ARM Cortex-A. Модул за векторна обработка на данни NEON в ARM Cortex-A.
Потребителски стандарт за извикване на функции (ABI)

Автор: доц. д-р инж. Любомир Богданов

Съдържание

Въведение в ARM NEON

ARM NEON* е модул на Cortex-A процесорите, който комбинира функциите на FPU с векторни SIMD инструкции за целите на обработка на данни в медиини приложения [1]:
*аудио, видео обработка;
*обработка на изображения;
*2D графика, базирана на правоъгълни блокове от пиксели;
*3D графика;
*конвертиране на цветово пространство (color space);
*симулации на физични процеси.

*a.k.a. **MPE** (Media Processing Engine) [2]

Въведение в ARM NEON

ARM NEON програмен модел:

*32 single precision (32-битови) регистъра
*32 double precision (64-битови) или 16 quad precision (128-битови) регистъра

Използва вектори – масиви от регистри.

Може да работи с числа със закръгляване (fused data types).

Регистрите на NEON са различни от регистрите на Cortex-A целочисленото ядро.

Въведение в ARM NEON

ARM NEON се състои от следните модули:

*регистров файл
*конвейер за изпълнение на целочислени операции
*конвейер за изпълнение на операции със single-precision числа с плаваща запетая
*load/store модул с възможност за преминаване от NEON в Cortex-A регистри

Въведение в ARM NEON

SISD (Single Instruction Multiple Data) – инструкции, при които една операция се прилага на само една стойност от един регистър-източник.

За обработка на няколко стойности, трябва да се използват няколко инструкции:

```
add r0, r5
add r1, r6
add r2, r7
add r3, r8
```

Въведение в ARM NEON

Обработка на **аудио/видео** потоци от данни би била **бавна**.

Затова тези информационни потоци се пренасочват към Graphics Processing Unit (GPU) или Media Processing Engine (MPE) модули.

С тяхна помощ, една инструкция може да обработва повече от една стойност.

Обработка на 8- и 16-битови стойности на ARM микропроцесор не товари хардуера на максимум, понеже той е проектиран за 32-битови операции.

Въведение в ARM NEON

SIMD VFP mode (Single Instruction Multiple Data, Vectored Floating Point mode) – инструкции, при които една операция се прилага на повече от една стойност от няколко регистъра-източници.

Ако от контролният регистър на процесора, битовото поле LEN е равно на 4, то следната инструкция:

vadd.f32 s24, s8, s16

Ще извърши 4 floating point операции:

s24 = s8 + s16

s25 = s9 + s17

s26 = s10 + s18

s27 = s11 + s20

Въведение в ARM NEON

Въпреки, че инструкцията от предишния пример е една, събирането на 4 числа ще стане на 4 **последователни (във времето)** стъпки.

В терминологията на ARM, ако инструкцията използва s или d регистри, тя се нарича още VFP – Vector Floating Point инструкция.

За пръв път в ARM такива инструкции се използват в набора ARMv5.

VFP модулите могат да изпълняват скаларни (**с 1 регистър**) и векторни операции (с **2 – 8 регистъра**) [2] на integer и floating point стойности.

Въведение в ARM NEON

SIMD packed mode (Single Instruction Multiple Data, packed mode) - инструкции, при които повече от една операция се прилага на повече от една стойност, записана в един многоразреден (дълъг) регистър-източник.

vadd.i16 q10, q8, q9

Една операция събира два 64-битови регистъра,
но всеки един от четирите 16-битови канала (lanes) в регистъра се събира отделно.

Няма пренос между каналите.

Въведение в ARM NEON

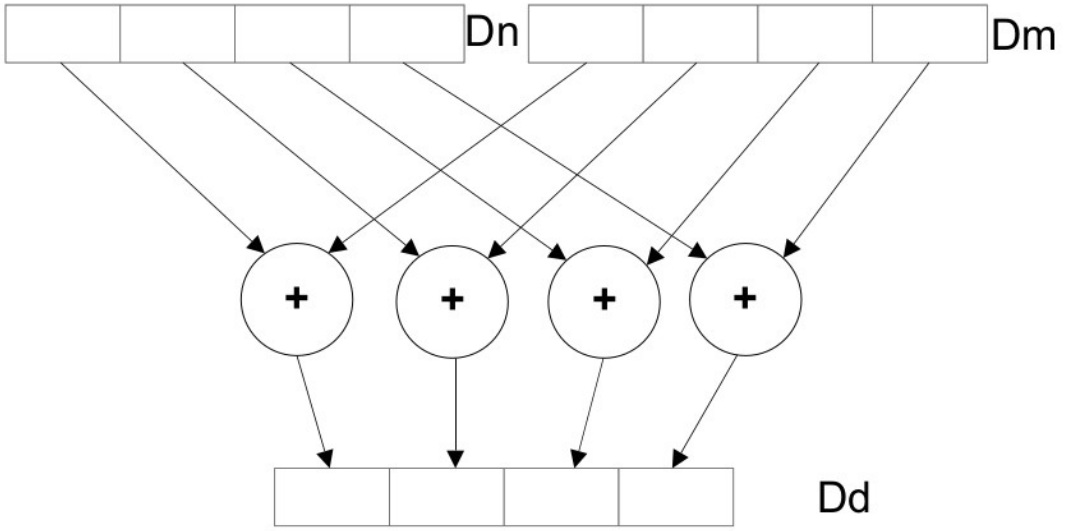


Figure 1-1 Four simultaneous additions

Dn = q8, Dm = q9, Dd = q10

Събирането на 4 целочислени стойности ще стане на 1 стъпка, т.е. **4 паралелни (във времето) операции**.

Инструкциите са налични за пръв път в набора ARMv7 и след това – нагоре ARMv8, ARMv9, и т.н.

Въведение в ARM NEON

Според терминологията на ARM, packed SIMD инструкциите се наричат още Advanced SIMD или **NEON technology** (разбирай – NEON набор от инструкции).

Операндите може също да са:

2x32-битови, 4x16-битови, 8x8-битови = 1x64-битов регистър

4x32-битови, 8x16-битови, 32x8-битови = 1x128-битов регистър

Сравнение на NEON със SIMD и DSP

Сравнение с по-стари SIMD - ARMv6 включва малко на брой SIMD инструкции, които позволяват да се работи с 8- и 16-битови стойности, които са част от 32-битови ARM регистри с общо предназначение. Тези инструкции позволяват да се извършват някои операции **от два до четири пъти** по-бързо от SISD, без да се добавят допълнителни модули.

Класическият ARM1176 процесор е с архитектура ARMv6.

Сравнение на NEON със SIMD и DSP

Пример: инструкцията UADD8 R0, R1, R2 събира 4 8-битови стойности от един 32-битов регистър с други 4 такива от друг 32-битов регистър.

NEON: използва 128-битови вместо 32-битови регистри.

NEON е включен в Cortex-A7 и Cortex-A15, но е опция в други ARMv7-базирани микропроцесори.

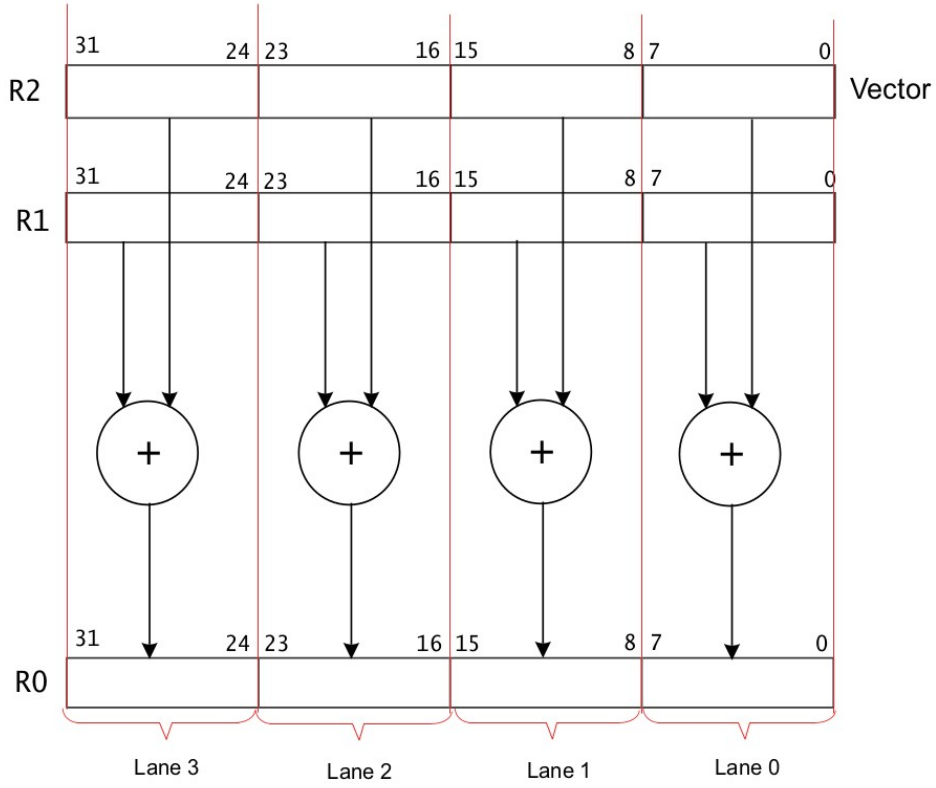


Figure 1-2 A 4-way 8-bit add operation is one of the ARMv6 SIMD instructions

Сравнение на NEON със SIMD и DSP

Table 1-1 compares NEON technology with ARMv6 SIMD.

Table 1-1 Comparison of NEON extensions and ARMv6 SIMD instructions

Feature	ARMv7 NEON extension	ARMv6 SIMD instructions
Packed integers	up to 8 x 8-bit, 4 x 16-bit, and 2 x 32 bit	4 x 8-bit
Data types	Supports integer and, if the processor has a VFP unit, single precision floating-point operations	Supports integer operations only
Simultaneous operations	Maximum sixteen (as 16 x 8-bit)	Maximum four (as 4 x 8-bit)
Dedicated registers	Operates on separate NEON register file that is shared with the VFP unit 32 64-bit registers (or 16 128-bit registers)	Uses 32-bit general purpose ARM registers
Pipeline	Has dedicated pipeline that is optimized for NEON execution	Uses the same pipeline as all other instructions

Сравнение на NEON със SIMD и DSP

Table 1-2 NEON technology compared with MMX and Altivec

	NEON technology	x86 MMX/SSE	Altivec
Number of registers	32 × 64-bit (also visible as 16 × 128-bit)	SSE2: 8 × 128-bit XMM (in x86-32 mode) Additional 8 registers in x86-64 mode	32 × 128-bit
Memory / register operations	Register-based 3-operand instructions	Mix of register and memory operations	Register-based 3- and 4-operand instructions
Load/store support for packed data types	Yes as 2,3, or 4 element	No	No
Move between scalar and vector register	Yes	Yes	No
Floating-point support	Single-precision 32-bit	Single-precision and Double-precision	Single-precision

Сравнение на NEON със SIMD и DSP

Сравнение с DSP

ARM Cortex-A + ARM NEON	ARM Cortex-A + DSP
Модулът е интегриран с конвейера на микропроцесора	DSP работи в паралел с ARM микропроцесора
Многозадачен OS ще се използва и от микропроцесора, и от NEON	DSP няма OS, трябва да се измисли начин за комуникация с ARM Cortex-A
Няма нужда от кеш, защото NEON е част от конвейера	Споделен кеш, който иска cache clean и cache flush операции при прехвърляне на данни
Един процесорен елемент, трябва само един toolchain.	Два процесорни елемента, трябват два различни toolchain-а.

Сравнение на NEON със SIMD и DSP

ВНИМАНИЕ: Cortex-A микропроцесори, които имат NEON, но нямат VFP, не могат да работят с числа с плаваща запетая!

ВНИМАНИЕ: понеже NEON извършва SIMD операции по-бързо от VFP, то от версия ARMv7 нагоре модулът VFP се нарича само FP(U)!

ВНИМАНИЕ: наборът от инструкции на NEON не дефинира тактове за изпълнение на всяка инструкция! Една и съща NEON инструкция може да се изпълнява за различни тактове на различни Cortex-A процесори.

Принцип на работа на NEON

На процесори, на които липсва NEON, инструкции от NEON ще бъдат третирани като *неопределени инструкции* (Undefined instructions).

На процесори, на които има NEON, но модулът **не е включен от CPACR регистъра**, инструкции от NEON ще бъдат също третирани като *неопределени инструкции* (Undefined instructions).

Принцип на работа на NEON

Със скаларни регистри на ARM Cortex-A ядрото може да се индексират векторите (масивите) на NEON.

Пример: копирай 8-битовата най-младша част от R3 в D0, канал 3.

VMOV.8 D0[3], R3

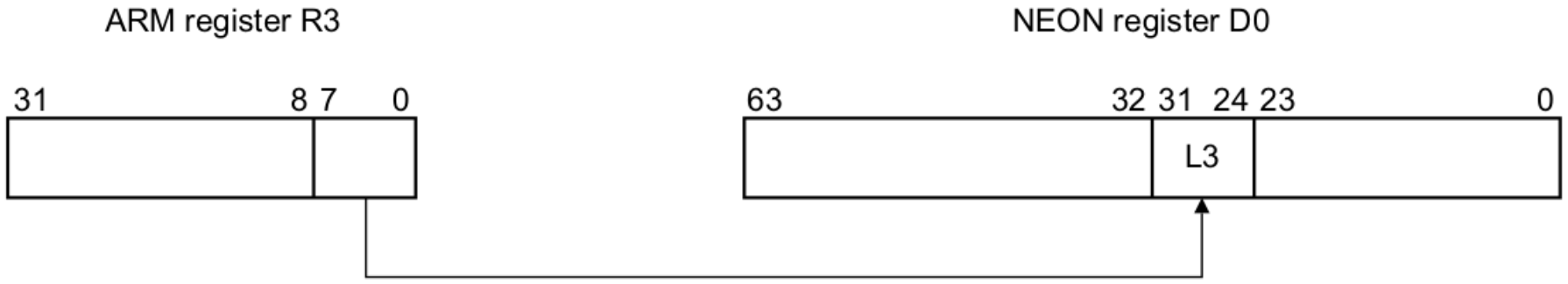


Figure 1-7 Moving a scalar to a lane

Принцип на работа на NEON

Всички инструкции на VFP и NEON започват с буквата **V**.

Инструкциите могат да работят с **променливи с различна разредност и вид**, което е записано в кода на инструкцията.

От гледна точка на Асемблер, разредността и вида се задават като **суфикс на мнемониката**, съставен от буква и цифри (виж следващия слайд).

Разредността на използваните регистри в операндите указва **колко най-много променливи** може да се съдържат в тези регистри – например double регистър може да побере 8 еднобайтови стойности.

Принцип на работа на NEON

Операндите на инструкциите се дефинират чрез суфикс, съставен от буква и цифри, които означават:
*буквата е вида на операндите;
*цифрите са разредността на операндите.

*P = полиноми → операции със степени на двойката

Пример: VMLAL.S8 (signed 8-bit integers)

Table 1-3 NEON data types

	8-bit	16-bit	32-bit	64-bit
Unsigned integer	U8	U16	U32	U64
Signed integer	S8	S16	S32	S64
Integer of unspecified type	I8	I16	I32	I64
Floating-point number	not available	F16	F32 or F	not available
* Polynomial over {0,1}	P8	P16	not available	not available

Принцип на работа на NEON

За VFP модула, операндите са аналогични:

Table 1-5 shows the available data types for VFP instructions.

Table 1-5 VFP data types

	16-bit	32-bit	64-bit
Unsigned integer	U16	U32	not available
Signed integer	S16	S32	not available
Floating point number	F16	F32 (or F)	F64 (or D)

See *VFP instructions* on page C-67 for a description of the instructions specific to the VFP unit.

Принцип на работа на NEON

По подразбиране NEON е изключен.

Преди да почнат да се изпълняват негови инструкции, той трябва да бъде включен със следния код (за armcc):

```
#include <stdio.h>

__asm void enable_neon(void){

MRC p15,0,r0,c1,c0,2      //Read CP Access register
ORR r0,r0,#0x00f00000  //Enable full access to NEON/VFP by enabling access to
                       //coprocessors 10 and 11
MCR p15,0,r0,c1,c0,2     //Write CP Access register
ISB
MOV r0,#0x40000000    //Switch on the VFP and NEON hardware
MSR FPEXC,r0             //Set EN bit in FPEXC

}
```

Инструкции на NEON

Видове поддържани инструкции:

*събиране
*събиране по двойки (събиране на съседни елементи от един вектор)
*умножение (с възможност за насищане)
*умножение с натрупване
*умножение с изваждане
*преместване наляво/дясно и внедряване (insert) на битове
*логически оператори (AND, OR, EOR, !AND, !OR)
*намиране на минимална и максимална стойност
*броене на старшите нули, броене на битове в логическа 1
*други

NEON не поддържа: *делене *корен квадратен

Инструкции на NEON

Инструкциите имат следния формат:

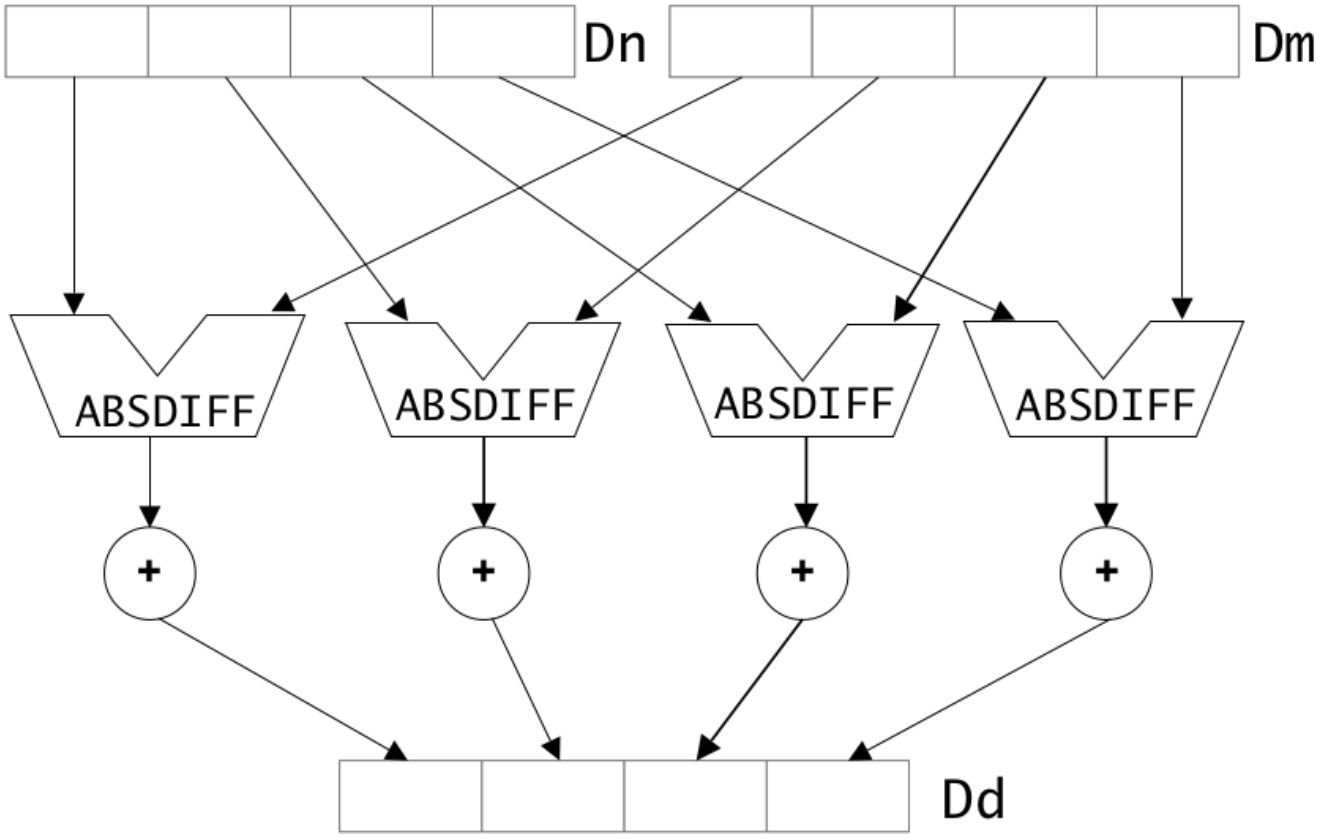
V{<mod>}<op>{<shape>}{<cond>}{.<dt>} <dest1>{, <dest2>}, <src1>{, <src2>}

mod (modifier) – Q(насищане), H(разделяне на две), D(умножение по две), R(закръгляне)
shape – L (long, дълъг), W(wide, широк), N(narrow, тесен)
cond – условие, ако участва в IT инструкция
op – мнемоника на операцията (напр. ADD, SUB)
src1, src2 – регистри-източници
dest1, dest2 – регистри-приемници на резултат
.dt – вид на данните

Инструкции на NEON

*събиране – пример

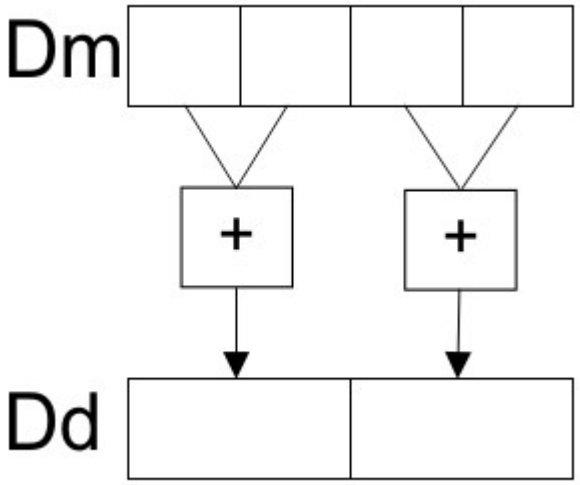
VABA (Vector Absolute Difference and Accumulate) – изважда елементите на един вектор с елементите от друг и изчислява модула на всеки елемент.



Инструкции на NEON

*събиране по двойки – пример

VPADDL (Vector Pairwise Add Long) – събира съседни двойки елементи от два вектора и записва резултата във вектор-приемник. Векторът-приемник е с двойно по-голяма разредност от разредността на операндите.



Инструкции на NEON

*умножение (с възможност за насищане) - пример

VQRDMULH (Vector Saturating Rounding Doubling Multiply Returning High Half) – умножение на операндите един с друг, след което резултатите се удвояват (x2). Накрая от резултата се запазва само старшата част.

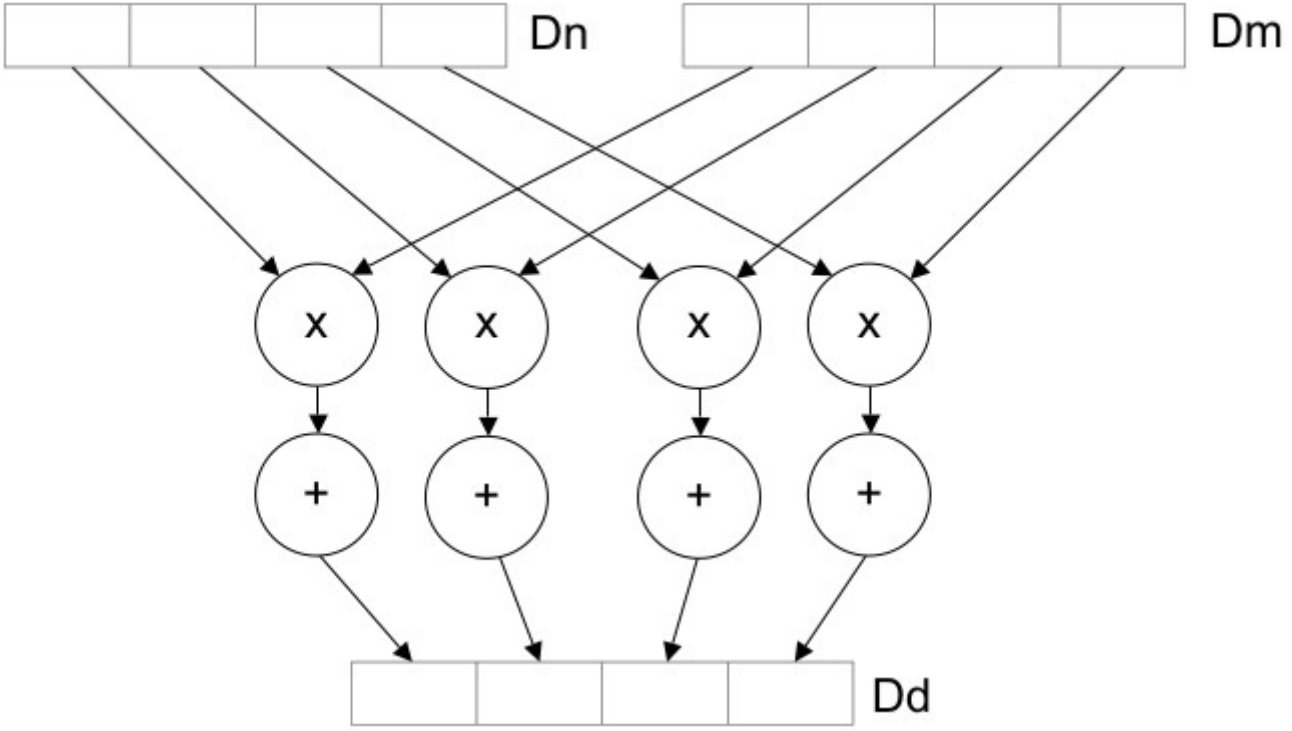
Ако някой от елементите достигне препълване (overflow), битът QC от статус регистъра се качва в 1 и стойността му се насища до максималното число.

Резултатите се закръгляват, ако в инструкцията има R. Ако няма R, резултатите се отрязват след запетаята.

Инструкции на NEON

*умножение с натрупване – пример

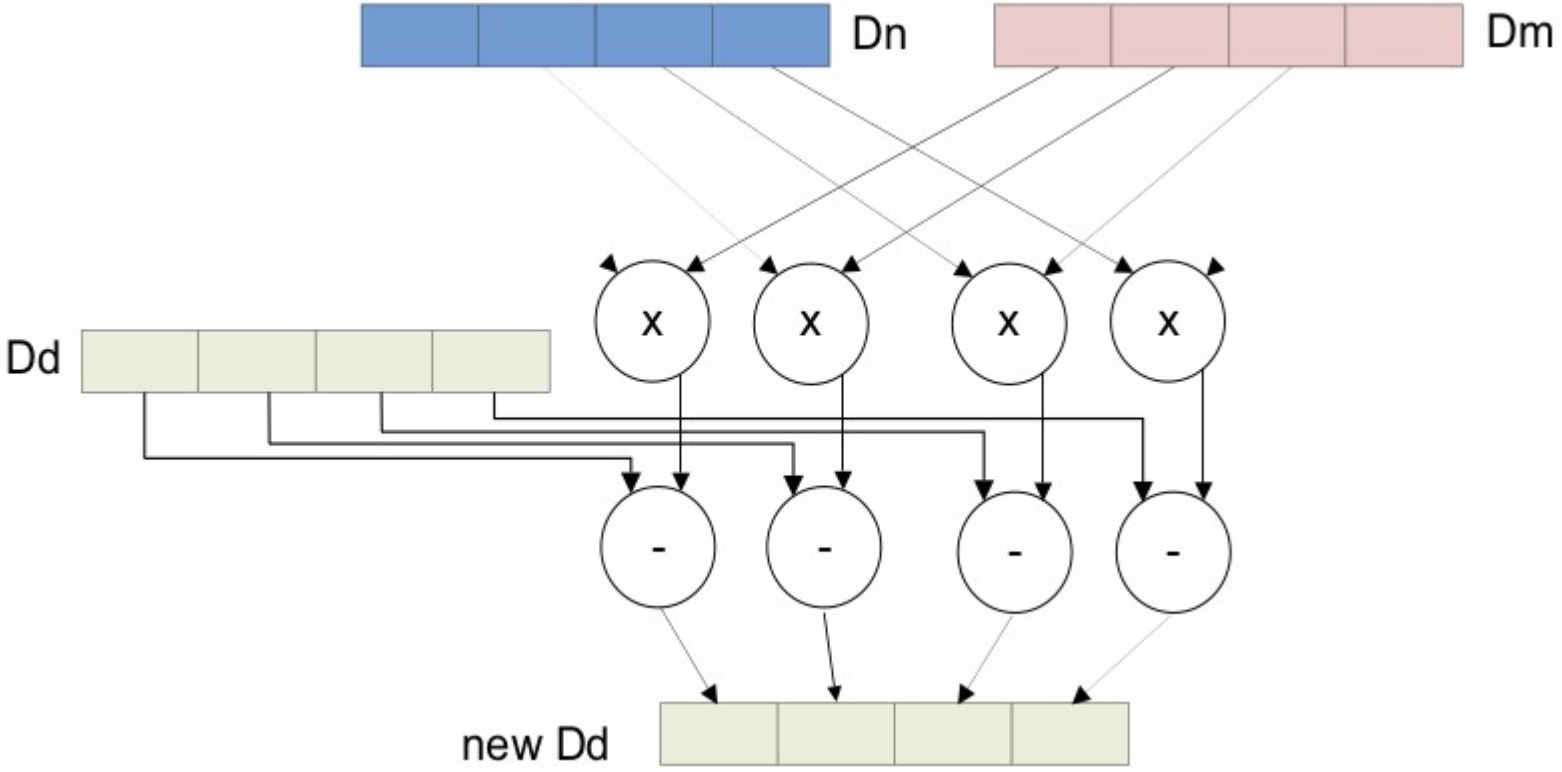
VFMA (Vector Fused Multiply Accumulate) – умножава елементите от един вектор с елементите на друг и записва резултат в приемния вектор. Резултатите се закръгляват. Закръглението не се извършва преди събирането.



Инструкции на NEON

*умножение с изваждане – пример

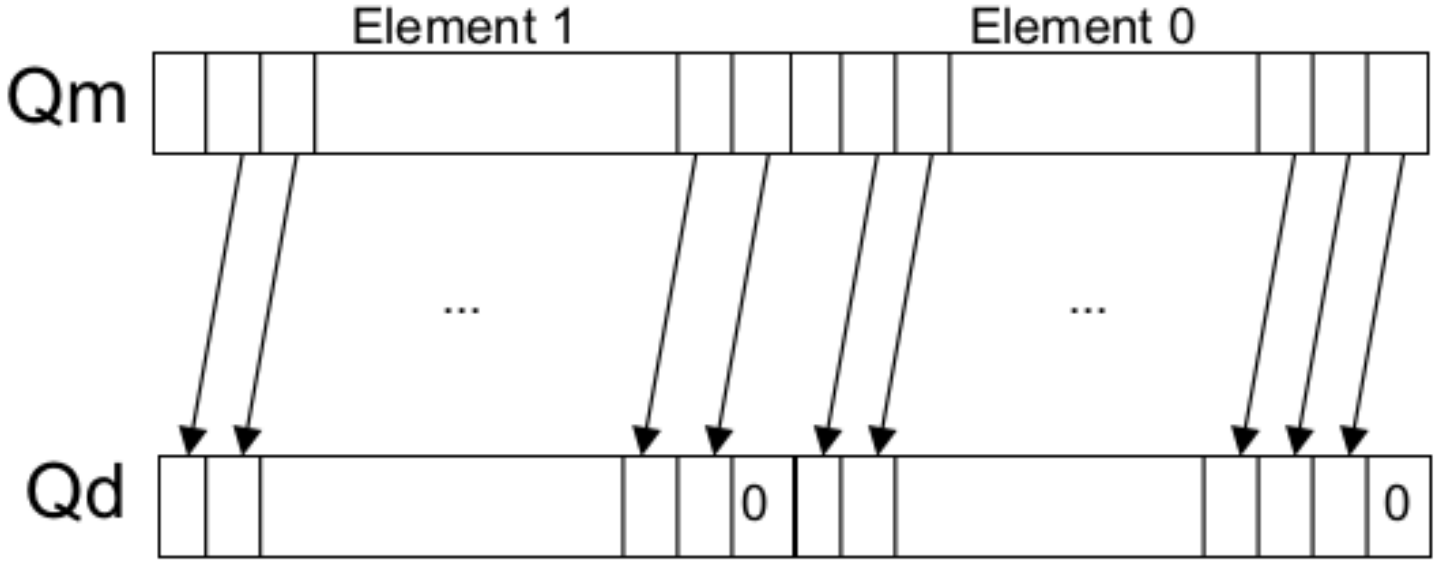
VMFS (Vector Fused Multiply Subtract) – умножава съответните елементи от два вектора. Получените резултати ги вади от стойносите на вектора приемник и ги записва във вектора приемник. Резултатите се закръгляват. Закръглението не се извършва преди изваждането



Инструкции на NEON

*преместване наляво/дясно и внедряване (insert) на битове – пример

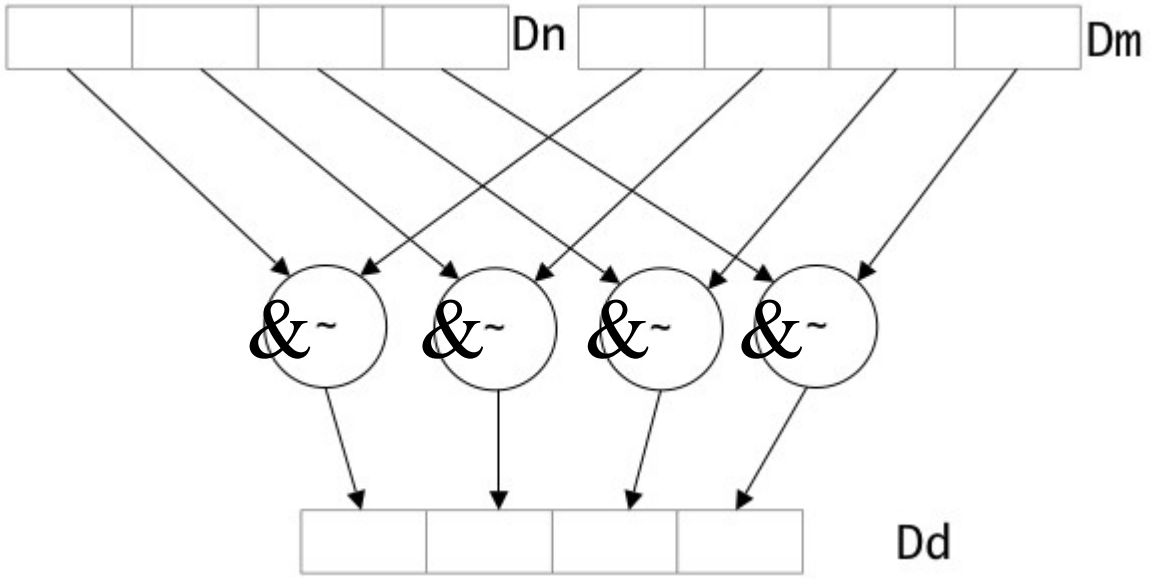
VSLI (Vector Shift Left and Insert) – преместват се наляво битовете от елементите на един вектор. Броя на позициите, с които ще се преместят битовете, се задава с константа.



Инструкции на NEON

*логически оператори – пример

VBIC (Vector Bitwise Clear) – прилага побитово И с инвертиране и записва резултатите в приемния вектор.



Инструкции на NEON

*намиране на минимална и максимална стойност – пример

VMAX (Vector Maximum) и VMIN (Vector Minimum) – сравнява съответните елементи от два вектора и записва по-големия / по-малкия елемент на съответното място във вектора приемник.

Инструкции на NEON

*броене на битове – пример

VCLZ (Vector Count Leading Zeros) – брои непрекъснатата поредица от нули от най-старшия бит на всеки елемент от един вектор и записва броя на нулите за съответните елементи във вектора приемник.

Инструкции на NEON

*други – пример

VCVT (Vector Convert) – копира и преобразува елементите от един вектор и ги записва във вектора приемник.

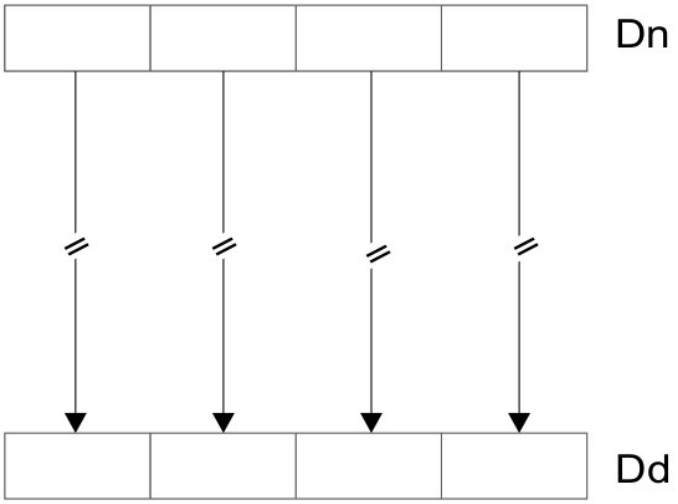
Преобразуването може да е:

-плаваща запетая → целочислена стойност

-целочислена стойност → плаваща запетая

-плаваща запетая → фиксирана запетая

-фиксирана запетая → плаваща запетая



Потребителски стандарт за извикване на функции

Стандартът† за извикване на функции (ABI – application binary interface) дефинира:

*кои регистри на ядрото се използват за предаване на **параметри на функции** [4], [5];

*кои регистри на ядрото се използват за **връщане на резултат**;

*кои регистри не се запазват на стека и може да се използват за временно съхранение на данни (**scratch-pad registers**).

†Procedure Call Standard for the ARM Architecture (AAPCS).

Потребителски стандарт за извикване на функции

Ако се пише Асемблер на ръка (от ARM съветват, че това е единствения начин да се получи свръх-бърз код [2], стр.2-17), трябва да се познава ABI стандарта.

Това е нужно, защото трябва обектовия код на **ръчно-написания** Асемблер да може да се свърже (link) с обектовия код на **автоматично-генерирания** Асемблер от компилатора (в случай, че се работи по голям проект, в който се смесват C и Асемблер).

Потребителски стандарт за извикване на функции

Table 2-2 Use of D registers for parameters and results

D0-D7	Argument registers and return register. If the subroutine does not have arguments or return values, then the value in these registers might be uninitialized.
D8-D15	callee-saved registers.
D16-D31	caller-saved registers.

D0-D7 – параметри на функцията
D0 – връщан резултат
D8-D15 – при влизане в подпрограма, трябва да се запазят на стека, ако ще бъдат използвани
D16-D31 – няма нужда да бъдат запазвани

Потребителски стандарт за извикване на функции

Пример:

void f(uint32_t a, uint64_t b) { … }

→

r0: променлива “a”
r1: неизползван
r2: младша част на променлива “b” – битове [31:0]
r3: старша част на променлива “b” – битове [63:32]

*Понеже 64-битовите числа трябва да започват на четен регистър, затова R1 е неизползван.

Потребителски стандарт за извикване на функции

Пример:

void f(float a, double b)

→

d0: s0: променлива “a”
    s1: неизползван
d1: младша част на променлива “b” – битове [31:0]
    старша част на променлива “b” – битове [63:32]

*d1 заема регистри s2+s3

Потребителски стандарт за извикване на функции

Пример:

void f(float a, double b, float c)

→

d0: s0: променлива “a”

s1: променлива “c”

d1: младша част на променлива “b” – битове [31:0]

старша част на променлива “b” – битове [63:32]

*За да не се заема s4, компилаторът е сложил “c” в регистър s1.

Потребителски стандарт за извикване на функции

Пример:

Ако NEON има само 8 d-регистъра, деветият и следващите параметри се **предават през стека (чрез стековия указател sp)**. В тези случай, параметрите се разполагат последователно в регистри от ядрото и компилатора не оптимизира както в миналия слайд.

void f(double a, double b, double c, double d, double e, double f, float g, double h, double i, float j)
d0: a
d1: b
d2: c
d3: d
d4: e
d5: f
d6: s12: g
    s13: неизползван
d7: h
*sp: i
*sp+8: j
*sp+12: неизползвани (4 байта padding, защото стековият указател sp трябва да е подравнен на 8-байта)

Потребителски стандарт за извикване на функции

Intrinsic функции, използващи NEON, ги има за следните компилатори:
*armcc
*gcc
*g++
*llvm

Компилаторът ще сложи директно инструкцията, отговаряща на intrinsic функцията и всъщност **няма да има извикване на подпрограма**.

Потребителски стандарт за извикване на функции

Включване на NEON инструкции става с аргумента -mfpu за компилатора GCC:

Table 2-4 -mfpu options for Cortex-A processors

Processor	FP only	FP + SIMD
Cortex-A5	-mfpu=vfpv3-fp16 -mfpu=vfpv3-d16-fp16	-mfpu=neon-fp16
Cortex-A7	-mfpu=vfpv4 -mfpu=vfpv4-d16	-mfpu=neon-vfpv4

Processor	FP only	FP + SIMD
Cortex-A8	-mfpu=vfpv3	-mfpu=neon
Cortex-A9	-mfpu=vfpv3-fp16 -mfpu=vfpv3-d16-fp16	-mfpu=neon-fp16
Cortex-A15	-mfpu=vfpv4	-mfpu=neon-vfpv4

Потребителски стандарт за извикване на функции

На компилатора трябва да се укаже кой ABI стандарт да използва. GCC има три варианта:

***-mfloat-abi=soft** – не се използват FPU или NEON инструкции. Работи се само с целочислените регистри. Емулира всички float операции чрез извиквания на библиотечни функции.

Потребителски стандарт за извикване на функции

***-mfloat-abi=softfp** – използва същия стандарт за извикване на функции като -mfloat-abi=soft, но където може използва FPU и NEON инструкции. Код, компилиран с този аргумент, **може да се линква с обектовия код**, компилиран със **soft** аргумента. Предаването на параметри на функции става само през целочислени регистри от ARM ядрото (и стека).

***-mfloat-abi=hard** – използва FPU и NEON инструкции. Стандартът за извикване на функции е съобразен с регистрите на FPU и NEON и се различава от стандарта на предишните два аргумента.

Потребителски стандарт за извикване на функции

*В стандартния език C не са предвидени оператори за паралелна обработка

*За това е необходимо да се укаже на компилатора, чрез аргументи, че се изисква автоматична векторизация на кода.

*За GCC и NEON автоматичната векторизация се включва с аргументите [3]:

-ftree-vectorize -mfpu-neon -O2 (или -O3) -Otime --vectorize

Потребителски стандарт за извикване на функции

*Аргументите на GCC **-fdump-tree-vect** и **-ftree-vectorizer-verbose**=level, където level=1-9, са за принтиране на информация относно векторизирането на кода, който се компилира.

Тази информация може да помогне за оптимизация на кода, така че да се използва NEON на максимум.

Потребителски стандарт за извикване на функции

Детектиране на NEON от кода може да стане по два начина:

*__статично__ (по време на компилацията) – чрез макроса _ _ARM_NEON_ _ от CMSIS библиотеката

*__динамично__ (по време на изпълнение на кода) – чрез регистър (само за четене, read-only) MVFR0 (Media and VFP Feature Register 0), битово поле за размер на регистровия файл A_SIMD:
0b0000 – NEON не се поддържа
0b0001 – има NEON, 16 x 64-битови регистъра
0b0010 – има NEON, 32 x 64-битови регистъра

Внедряване на код на Асемблер в програма на C

Внедряване на асемблерна програма в програма на C може да стане чрез:

*вмъкване на **компилаторни asm конструкции** (statements)

*използване на C вътрешни функции (**instrinsic functions**) , които съответстват на една или няколко инструкции напр. Wfi → __WFI();

***линкване** на асемблерен обектов файл с обектовите файлове на C

Внедряване на код на Асемблер в програма на C

*вмъкване на **компилаторни asm конструкции** (statements)

asm(“nop”); → рисковано, подлежи на оптимизация

asm **volatile** (“nop”); → безопасно, не подлежи на оптимизация

Внедряване на код на Асемблер в програма на C

asm volatile (асемблерна инструкция %[*oop1*], %[*iop1*], %[*iop2*]
: *oop1*
: *iop1*, *iop2*
:засегнати регистри);

където *oop1* е изходен операнд1, *iop1 е* входен операнд1 , *iop2 е* входен операнд2.

oop1, iop1, iop2 са имена на C променливи от програмата

Внедряване на код на Асемблер в програма на C

Вместо %[*oop1*], %[*iop1*], %[*iop2*] може да се използват безименните %0, %1, %2, където цифрата указва индекса на операнда в инструкцията

Дали ще има нула, един или два входни операнда зависи от инструкцията.

Засегнати регистри (clobber list) – незадължително поле, указва асемблерната инструкция дали ще засегне **регистър от ядрото, статус бит или памет.** Така компилаторът ще знае да запази засегнатите регистри преди инструкцията и ще ги възстанови след инструкцията.

Внедряване на код на Асемблер в програма на C

Операндите могат да имат т.нар. ограничения (**constraints**), които указват вида на операнда и достъпа до него.

Видът на операнда се задава с една буква. На следващия слайд е показана таблица със значението на всички букви.

Достъпът до операнда се задава със символ:

= операндът е само за запис (write only).
+ операндът е за запис и четене (read-write). Да се използва само за изходни операнди.
& регистър, който може да бъде използван само като изходен.
без специален символ - операндът е само за четене (read only).

Внедряване на код на Асемблер в програма на C

Буква (constraint) [6]	Значение
t	32 x32-битови регистри s0 - s31 за числа с плаваща запетая.
h	Регистри r8 – r15.
G	Непосредствена константа с плаваща запетая.
H	Същата като G, но с отрицание.
I	Непосредствена целочислена константа.
J	Константа за индексна адресация (-4095 ↔ +4095).
K	Същата като I, но инвертирана (inverted).
L	Същата като I, но с отрицание (negated).
I	Същата като r.
M	Константа в диапазона 0 ↔ 32.
m	Валиден адрес от картата на паметта.
N	Константа в диапазона 0 ↔ 31.
O	Константа в диапазона -508 ↔ 508, кратна на 4.
r	Регистри r0 - r15
w	16 x64-битови регистри d0 - d15 за числа с плаваща запетая.
X	Всички операнди.

Внедряване на код на Асемблер в програма на C

Възможно е променлива на C да бъде поместена в точно определен регистър от ядрото:

```
void func(void){
register unsigned int my_var asm(“r5”);
…
asm volatile (“rev r5, r5”);
}
```

ВНИМАНИЕ! Ако променливата спре да бъде използвана в програмата, компилаторът може да реши да използва въпросния регистър за други цели.

Внедряване на код на Асемблер в програма на C

Възможно е в една **asm** конструкция да се поместят няколко асемблерни инструкции.

Пример: код от Линукс, който модифицира CPSR регистъра, за да промени режимите на работа на ядрото. Понеже CPSR не е част от картата на паметта на процесора, а е част от ядрото на процесора и се достъпва само с MSR/MRS инструкции, то в програма, използваща стандартно C, няма друг начин да се извърши тази операция.

ВНИМАНИЕ! Забележете новите редове \n на всеки ред и терминираща наклонена черта \ за многоредови низове.

Внедряване на код на Асемблер в програма на C

```
void __naked get_fiq_regs(struct pt_regs *regs)
{
    register unsigned long tmp;
    asm volatile (
                "mov ip, sp                 \n\
                stmfd sp!, {fp, ip, lr, pc}  \n\
                sub fp, ip, #4              \n\
                mrs %0, cpsr                \n\
                msr cpsr_c, %2              @ select FIQ mode\n\
                mov r0, r0                  \n\
                stmia %1, {r8 – r14}        \n\
                msr cpsr_c, %0              @ return to SVC mode\n\
                mov r0, r0                   \n\
                ldmfd sp, {fp, sp, pc}"
        : "=&r" (tmp)
        : "r" (&regs->ARM_r8), "I" (PSR_I_BIT | PSR_F_BIT | FIQ_MODE)
        );
}
```

Внедряване на код на Асемблер в програма на C

*използване на C вътрешни функции (**instrinsic functions**) за NEON:

→ за ARM GCC, пълен списък с функции има в [a]
→ за ARMCC, пълен списък с функции има в [b]

Примери (валидни за armcc и gcc):

int8x8_t vadd_s8(**int8x8_t** a, **int8x8_t** b);
// VADD.I8 d0,d0,d0
int32x2_t vmul_s32(**int32x2_t** a, **int32x2_t** b);
// VMUL.I32 d0,d0,d0
float32x2_t vrsqrts_f32(**float32x2_t** a, **float32x2_t** b);
// VRSQRTS.F32 d0, d0, d0

Внедряване на код на Асемблер в програма на C

Програма на C за събиране на 100 x 16-битови целочислени стойности от масив, в паралел по 4 [c].

Внедряване на код на Асемблер в програма на C

```
/* neon_example.c - Neon intrinsics example program */
#include <stdint.h>
#include <stdio.h>
#include <assert.h>
#include <arm_neon.h>

/* fill array with increasing integers beginning with 0 */
void fill_array(int16_t *array, int size){
    int i;
    for (i = 0; i < size; i++){
        array[i] = i;
     }
}
```

Внедряване на код на Асемблер в програма на C

```
/* main function */
int main()
{
    int16_t my_array[100];
    int result;

    fill_array(my_array, 100);

    result = sum_array(my_array, 100);

    printf("The total sum is %d\n", result);

    return 0;
}
```

Внедряване на код на Асемблер в програма на C

```
// return the sum of all elements in an array. This works by calculating 4 totals (one for each
// lane) and adding those at the end to get the final total
int sum_array(int16_t *array, int size){
    int16x4_t acc = vdup_n_s16(0); // initialize the accumulator vector to zero
    int32x2_t acc1;
    int64x1_t acc2;

    assert((size % 4) == 0); // assumes the size of the array is a multiple of 4

    for (      ; size !=0; size -= 4){ // counting backwards gives better code
        int16x4_t vec;

        vec = vld1_s16(array); // load 4 values in parallel from the array
        acc = vadd_s16(acc, vec);// add the vector to the accumulator vector
        array += 4; // increment the array pointer to the next element
    }

    // calculate the total, see next slide
    acc1 = vpaddl_s16(acc);
    acc2 = vpaddl_s32(acc1);

    return (int)vget_lane_s64(acc2, 0);  // return the total as an integer
}
```

Внедряване на код на Асемблер в програма на C

***линкване** на асемблерен обектов файл с обектовите файлове на C [d] – има предимства пред изброените досега два метода (instrinsics, inline asm):

-може да се използват всички асемблерни директиви;
-няма оптимизация на кода – съответно няма странични ефекти;
-достъп до всички регистри на ядрото.

За да е успешно едно такова линкване, трябва да бъде спазено изискването обектовия файл да отговаря на Procedure Call Standard for the ARM Architecture (AAPCS).

Внедряване на код на Асемблер в програма на C

В асемблерния файл може да се достъпват **глобални C променливи**.

Пример: променливата globvar е в C файл и ще бъде заредена в регистър R1, увеличена с 2, записана обратно в globvar.

```
    PRESERVE8
    AREA    globals,CODE,READONLY
    EXPORT   asmsubroutine
    IMPORT   globvar
asmsubroutine
    LDR  R1, =globvar   ; read address of globvar into R1
    LDR  R0, [R1]       ; load value of globvar
    ADD  R0, R0, #2
    STR  R0, [R1]       ; store new value into globvar
    BX   lr
    END
```

Внедряване на код на Асемблер в програма на C

Във файл на C може да се **извикват асемблерни подпрограми**.

За целта, името на подпрограмата от асемблерния файл трябва да се направи глобално с директивата EXPORT.

След това, във файлът на C трябва да се декларира extern прототип на функция. Името на функцията трябва да съвпада с името на EXPORT-натата подпрограма.

Внедряване на код на Асемблер в програма на C

Пример:

```
    PRESERVE8
    AREA    CODE, READONLY
    EXPORT strcopy
;R0 points to destination string.
;R1 points to source string.

strcopy
    LDRB R2, [R1],#1
    STRB R2, [R0],#1
    CMP  R2, #0
    BNE  strcopy
    BX   lr

    END
```

```
#include <stdio.h>

extern void strcopy(char *d, const char *s);

int main(){
    const char *srcstr = "source ";
    char dststr[] = "destination";

    printf("Before copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    strcopy(dststr, srcstr);
    printf("After copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);

    return 0;
}
```

Внедряване на код на Асемблер в програма на C

Във файл на Асемблер може да се **извикват функции на C**.

За целта, името на C функцията от C файла трябва да се добави с директивата IMPORT.

Пример: на следващия слайд, асемблерната програма е еквивалентна на:

```
int asm_func(int i) {
    return c_func(i, 2*i, 3*i, 4*i, 5*i);
}
```

Внедряване на код на Асемблер в програма на C

Пример:

```
PRESERVE8
EXPORT f
AREA f, CODE, READONLY
IMPORT c_func

STR lr, [sp, #-4]!
ADD R1, R0, R0
ADD R2, R1, R0
ADD R3, R1, R2
STR R3, [sp, #-4]!
ADD R3, R1, R1
BL c_func
ADD sp, sp, #4
LDR pc, [sp], #4

END
```

```
int c_func(int a, int b, int c, int d, int e)
{
    return a + b + c + d + e;
}
```

Литература

[1] “The ARM Cortex-A9 Processors”, white paper, ARM Ltd, rev.2.0, 2009.
[2] “NEON”, Programmer’s Guide, v1.0, ARM DEN0018A (ID071613), 2013.
[3] “ARM Cortex-A Series”, Programmer’s Guide, DEN0013D (ID012214), p.95-p.107, ARM Ltd, 2013.
[4] “ARM Cortex-A Series”, Programmer’s Guide, DEN0013D (ID012214), p.207-p.217, ARM Ltd, 2013.
[5] “ARM Developer Suite”, Developer Guide, v1.2, ARM DUI 0056D, p.21-p.43, ARM Ltd, 2001.

Външни връзки

[a] https://gcc.gnu.org/onlinedocs/gcc-4.9.4/gcc/ARM-NEON-Intrinsics.html
[b] https://developer.arm.com/documentation/dui0205/g/neon-intrinsics?lang=en (RealView Development Suite / Appendix F. NEON Intrinsics)
[c] https://developer.arm.com/documentation/dui0205/g/neon-intrinsics/introduction?lang=en
[d] https://developer.arm.com/documentation/dui0471/g/Mixing-C--C----and-Assembly-Language/Instruction-intrinsics--inline-and-embedded-assembler?lang=en