен на една дума. Тази директива може да се използва с параметър:

.align 4

=> кодът ще е подравнен на 2^4 думи

Асемблер на GNU и ARM

.ascii “символен низ” – въведи символен низ в обектовия файл, без терминираща нула накрая.

.asciiz “символен низ” – въведи символен низ в обектовия файл, с терминираща нула накрая.

.byte [число] – въведи 1 байт в обектовия файл.
.hword [число] – въведи 2 байта в об. файл.
.word [число] – въведи 4 байта в об. файл.

Може да се използва запетая за множество:
.byte 0xff, 0x7f, 0x33, 0x55

и т.н.

Асемблер на GNU и ARM

.data – след тази директива всички асемблерни инструкции/променливи ще бъдат поместени в сегмента “data” (най-често SRAM).

.text - след тази директива всички асемблерни инструкции/константи ще бъдат поместени в сегмента “text” (най-често Flash).

.end – директива за край на асемблерния файл.

.equ var, value – задава “value” на символа “var”. Аналогично на .set var, value, както и на var=value.

Символ – функция или глобална променлива.

Асемблер на GNU и ARM

.extern “var” – символът “var” е деклариран в друг файл.

.global “var” – указва на асемблера да направи символът “var” видим в други файлове.

.include “file” – директива, която внедрява съдържанието на файл “file” на настоящия ред от асемблерния файл. Използва се най-често за включване на хедърни файлове.

Асемблер на GNU и ARM

Операндите на инструкциите могат да бъдат:

*__числа__ десетични (без префикс), шестнадесетични (с префикс 0x), двоични (с префикс 0b)

*__низове__ – използват се единични кавички

Забележка: стандартни математически и логически операции може да бъдат преработени от асемблера **до константи**, за да не се използват инструкции.

Асемблер на GNU и ARM

Регистрите на ядрото може да се включат като операнд по следния начин:
*R0 – R15 регистри с общо предназначение
*SP (или още R13) – стеков указател
*FP (или още R11) – фреймов указател
*LR (или още R14) – регистър за съхранение на адрес на връщане
*PC (или още R15) – програмен брояч

Битовете от статус регистъра си имат псевдоними:
* xPSR, xPSR_all, xPSR_f, xPSR_x, xPSR_ctl, xPSR_fs, xPSR_fx, xPSR_f, xPSR_cs, xPSR_cf, xPSR_cx (x може да е C – current, S - saved)

Асемблер на GNU и ARM

ARM асемблер (armasm) - програма, която конвертира сорс код на Асемблер в двоичен обектов файл. Този файл е входен за линкера. Асемблерът е част от ARM toolchain v5 и се разработва от компанията ARM Ltd. Платен.

ВНИМАНИЕ! От ARM toolchain v6 и нагоре, използването на armasm (и съответния компилатор armcc) не се препоръчва. **За компилатор и асемблер** се препоръчва armclang (clang и LLVM базиран). Асемблерът на armclang използва синтаксиса на GNU асемблера.

Асемблер на GNU и ARM

Arm Compiler 5	Arm Compiler for Embedded 6	Function
`armcc`	`armclang`	Compiles C and C++ language source files, including inline assembly.
`armcc`	`armclang`	Preprocessor.
`armasm`	`armasm`	Legacy assembler for assembly language source files written in `armasm` syntax. Use the `armclang` integrated assembler for all new assembly files.
Not available	`armclang`. This is also called the `armclang` integrated assembler.	Assembles assembly language source files written in GNU assembly syntax.
`fromelf`	`fromelf`	Converts Arm ELF images to binary formats and can also generate textual information about the input image, such as its disassembly and its code and data size.
`armlink`	`armlink`	Combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.
`armar`	`armar`	Enables sets of ELF object files to be collected together and maintained in archives or libraries.

Таблицата е показана в [2].

Асемблер на GNU и ARM

В armclang асемблерът се нарича “**armclang Integrated Assembler**” – интегриран асемблер на армкланг (ИАНА)

Асемблер на GNU и ARM

ИАНА съдържа:
*етикети
*инструкции
*директиви
*макроси
* коментари - //, /* … */, @

ИАНА директиви:
.byte, .hword, .word
.quad число – 8 байта
.octa число – 16 байта

.ascii, .asciz
.string “символен низ” – еквивалентна на .asciz

Асемблер на GNU и ARM

.float [число с пл. зап.] - заделя 4 байта в обектовия файл и съхранява числото в IEEE754 single precision формат.

.double [число с пл. Зап.] - заделя 8 байта в обектовия файл и съхранява числото в IEEE754 double precision формат.

Пример:

```
float_pi:
    .float 3.1415

double_pi:
    .double 3.1415926
```

Асемблер на GNU и ARM

.pushsection [име] – всички инструкции/данни след тази директива ще станат част от нов сегмент на паметта с име “име”.

.popsection - всички инструкции/данни след тази директива ще бъдат част от сегмент на паметта с име, което е било назначено преди извикването на последната .pushsection директива.

.text, .data, .rodata, .bss

Асемблер на GNU и ARM

.global
.local [var] – символът “var” ще бъде видим само в настоящия файл, което ще позволи използването на едно и също име в различни файлове.
.weak [var] – символът “var” ще бъде видим във всички файлове, като при .global, но с три съществени разлики:
*ако никъде не е деклариран/имплементиран символа, това няма да предизвика грешка
*ако има декларация/имплементация на повече от един weak символ с едно и също име, взима се последния срещнат

Асемблер на GNU и ARM

*ако има не-weak декларация/имплементация със същото име, асемблерът ще вземе нея.

Пример:

```
//Файл №1
.weak pi:
    .float 0.00

//Файл №2
pi:
    .float 3.1415       //ИАНА ще вземе pi=3.1415
```

Асемблер на GNU и ARM

ИАНА и GNU поддържат формат на инструкциите наречен **UAL (Unified Assembly Language)** – това е универсален формат, чийто синтаксис е еднакъв за програми с ARM и Thumb набор от инструкции. Тоест, издига се нивото на абстракция на езика Асемблер.

*Етикет – не е нужно да има двуеточие

```
Loop    MUL R5, R5, R1
        SUBS R1, R1, #1
        BNE Loop
```

Асемблер на GNU и ARM

* Инструкция – използват се псевдо-инструкции (емулирани инструкции). Това са несъществуващи инструкции, които на по-късен етап се заменят с една или няколко истински такива.

*Директиви – има различия спрямо оригиналните (GNU и ИАНА) асемблери – **не** започват с точка.

DCB число – 1 байт
DCW число – 2 байта
DCD число – 4 байта
адрес **EQU** число – задаване на име на адрес или число

Асемблер на GNU и ARM

Пример:
MESSAGE **DCB** “Hello World!”,0
→ създава нулево-терминиран символен низ.

Masks **DCD** 0x100, 0x80, 0x40, 0x20, 0x10
→ създава масив от 32-битови числа

FLASH_START **EQU** 0x08000000
→ задава име на адрес 0x08000000 (като #define в C)

Асемблер на GNU и ARM

ALIGN 4 → подравни на 2^4 байта, еквивалентна на .align
AREA → еквивалентна на .section
END → еквивалентна на .end

и т.н.

Асемблер на GNU и ARM

Как да разпознаем 2-та типа ARM Асемблер?

***стари версии** на ARM асемблера съдържат инструкции с 2/3/4 операнда. Възможно е да се срещне условно изпълнение на инструкции, които не са за преход – например ADDNE R0, R0, #1. Файловете се означават .s или .S.

***UAL-съвместими версии** – съдържат директивата **.syntax unified** . Пред константите може да не се пише #. Условните инструкции задължително трябва да са предшествани от IT мнемоника. С директиви .arm и .thumb се отбелязва код с 32-битови и 16-/32-битови инструкции съответно.

Асемблер на GNU и ARM

Асемблер само с 16-битови Thumb инструкции – познава се по **липсата на .syntax unified**, но съдържа други директиви - .code 16, .thumb или .thumb_func .

Асемблер на GNU и ARM

Как инструкция на ARM която е 32-битова ще работи с 32-битова константа, при положение, че са нужни битови полета за КОП (код на операцията), както и служебни битове в инструкцията?

Има три възможности за решаване на проблема:

*използват се **12 бита за константа**. На теория това ограничава числата, която тя може да представя в обхвата от -2048 до +2047. Но всъщност се използва:
→ 8-битова константа и
→ 4-битово число, указващо ротиране надясно.

Асемблер на GNU и ARM

→ Ротирането се извършва в рамките 0 – 30 (2^4.2) със стъпка 2 – 0, 2, 4, 6, 8 и т.н.
→ Това означава, че само някои константи могат да бъдат кодирани в инструкцията, а другите са невъзможни комбинации.

Пример: 0x23 и 0xff са възможни. Също и 0x2300.0000 чрез 0x23 ROR 8 (ROtate Right Register).

Константата 0x3ff не може да бъде представена в описания формат.

Асемблер на GNU и ARM

Обобщение – с настоящия формат може да се кодират константи по следния начин:
- 8 бита за константа + 4 за ротиране
- константа под формата 0x00XY00XY
- константа под формата 0xXY00XY00
- константа под формата 0xXYXYXYXY

където XY е число в обхвата 0x00 - 0xff

Асемблер на GNU и ARM

*чрез използване на **специализирани инструкции:**

→ movw (move wide) – копиране на 16 бита, кодирани в самата инструкция, в младшите 16 бита на регистър от ядрото. Старшите 16 бита се зануляват.

→ movt (move top) – копиране на 16 бита, кодирани в самата инструкция, в старшите 16 бита на регистър от ядрото. Младшите 16 бита на регистъра не се променят.

Асемблер на GNU и ARM

UAL асемблера има специален синтаксис за улеснение при работа с тези инструкции:

```
my_value EQU  0x11223344

movw   R0, #:lower16:my_value
movt    R0, #:upper16:my_value
```

Недостатък: инструкциите са 2.
Предимство: константите са кодирани в инструкцията и не е нужен достъп до данновата памет

Асемблер на GNU и ARM

*чрез **използване на PC-относителна адресация** – с LDR инструкция се копира 32-битова константа от *регион за константи*† (literal pool) в регистър от ядрото.

Пример:

```
    ldr r0, =my_value (=my_value е еквивалент на [PC, #offset])
…
…
…
my_value:
    DCD 0x3ff00000
```

†*Регион за константи* (literal pool) – област в паметта, съдържаща константи в края на подпрограмата и преди началото на следващата такава [1].

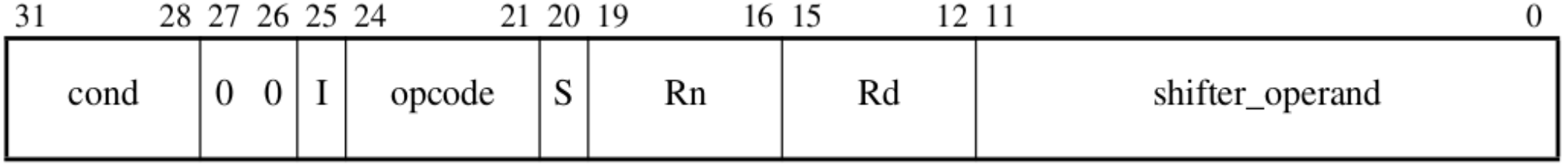
Инструкции за обработка на данни

Най-важните инструкции за обработка на данни са:
*аритметични
ADC, ADD, MOV, MVN, RSB, RSC, SBC, SUB

*логически
AND, BIC, EOR, ORR

*за сравнение и тест
CMP, CMN, TEQ, TST

Инструкции за обработка на данни



Формат на 32-битова ARM инструкция за обработка [3], **A32**

I – указва дали полето shifter_operand съдържа константа или регистър

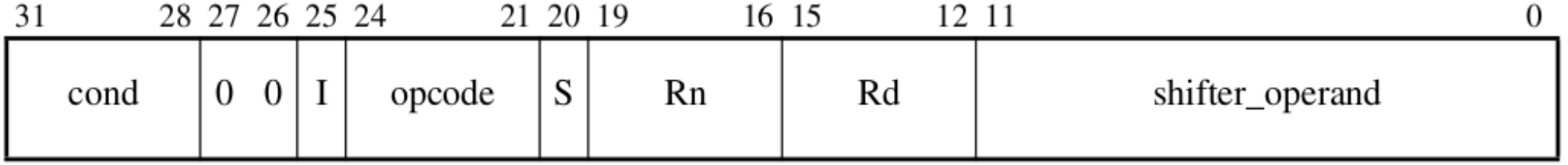
S – указва дали да се актуализира STATUS регистъра на ядрото

Rn – указва регистър от ядрото, който ще е операнд-източник

Rd – указва регистър от ядрото, който ще е операнд-приемник на резултат

shifter_operand – указва втори операнд-източник.

Инструкции за обработка на данни



Формат на 32-битова ARM инструкция за обработка [3], **A32**

cond – условие за изпълнение на инструкцията - тества се някой от битовете N, Z, C, V от STATUS (APSR) регистъра. Ако той е активиран, инструкцията се изпълнява. Ако не – инструкцията се заменя с NOP и се изпълнява следващата инструкция.

Ако cond = AL (always), инструкцията е безусловна и винаги се изпълнява.

Инструкции за обработка на данни

Table A3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See *Condition code 0b1111*	-

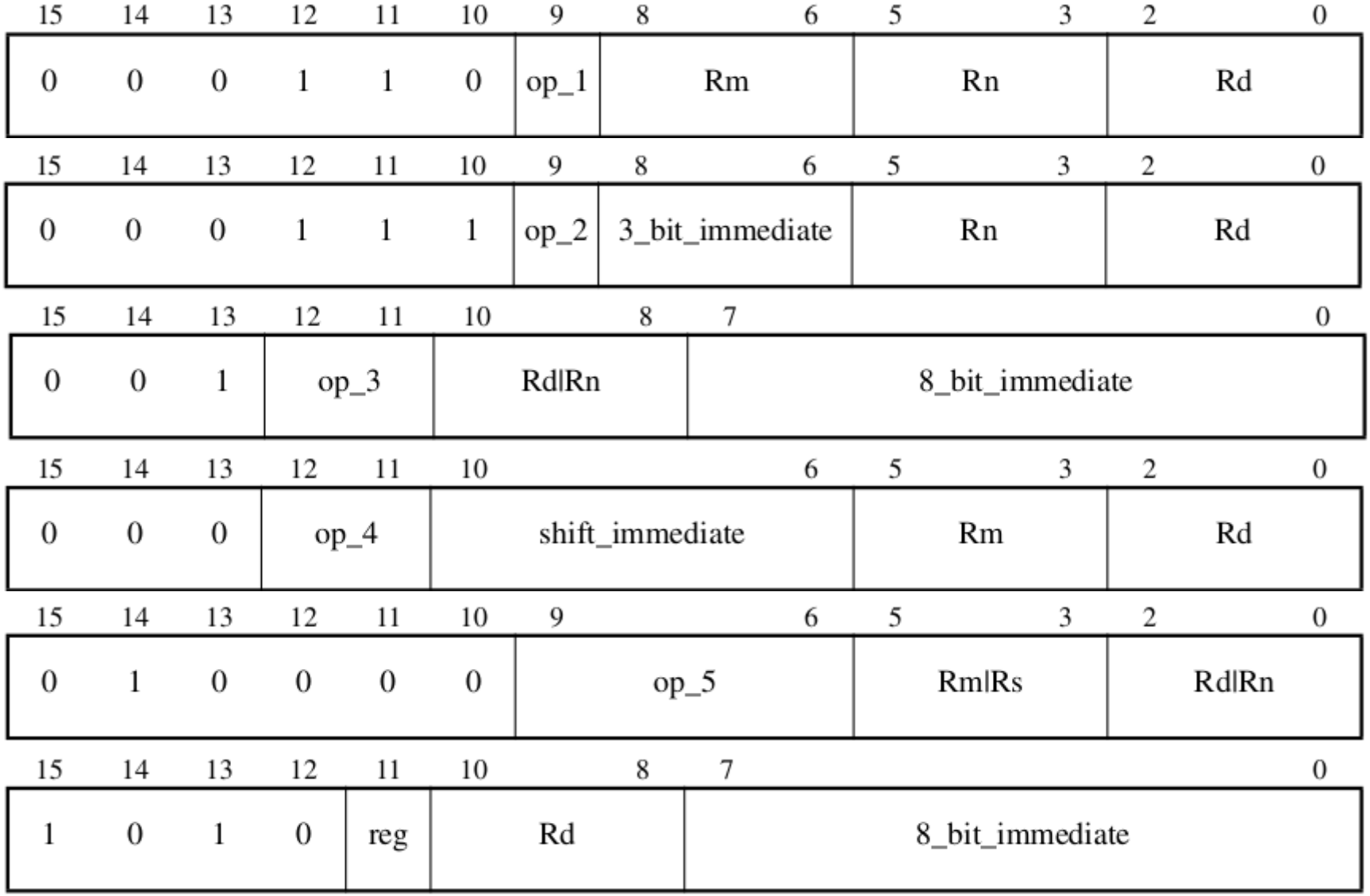
Инструкции за обработка на данни

Пример:

```
subs    r0, r1, r2  //r0 = r1 - r2
addeq   r0, r0, r3 //Ако r0 == 0, изпълни r0 = r0 + r3
                   //Ако r0 != 0, изпълни NOP
orr     r0, #0x40
```

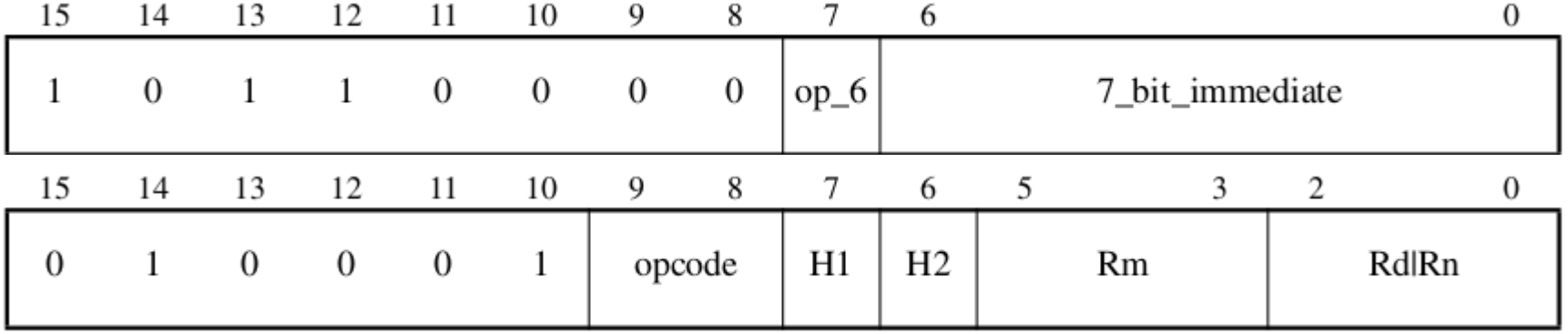
В този пример addeq тества дали бит Z е равен на 1 в резултат от изпълнение на предишната инструкция subs. Това отговаря на условието EQ от таблицата на миналия слайд.

Инструкции за обработка на данни



Формат на 16-битова Thumb инструкция за обработка [3], **T32**

Инструкции за обработка на данни



Формат на 16-битова Thumb инструкция за обработка [3], **T32**

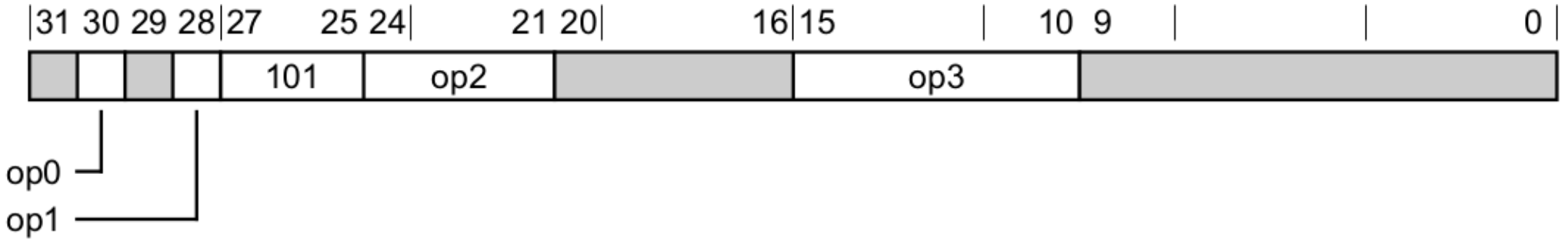
Rm – заедно с **H2** формират 4-битово число, което указва номер на регистър от ядрото, който ще е първи операнд-източник

Rn – указва регистър от ядрото, който ще е втори операнд-източник

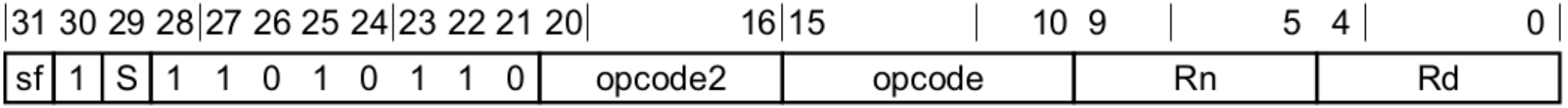
Rd – заедно с **H1** формират 4-битово число, което указва номер на регистър от ядрото, който ще е операнд-приемник на резултат. **H1** е най-старшият бит.

op_1, op_2, op_3, op_4, op_5, op_6 – КОП (== opcode)

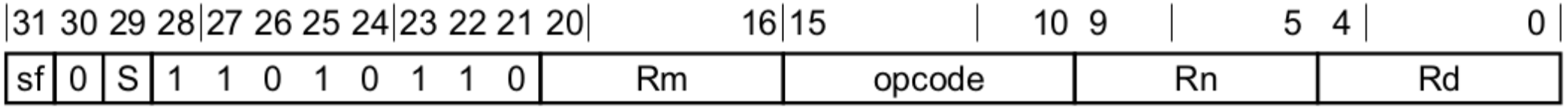
Инструкции за обработка на данни



Общ формат на 32-битова ARM инструкция за обработка [4], **A64**



Формат на 32-битова ARM инструкция за обработка [4], с **1 източник-операнд, A64**



Формат на 32-битова ARM инструкция за обработка [4] с **2 източника-операнди**, **A64**

Инструкции за обработка на данни

Table C4-90 Encoding table for the Data Processing -- Register group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	1	0110	-	Data-processing (2 source)
1	1	0110	-	Data-processing (1 source)
-	0	0xxx	-	Logical (shifted register)
-	0	1xx0	-	Add/subtract (shifted register)
-	0	1xx1	-	Add/subtract (extended register)
-	1	0000	000000	Add/subtract (with carry)
-	1	0000	x00001	Rotate right into flags
-	1	0000	xx0010	Evaluate into flags
-	1	0010	xxxx0x	Conditional compare (register)
-	1	0010	xxxx1x	Conditional compare (immediate)
-	1	0100	-	Conditional select
-	1	1xxx	-	Data-processing (3 source)

Инструкции за достъп до паметта

Само две инструкции могат да копират данни от/в регистри на ядрото в/от паметта:
***ldr** ***str**

Добавянето на b, h или d към инструкцията указва вида на копирането, а s – дали има знак:
***ldrb** – копирай байт от паметта в регистър от ядрото
***ldrh** – копирай полудума (16-бита)
***ldr** – копирай дума (32-бита)
***ldrd** – копирай двойна дума (64-бита)

***ldrsb** – копирай байт със знак
***ldrsh** – копирай полудума със знак

Инструкции за достъп до паметта

***strb** – копирай байт от регистър на ядрото в паметта
***strh** – копирай полудума (16-бита)
***str** – копирай дума (32-бита)
***strd** – копирай двойна дума (64-бита)

Инструкции за достъп до паметта

*4 вида адресации [1]:
→ регистрова (**register**)
→ пред-индексна (**pre-indexed**)
→ пред-индексна с обратен запис (**pre-indexed with write-back**)
→ след-индексна с обратен запис (**post-indexed with write-back**)

Инструкции за достъп до паметта

Регистрова (**register**) адресация: регистър от ядрото се слага в скоби и стойноста на регистъра е указател към адрес, от/в който ще се копира стойността.

ldr r0, [r1] //копирай в r0 стойността от адреса, сочен от r1

str r0, [r1] //копирай стойността от r0 на адреса, сочен от r1

Инструкции за достъп до паметта

Пред-индексна (**pre-indexed**) адресация: регистър от ядрото се слага в скоби **[]** и стойноста на регистъра е указател към адрес, към който се добавя отместване, от/в който ще се копира стойността..

ldr r0, [r1, r2] //копирай в r0 стойността от адреса, сочен от r1+r2
ldr r0, [r1, #32] //копирай в r0 стойността от адреса, сочен от r1+32
ldr r0, [r1, r2, LSL#2] //копирай в r0 стойността от адреса, сочен от r1+(r2<<2)
str r0, [r1, r2] //копирай стойността от r0 на адреса, сочен от r1+r2
str r0, [r1, #32] //копирай стойността от r0 на адреса, сочен от r1+32
str r0, [r1, r2, LSL#2] //копирай стойността от r0 на адреса, сочен от r1+(r2<<2)

Инструкции за достъп до паметта

Пред-индексна с обратен запис (pre-indexed with write-back) адресация: регистър от ядрото се слага в скоби [] с **!** и стойноста на регистъра е указател към адрес, към който се добавя отместване, от/в който ще се копира стойността. След изпълнението на инструкцията, адресният регистър се увеличава със стойността на отместването.

ldr r0, [r1, r2]! //копирай в r0 стойността от адреса, сочен от r1+r2, след това r1 = r1 + r2

ldr r0, [r1, #32]! //копирай в r0 стойността от адреса, сочен от r1+32, след това r1 = r1 + 32

str r0, [r1, r2]! //копирай стойността от r0 на адреса, сочен от r1+r2, след това r1 = r1 + r2

str r0, [r1, #32]! //копирай стойността от r0 на адреса, сочен от r1+32, след това r1 = r1 + 32

Инструкции за достъп до паметта

След-индексна с обратен запис (post-indexed with write-back) адресация: регистър от ядрото се слага в скоби [] и след тях отместване. Стойноста на регистъра е указател към адрес, от/в който ще се копира стойността. След изпълнението на инструкцията, адресният регистър се увеличава със стойността на отместването.

ldr r0, [r1], r2 //копирай в r0 стойността от адреса, сочен от r1, след това r1 = r1 + r2

ldr r0, [r1], #32 //копирай в r0 стойността от адреса, сочен от r1, след това r1 = r1 + 32

str r0, [r1], r2 //копирай стойността от r0 на адреса, сочен от r1, след това r1 = r1 + r2

str r0, [r1], #32 //копирай стойността от r0 на адреса, сочен от r1, след това r1 = r1 + 32

Инструкции за достъп до паметта

Офсетите с константа може да използват **отрицателни** числа:

ldr r0, [r1, #-32]

str r0, [r1], #-8

ldr r0, [r1, #-4]!

Инструкции за преход

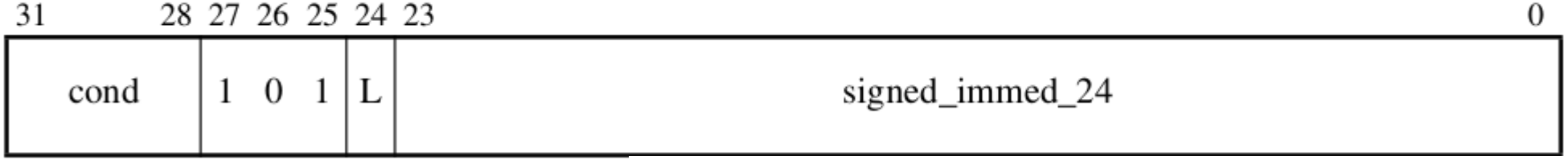
B (branch) – безусловен/условен преход към относителен адрес (относителен спрямо настоящия адрес на инструкцията B).

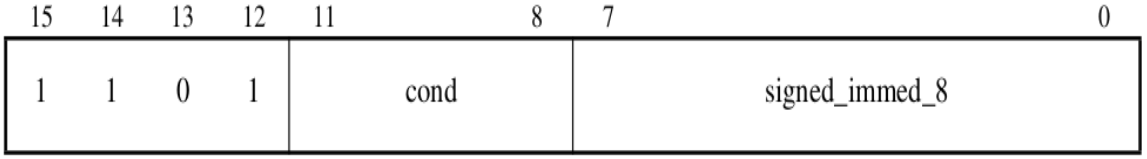
BL (branch with link) – безусловен/условен преход към относителен адрес със записване адреса на връщане в Link регистъра (R14). Използва се за извикване на подпрограми.

BX (branch with exchange) – безусловен преход към относителен адрес със смяна на набора от инструкции (от ARM към Thumb или от Thumb към ARM).

Инструкции за преход

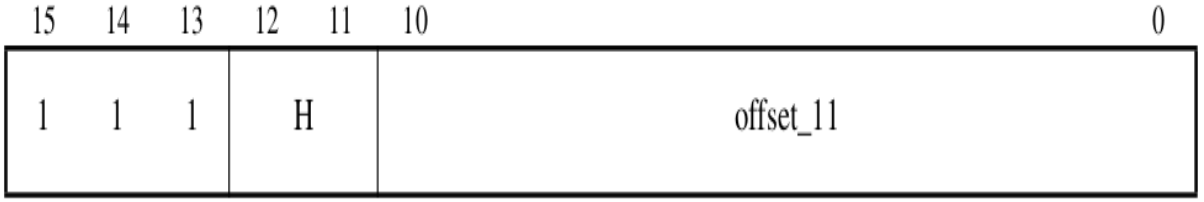
B, BL





B (1) (Branch) provides a conditional branch to a target address.

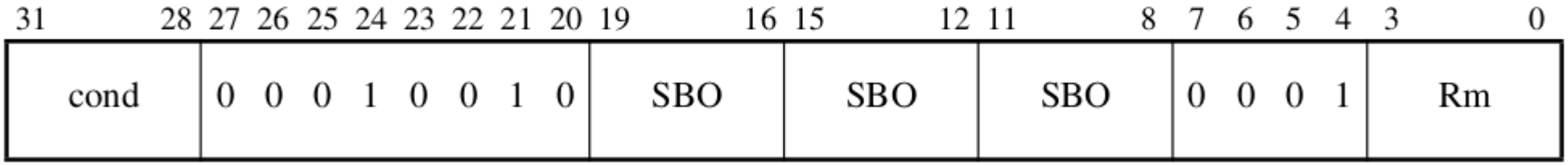
BL, BLX (1)



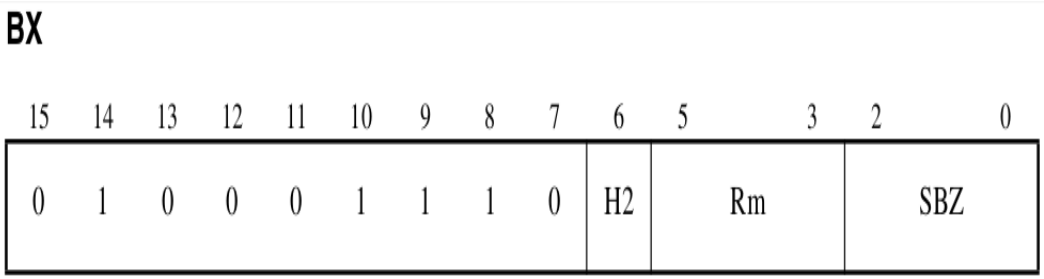
cond – условие, което да се тества (от статус флаговете)
L = 1 → инструкцията = BL
L = 0 → инструкцията = B
H – (част от) код на операцията

Инструкции за преход

BX



BX



SBO – should be one
SBZ – should be zero

Инструкции за преход

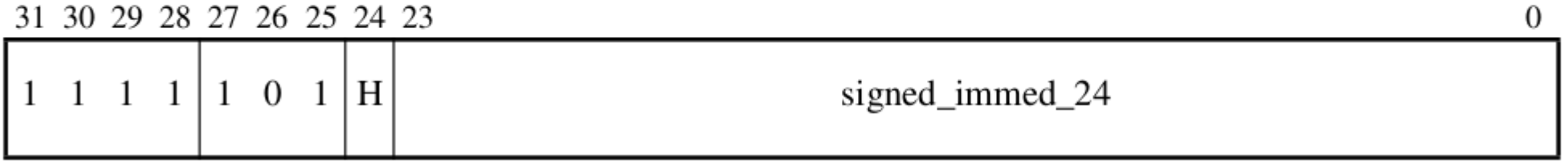
BLX (branch with link and exchange) – безусловен преход към относителен адрес със смяна на набора от инструкции (от ARM към Thumb или от Thumb към ARM) и запаметяване адреса на връщане в Link регистъра R14.

ADD, SUB с операнд PC – аритметика с приемен регистър PC. Остарял метод, не се препоръчва. Не се поддържа в Thumb режим.

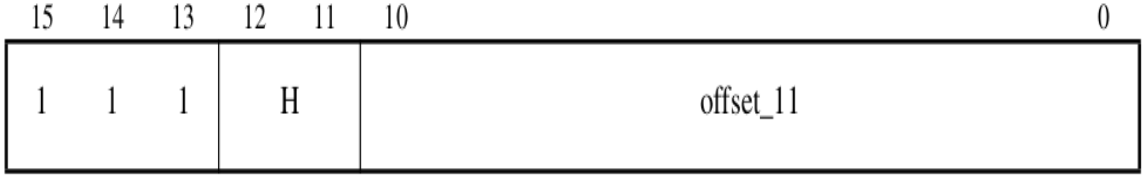
LDR, LDM и **POP** с операнд PC – директно зареждане на PC със стойност на абсолютен адрес.

Инструкции за преход

BLX (1)



BL, BLX (1)



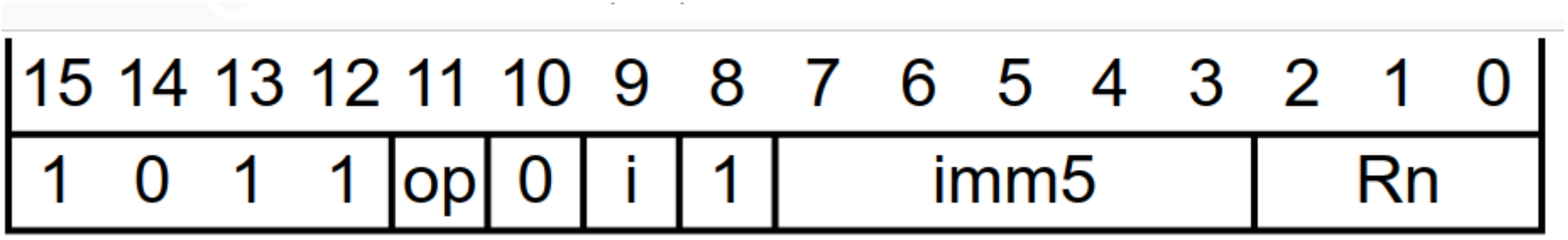
Инструкции за преход

CBZ (compare and branch on zero) – провери дали операндът източник е **равен на 0**. Ако да, иди на относителен адрес. Ако не, продължи напред. Тази инструкция **не засяга CPSR флаговете**.

CBNZ (compare and branch on not zero) – провери дали операндът източник е **различен от 0**. Ако да, иди на относителен адрес. Ако не, продължи напред. Тази инструкция **не засяга CPSR флаговете**.

Забележка: обхватът на преходите за тези инструкции е 4 – 130 байта.

Инструкции за преход



Няма 32-битов вариант на тези инструкции.
op = 1 → CBNZ [7]
op = 0 → CBZ
i – указва знак на 5-битовата константа imm5 [6]

Инструкции за преход

TBB (Table Branch Byte) – прочита байт от таблица с отмествания и извършва PC-относителен преход напред в програмата с удвоената стойност на байта (т.е. преходите са на четни адреси).

TBH (Table Branch Half-word) – аналогична на TBB.

Базовият адрес на таблицата и индексът на байта от таблицата се записват в два отделни регистъра.

Използват се за имплементация на switch( ) конструкции в C.

Инструкции за преход

T1 TBB [<Rn>,<Rm>] Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	Rm			

T1 TBH [<Rn>,<Rm>,LSL #1] Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	Rm			

Пример:

TBB [Rn, Rm]

TBB [Rn, Rm, LSL #2]

- Rn съдържа адрес на таблицата с отмествания;
- Rm съдържа индекс от таблицата с отмествания;
- LSL #1 добавя логическо преместване наляво.

```
    ADR.W  R0, BranchTable_Byte
    TBB    [R0, R1]          ; R1 is the index, R0 is the base address of the
                             ; branch table
Case1
; an instruction sequence follows
Case2
; an instruction sequence follows
Case3
; an instruction sequence follows
BranchTable_Byte
    DCB    0                 ; Case1 offset calculation
    DCB    ((Case2-Case1)/2) ; Case2 offset calculation
    DCB    ((Case3-Case1)/2) ; Case3 offset calculation
```

Пример[5]:

```
    TBH    [PC, R1, LSL #1]  ; R1 is the index, PC is used as base of the
                             ; branch table
BranchTable_H
    DCI    ((CaseA - BranchTable_H)/2)  ; CaseA offset calculation
    DCI    ((CaseB - BranchTable_H)/2)  ; CaseB offset calculation
    DCI    ((CaseC - BranchTable_H)/2)  ; CaseC offset calculation

CaseA
; an instruction sequence follows
CaseB
; an instruction sequence follows
CaseC
; an instruction sequence follows
```

Литература

[1] “ARM Cortex-A Series Programmer’s Guide”, Version 4.0, ARM DEN0013D (ID012214), 2013.
[2] “ARM Compiler for Embedded v6.21” , Migration and Compatibility Guide, Issue 00 100068_6.21_00_en, 2023.
[3] “ARM Architecture Reference Manual”, ARM DDI 0100l, ARM Ltd, 2005.
[4] “ARM Architecture Reference Manual for A-profile architecture”, ARM DDI 0487J.a (ID042523), 2023.
[5] “Cortex-M3 Devices Generic User Guide” – ARM DUI 0552A (ID121610), 2010.
[6] “ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition”, ARM DDI 0406C ID112311, 2011.
[7] “ARM Architecture Reference Manual Thumb-2 Supplement”, ARM DDI 0308D, 2005.

Външни връзки

[a] http://sourceware.org/binutils/docs/as/index.html