

# Модули за числа с плаваща запетая



**Автор: гл. ас. д-р инж. Любомир Богданов**



Европейски съюз

**ПРОЕКТ BG051PO001--4.3.04-0042**

***„Организационна и технологична инфраструктура за учене през целия живот и развитие на компетенции”***

Проектът се осъществява с финансовата подкрепа на  
Оперативна програма „Развитие на човешките ресурси”,  
съфинансирана от Европейския социален фонд на Европейския съюз

***Инвестира във вашето бъдеще!***



Европейски социален фонд

# Съдържание

1. Представяне на числа в двоичен вид
2. Барабанен премествач (barrel shifter)
3. Модули за числа с плаваща запетая (FPU)
4. Умножители (MPU)
5. Модул за защита на паметта (MPU)
6. Модул за организация на паметта (MMU)
7. Кеш памети

# Представяне на числа в двоичен

**ВИД**  
Цифровите схеми могат да обработват само числа, представени с битове, които може да са 0 или 1.

*Проблем:*

\*с 1 бит може да се представят само две числа, но за да върши нещо полезно,  $\mu$ PU трябва да може да обработва и по-големи числа, например  $0 \div 10$ ,  $100 \div 200$ ,  $1000000 \div 3500000$ , и т.н.

*Решение:*

\*да се използва група от битове, която представя числата в двоична бройна система. Колкото повече бита включва една група, толкова по-голямо десетично число може да представи тя.

# Представяне на числа в двоичен ВИД

**Прав код без знак (straight binary)** – групата от битове се съпоставя директно на положителни числа от десетичната бройна система [1].

Decimal number	Binary number
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

# Представяне на числа в двоичен

## ВИД

*Проблем:*

\*Как може да се представят целочислени стойности със знак?

*Решение:*

\*прав код (signed magnitude)

\*обратен код (one's complement)

\*допълнителен код (two's complement)

*Проблем:*

\*Как може да се представят дробни числа (числа със запетая)?

*Решение:*

\*Числа с фиксирана запетая

\*Числа с плаваща запетая (IEEE 754-1985)

# Представяне на числа в двоичен ВИД

**Прав код** (signed magnitude) – най-старшият бит (т.е. битът най-вляво) от двоичното число се заделя, за да се указва знак. Когато  $MSb = 1$ , значи че знакът е минус, а оставащите младши битове указват отрицателно число. Когато  $MSb = 0$ , знакът е плюс и числото е положително.

*Проблем:*

при такова представяне, в обхвата от всички числа има **две нули**.

1000.0000 (0-)  
0000.0000 (0+)

# Представяне на числа в двоичен

## ВИД

Работата с такива числа ще доведе до усложняване на хардуера, за да се преодолее този проблем.

Decimal number	8-bit binary signed magnitude number
-127	1111 1111
...	...
-3	1000 0011
-2	1000 0010
-1	1000 0001
-0	1000 0000
+0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
...	...
127	0111 1111



# Представяне на числа в двоичен ВИД

**Обратен код** (one's complement) – отрицателните числа се представят с инвертирани битове на техния положителен еквивалент.

При събиране на две такива числа трябва да се следи за пренос (Carry). Ако има, този бит трябва да бъде добавен обратно към резултата, иначе числото ще е грешно.

*Проблем:*

при такова представяне, също има **две нули**.

1111.1111 (0-)  
0000.0000 (0+)

# Представяне на числа в двоичен ВИД

Decimal number	8-bit binary one's complement number
-127	1000 0000
...	...
-3	1111 1100
-2	1111 1101
-1	1111 1110
-0	1111 1111
+0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
...	...
127	0111 1111

# Представяне на числа в двоичен

## ВИД

*Пример* – да се съберат числата -5 и +3. Нека те се представят с обратен код.

$$\begin{array}{r} + \quad 1111 \ 1010 \\ \quad 0000 \ 0011 \\ \hline 1111 \ 1101 \end{array}$$

$$\begin{array}{r} -5 \\ +3 \\ \hline -2 \end{array}$$

CORRECT

# Представяне на числа в двоичен ВИД

*Пример* – да се съберат числата -2 и +6. Нека те се представят с обратен код.

**ВНИМАНИЕ!** Дължината на целочислените стойности е фиксирана ( $n = 4, 8, 16, 24, 32$  бита). Всеки бит, който се пренесе на позиция  $n + 1$  **бива загубен**.

$$\begin{array}{r} + 1111\ 1101 \\ 0000\ 0110 \\ \hline \end{array}$$

$$\begin{array}{r} -2 \\ +6 \\ \hline \end{array}$$

$$1\ 0000\ 0011$$

$$3$$

INCORRECT

Carry :

+

1

$$\begin{array}{r} + 1 \\ \hline 0000\ 0100 \end{array}$$

$$4$$

CORRECT

this bit is cut off

# Представяне на числа в двоичен ВИД

**Допълнителен код** (two's complement) — отрицателните числа се представят с инвертирани битове на техния положителен еквивалент и след това добавяне на числото 1.

При такова представяне **има само една нула** (0000.0000).

Аритметиката се извършва **със същия хардуер**, с който се извършва аритметиката на числа, представени с **прав код без знак**.

Компиляторът е отговорен за преобразуването на числата в допълнителен код, преди да се заредят в паметта.

# Представяне на числа в двоичен ВИД

С 8-битов допълнителен код могат да се представят числата  $-128 \div +127$  (докато с обратния код имаме неефективно ползване на битовете:  $-127 \div +127$ ).

Допълнителният код е най-често използвания в цифровите системи.

# Представяне на числа в двоичен ВИД

Decimal number	8-bit binary one's complement number
-128	1000 0000
...	...
-3	1111 1101
-2	1111 1110
-1	1111 1111
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
...	...
127	0111 1111

# Представяне на числа в двоичен ВИД

*Пример* – да се изчисли допълнителния код на числото -87.


$$\begin{array}{r} \text{(bitwise invert)} \sim 0101 \ 0111 \quad +87 \\ \hline + 1010 \ 1000 \\ \phantom{+} \phantom{1010} \phantom{1000} 1 \\ \hline 1010 \ 1001 \quad -87 \end{array}$$



# Представяне на числа в двоичен ВИД

*Пример* – да се изчисли сумата на числата в допълнителен код  $(-87) + (+95)$ .

$$\begin{array}{r} + 1010\ 1001 \\ 0101\ 1111 \\ \hline 1\ 0000\ 1000 \end{array} \quad \begin{array}{r} -87 \\ +95 \\ \hline +8 \end{array}$$

Carry :   
this bit is cut off

CORRECT

*Забележка:* всъщност Carry се съхранява в STATUS регистъра на  $\mu$ PU ядро и може да се използва за аритметика с числа с произволна разредност (софтуерно подсилване на  $\mu$ PU).

# Представяне на числа в двоичен ВИД

**Логическо преместване наляво/дясно (Logical Shift Left / Logical Shift Right)** – добавяне на 0 към най-младшата/най-старшата част на едно число, която нула премества всички останали битове един индекс наляво/надясно.

# Представяне на числа в двоичен ВИД

LSL

```
//0x32 = 50
uint8_t myvar = 0x32;

for(i = 0; i < 8; i++){
    myvar = myvar << 1;
}
```

(i[-1] 0050) 00110010

=====

(i[0] 0100) 01100100

(i[1] 0200) 11001000

(i[2] 0144) 10010000

(i[3] 0032) 00100000

(i[4] 0064) 01000000

(i[5] 0128) 10000000

(i[6] 0000) 00000000

(i[7] 0000) 00000000

# Представяне на числа в двоичен ВИД

LSR

```
//0x32 = 50
uint8_t myvar = 0x32;

for(i = 0; i < 8; i++){
    myvar = myvar >> 1;
}
```

(i[-1] 0050) 00110010

=====

(i[0] 0025) **00011001**

(i[1] 0012) **00001100**

(i[2] 0006) **00000110**

(i[3] 0003) **00000011**

(i[4] 0001) **00000001**

(i[5] 0000) **00000000**

(i[6] 0000) **00000000**

(i[7] 0000) **00000000**

# Представяне на числа в двоичен ВИД

**Аритметично преместване наляво/дясно (Arithmetic Shift Left / Arithmetic Shift Right)** – добавяне на 0 към най-младшата/най-старшата част на едно число, която нула премества всички останали битове един индекс наляво/надясно, **но знакът на числото се запазва при преместване надясно**. Преместването наляво не запазва знака, т.е.  $ASR = LSR$ .

**Използват се числа в допълнителен код (two's complement).**

# Представяне на числа в двоичен ВИД

```
//0x32 = 50
int8_t myvar = 0x32;

for(i = 0; i < 8; i++){
    myvar = myvar << 1;
}
```

ASL+

(i[-1] 0050)	00110010
=====	
(i[0] 0100)	01100100
(i[1] -056)	11001000
(i[2] -112)	10010000
(i[3] 0032)	00100000
(i[4] 0064)	01000000
(i[5] -128)	10000000
(i[6] 0000)	00000000
(i[7] 0000)	00000000

# Представяне на числа в двоичен

## ВИД

ASR+

```
//0x32 = 50
int8_t myvar = 0x32;

for(i = 0; i < 8; i++){
    myvar = myvar >> 1;
}
```

(i[-1] 0050) 00110010

=====

(i[0] 0025) **00011001**

(i[1] 0012) **00001100**

(i[2] 0006) **00000110**

(i[3] 0003) **00000011**

(i[4] 0001) **00000001**

(i[5] 0000) **00000000**

(i[6] 0000) **00000000**

(i[7] 0000) **00000000**

# Представяне на числа в двоичен

ВИД

ASL-

(i[-1] -050) 11001110

=====

```
int8_t myvar = -50;
```

```
for(i = 0; i < 8; i++){  
    myvar = myvar << 1;  
}
```

(i[0] -100) 10011100

(i[1] 0056) 00111000

(i[2] 0112) 01110000

(i[3] -032) 11100000

(i[4] -064) 11000000

(i[5] -128) 10000000

(i[6] 0000) 00000000

(i[7] 0000) 00000000



# Представяне на числа в двоичен

**ВИД**

**ASR-**

(i[-1] -050) 11001110

=====

```
int8_t myvar = -50;
```

```
for(i = 0; i < 8; i++){  
    myvar = myvar >> 1;  
}
```

(i[0] -025) 11100111

(i[1] -013) 11110011

(i[2] -007) 11111001

(i[3] -004) 11111100

(i[4] -002) 11111110

(i[5] -001) 11111111

(i[6] -001) 11111111

(i[7] -001) 11111111

В C компилаторът избира правилните инструкции за премествания оператор в зависимост от типа на променливата.

# Представяне на числа в двоичен ВИД

*Изводи:*

**\*операторът за преместване може да се използва за преобразуване на паралелна в последователна информация**

**\*операторът за преместване може да се използва за умножение и деление, НО:**

→ умножението на числа без знак е валидно, докато не настъпи пренос;

→ умножението на отрицателни числа е валидно, докато не настъпи препълване (т.е. резултат  $> -128$  за 8-битови числа,  $> -32768$  за 16-битови числа, и т.н.);

→ делението на отрицателни числа е валидно, докато се стигне -1.

# Представяне на числа в двоичен ВИД

**Ротиране наляво/дясно (ROtate Left/Right)** – изместване на най-старшия/най-младшия бит на едно число, и добавянето му в началото/края на числото, като всички останали битове се преместват един индекс наляво/надясно.

# Представяне на числа в двоичен ВИД

## ROL

```
uint8_t myvar = 0x53;
```

```
for(i = 0; i < 8; i++){  
    myvar = __rotate_left(myvar);  
}
```

В С няма оператор за ротиране, :-)

но има inline Асемблер :-)  
и intrinsic функции, :-)

с който може да се извикат директно  
инструкции за ротиране, ако се  
поддържат от микропроцесора.

Първоначално: 1010 0011

=====

Итерация (0): 0100 0111

Итерация (1): 100 01110

Итерация (2): 0001 1101

Итерация (3): 0011 1010

Итерация (4): 0111 0100

Итерация (5): 1110 1000

Итерация (6): 1101 0001

Итерация (7): 1010 0011

# Представяне на числа в двоичен ВИД

## ROR

```
uint8_t myvar = 0x53;
```

```
for(i = 0; i < 8; i++){  
myvar = __rotate_right(myvar);  
}
```

Първоначално: 1010 0011

=====

Итерация (0): 1101 0001

Итерация (1): 1110 1000

Итерация (2): 01110 100

Итерация (3): 001110 10

Итерация (4): 0001110 1

Итерация (5): 1000 1110

Итерация (6): 0100 0111

Итерация (7): 1010 0011

# Представяне на числа в двоичен ВИД

**Числа с фиксирана запетая** (fixed point number) - двоични числа, които се съпоставят на дробни числа в десетична бройна система, при които разделителната способност (резолюцията) на числото преди и числото след запетаята е фиксирана. Такива числа може да се каже, че са фиксирани целочислени стойности, кратни на някакво малко число (напр. един час се състои от  $6 \times 10$  минутни части).

Хардуерният модул, който ще извършва изчисленията с тези числа, трябва да знае предварително колко бита са заделени за цялото число и колко за числото след запетаята.

# Представяне на числа в двоичен

## ВИД

За числа с фиксирана запетая  $Q_n$  се използва формулата [3]:

$$Q_n(x_q) = x_i * 2^{-n}$$

където  $x_i$  е цяло число, отговарящо на дробното число  $x_q$ , а  $n$  е броя на битовете, заделени за числото, представлящо дробната част.

*Пример* – нека разредността на **цялото число** (битове за целочислена + битове за дробна част) с фиксирана запетая да е **16 бита**. Ако за **дробна част** се заделят **12 бита**, остават **4 бита** за **целочислената част**. Числото се отбелязва като **Q12**.

# Представяне на числа в двоичен ВИД

*Пример* – числото 3.625, ако се представи с фиксирана запетая, ще бъде записано в паметта на контролера като  $0011.101000000000_{(2)} = 14848_{(10)}$ . Това число се получава по следния начин:

$$Q12(3.625) = 14848 * 10^{-12}$$

откъдето се вижда, че **хардуерът предварително трябва да знае за  $10^{-12}$**



# **Представяне на числа в двоичен ВИД**

Числата с фиксирана запетая може да се използват в много приложения, където изискванията за точността на дробните числа не са големи.

**Числата с фиксирана запетая са със знак.**

Texas Instruments са приготвили таблица с обхватите на всички възможни 16-битови числа с фиксирана запетая и съответните им резолюции (виж следващия слайд).

# Представяне на числа в двоичен ВИД

Type	Bits		Range		Resolution
	Integer	Fractional	Min	Max	
_q15	1	15	-1	0.999 970	0.000 030
_q14	2	14	-2	1.999 940	0.000 061
_q13	3	13	-4	3.999 830	0.000 122
_q12	4	12	-8	7.999 760	0.000 244
_q11	5	11	-16	15.999 510	0.000 488
_q10	6	10	-32	31.999 020	0.000 976
_q9	7	9	-64	63.998 050	0.001 953
_q8	8	8	-128	127.996 090	0.003 906
_q7	9	7	-256	255.992 190	0.007 812
_q6	10	6	-512	511.984 380	0.015 625
_q5	11	5	-1,024	1,023.968 750	0.031 250
_q4	12	4	-2,048	2047.937 500	0.062 500
_q3	13	3	-4,096	4,095.875 000	0.125 000
_q2	14	2	-8,192	8,191.750 000	0.250 000
_q1	15	1	-16,384	16,383.500 000	0.500 000

# Представяне на числа в двоичен ВИД

От тази таблица се вижда основния недостатък — **максималното и минималното число**, което може да се представи е **ограничено от разредността на дробната част**.

При числата с плаваща запетая, ако дробната част е малка, то целочислената стойност може да е голяма. И обратното — малки целочислени стойности ще позволят представянето на много знаци след запетаята.

# Представяне на числа в двоичен ВИД

**Числа с плаваща запетая** (floating point numbers) – двоични числа, които се съпоставят на дробни числа в десетична бройна система. Съществуват различни стандарти, но най-често използвания е **IEEE754-1985**, който гласи:

Едно 32-битово дробно число се представя със следните битови полета:

- \*1 бит за **знак**

- \*8 бита за **експонента**

- \*23 бита за **мантиса** (mantissa, significand)

# Представяне на числа в двоичен ВИД

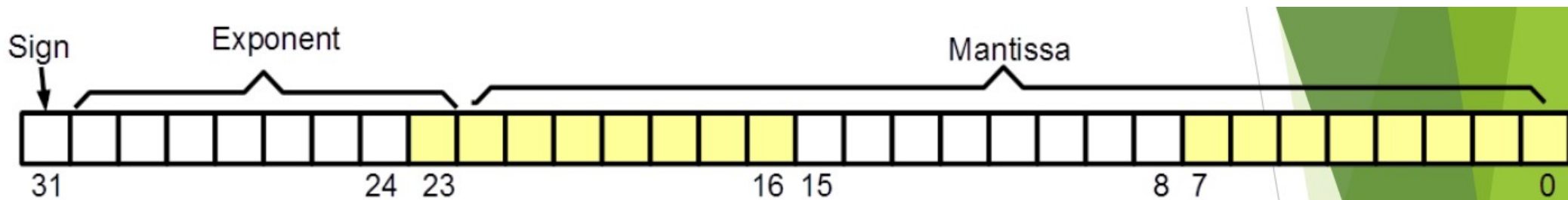
Числото  $+4.567$  може да се представи изцяло с  
целочислени стойности:  $+4567 \times 10^{-3}$

$+$   $\rightarrow$  знак

$4567$   $\rightarrow$  мантиса

$10^{-3}$   $\rightarrow$  експонента  $-3$  с основа  $10$

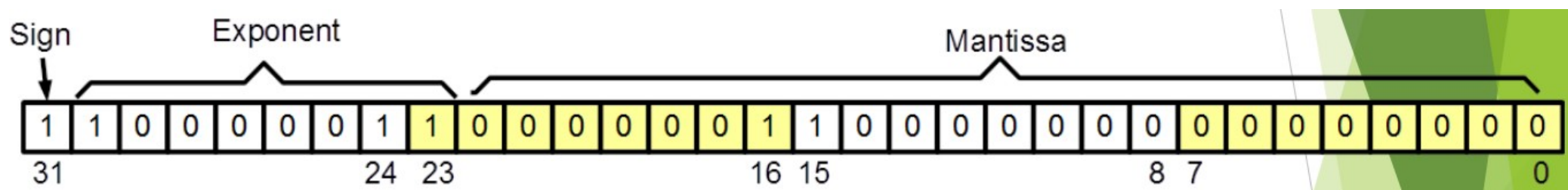
# Представяне на числа в двоичен ВИД



$$float = (-1)^{sign} \times 2^{exponent-127} \times \left( 1 + \sum_{n=1}^{23} bit_{23-n} \times 2^{-n} \right)$$

# Представяне на числа в двоичен ВИД

*Пример* – намерете десетичния еквивалент на числото, представено с IEEE754-1985 дробно число:



\*Знак:  $(-1)^1 = -1$

\*Експонента:  $10000011_{(2)} = 131_{(10)} \rightarrow 2^{131-127} = 2^4$

\*Мантиса:  $1 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + 0 \times 2^{-6} + 1 \times 2^{-7} + 1 \times 2^{-8} + \dots = 1 + 1/2^7 + 1/2^8 = 1.01171875$

\*Резултат:  $-1 \times 2^4 \times 1.01171875 = -16.1875$

\*Или използвайте онлайн конвертора [2] :-)

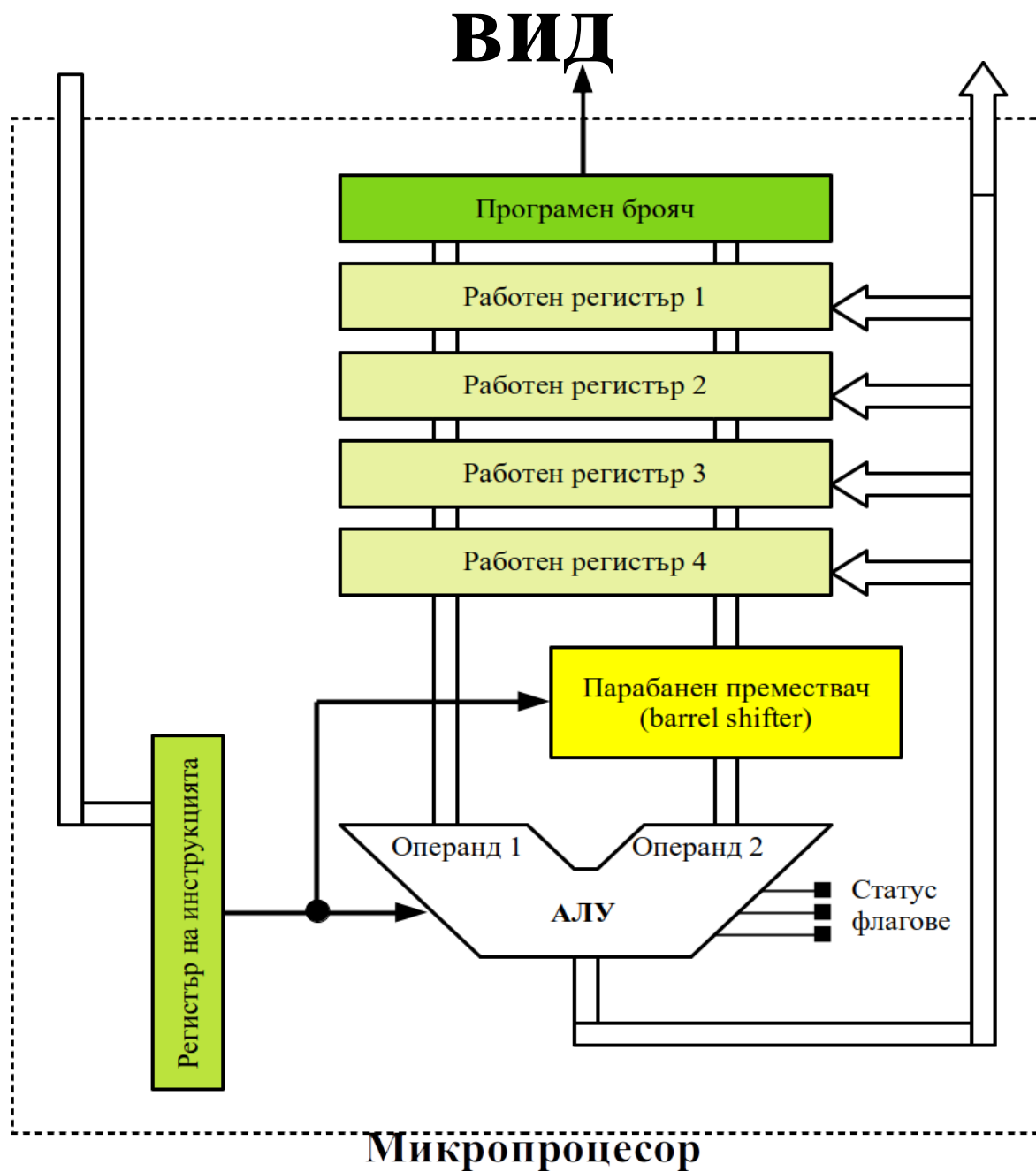
# Представяне на числа в двоичен ВИД

**Барабанен премествач (barrel shifter)** – комбинационна логическа схема, която може да премести (shift) и ротира (rotate) логически битове наляво или дясно с произволен брой позиции за един такт. В съвременните  $\mu$ PU се свързва към поне един вход на АЛУ и специален вид адресация на инструкциите го активира.

Така вместо  $\mu$ PU да изпълнява отделна инструкция по преместване и след това инструкция, която да ползва резултата, може директно да се използва инструкция с адресация с преместване. Последното ще отнеме по-малко тактове от стандартния подход.



# Представяне на числа в двоичен



# Представяне на числа в двоичен ВИД

*Пример* – всички ARM Cortex v7 микропроцесори имат барабанен премествач.

Премести 2 позиции наляво числото в r10, събери го с r9, запиши резултата в r9:

```
add r9, r9, r10, LSL2
```

Барабанен премествач има и на адресния генератор. Премести битовете на отместването (offset) в r10 с 4 позиции надясно (scaled offset), събери полученото число с числото в r9, иди на получения адрес и каквото има там го зареди в r11:

```
mov r11, [r9, r10, LSR4]
```

# Модули за числа с плаваща запетая (FPU)

**Модул за числа с плаваща запетая (Floating Point Unit, FPU)** – функционален блок от  $\mu$ PU, извършващ аритметиката и някои специални операции върху числа с плаваща запетая.

В зависимост от връзката на FPU с  $\mu$ PU, има няколко възможни реализации:

- \*FPU модулът е отделен, външен чип (копроцесор);
- \*FPU модулът е интегриран в скаларен  $\mu$ PU;
- \*FPU модулът е интегриран в суперскаларен  $\mu$ PU.

# Модули за числа с плаваща запетая (FPU)

\*FPU модулят е отделен, външен чип (копроцесор) – исторически работата с числа с плаваща запетая се е прехвърляла на чип извън микропроцесора. Този чип е получил названието “копроцесор”. Двоичният код на инструкциите за копроцесорът трябва да е различен от останалите инструкции за целочислена обработка. Затова и мнемониката на инструкцията за FPU се различава от тази на  $\mu$ PU:

add r1, r2	fadd s1, s2
sub r4, r5	fsub s4, s5

FPU копроцесорът си има **отделен регистров файл**, затова регистрите от операндите се представят с различни символи от тези обикновените инструкции ( $r1 \leftrightarrow s1$ ,  $r2 \leftrightarrow s2$ , и т.н.)<sup>44/57</sup>

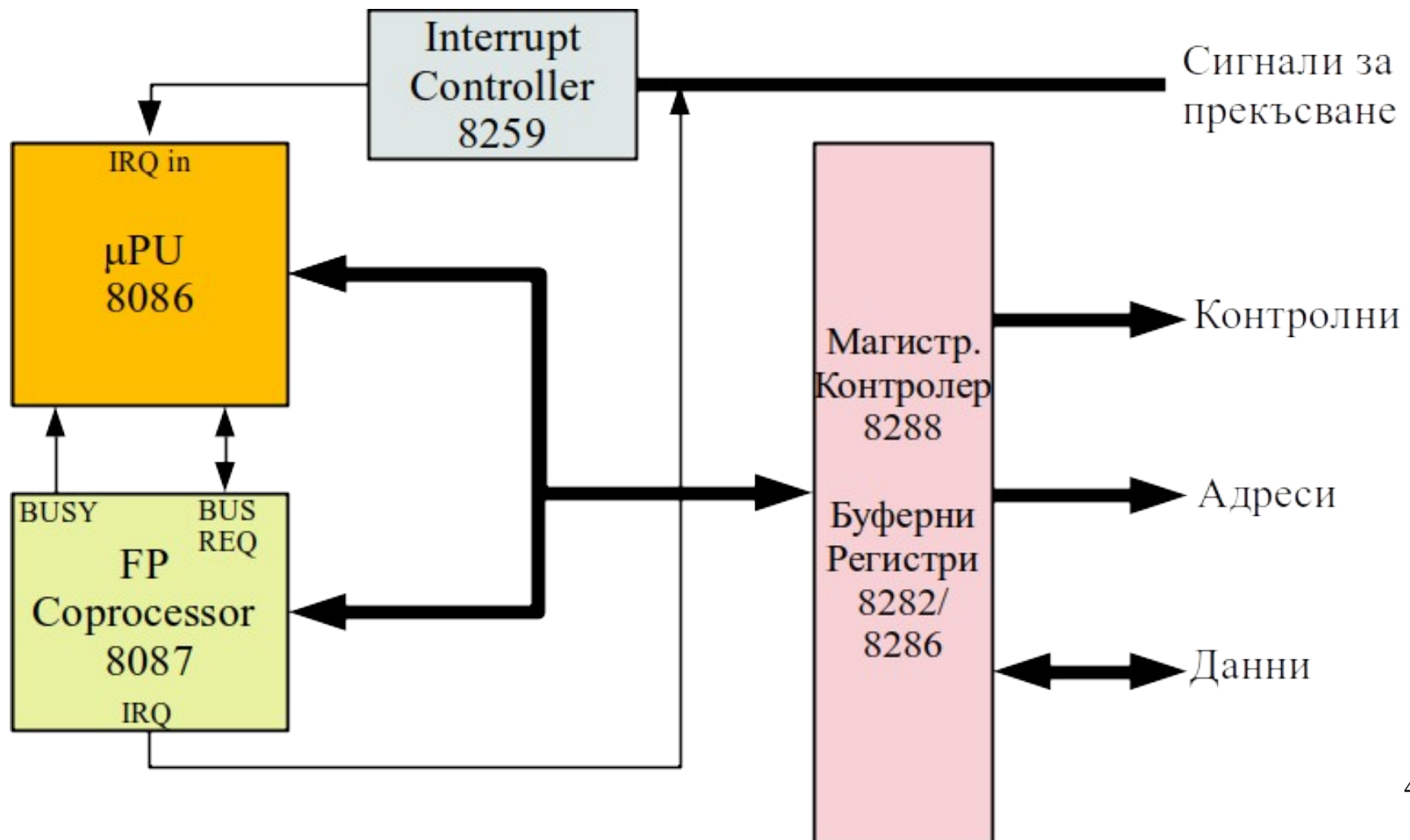
# Модули за числа с плаваща запетая (FPU)

FPU копроцесорите си имат **собствен стек** (dedicated stack) и могат да **генерират прекъсвания**.

FPU се свързват в паралел на магистралата за данни/инструкции на  $\mu$ PU и когато разпознаят КОП на FP инструкция започват да я изпълняват. През това време  $\mu$ PU може да изпълнява други /целочислени/ инструкции.

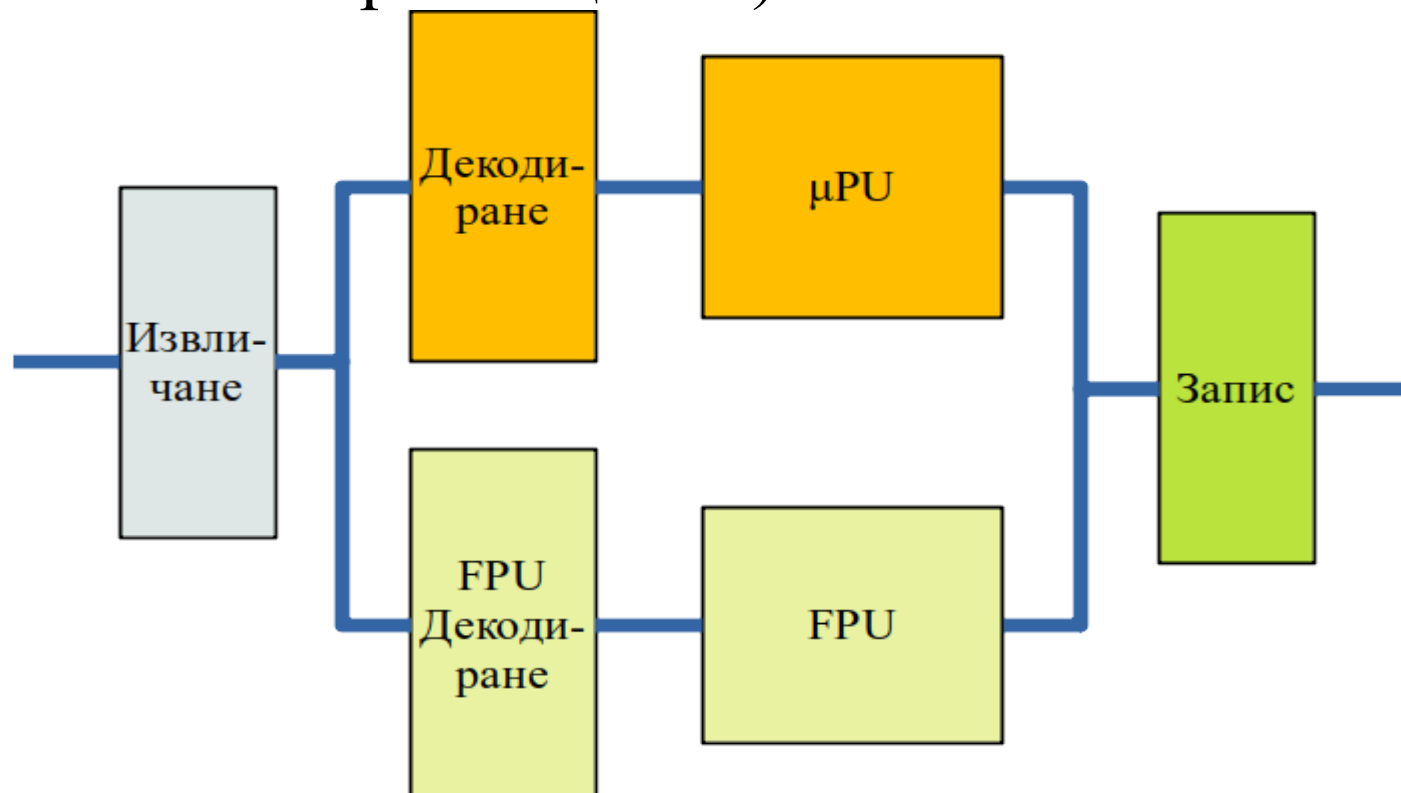
# Модули за числа с плаваща запетая (FPU)

*Пример* – микропроцесор 8086 и копроцесор 8087.



# Модули за числа с плаваща запетая (FPU)

\*FPU модульът е интегриран в скаларен  $\mu$ PU – представлява блок от микропроцесора, който е свързан към модула за извличане на инструкцията. Има си отделен декодер на инструкцията. Когато се разпознае КОП за FP, конвейерът се превключва към FPU. През това време  $\mu$ PU е в неактивен режим (изпълнява празни цикли).



# Модули за числа с плаваща запетая (FPU)

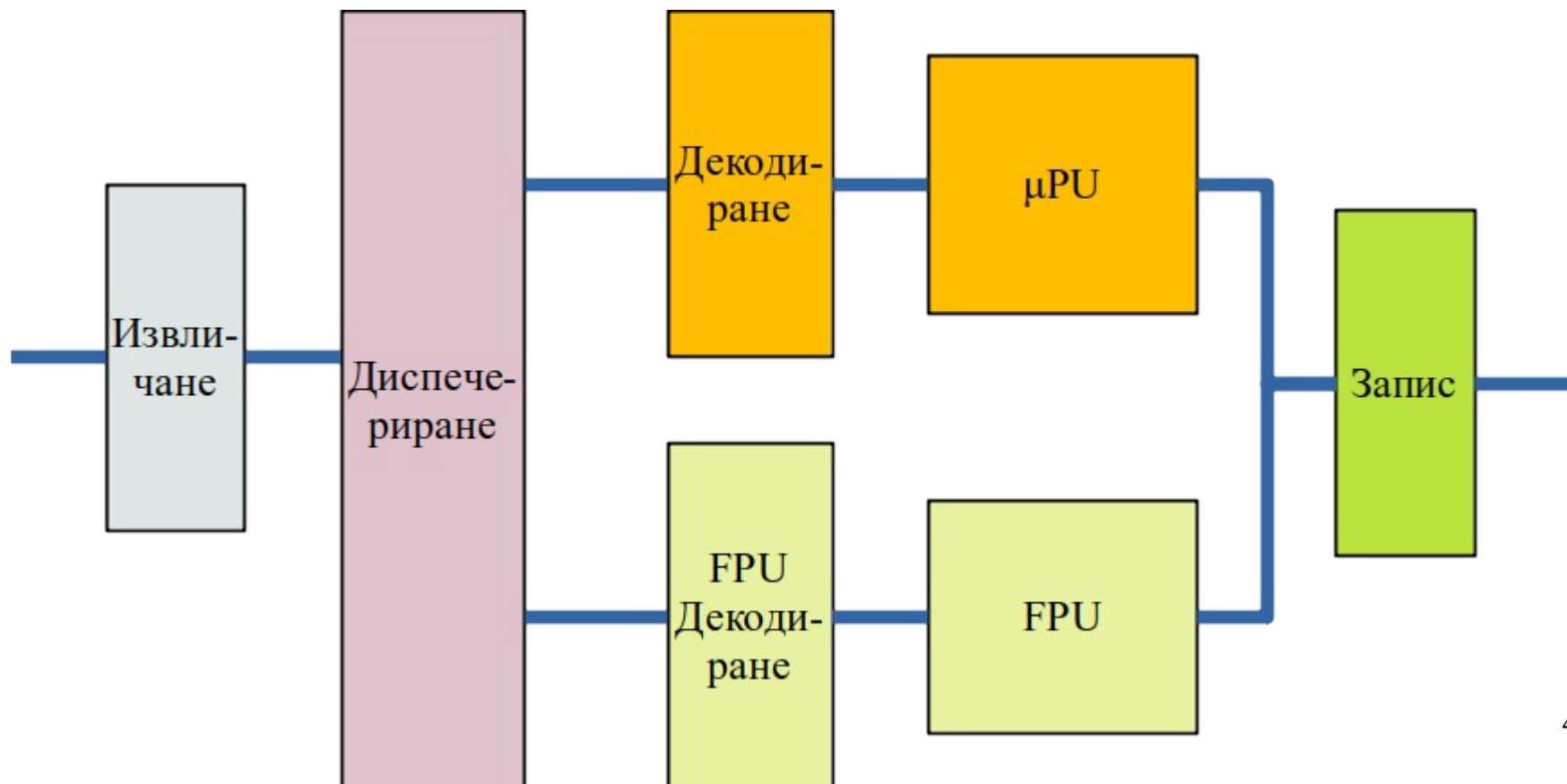
*Пример* – ARM Cortex-M4F е скаларен процесор с FPU [5]. Докато се изпълняват FPU инструкции, Cortex-M4F чака.

Операция	Тактове
+ / -	1
Делене	14
Умножение	1
MAC	3
MAC със закръгляне (fused MAC)	3
Корен квадратен	14



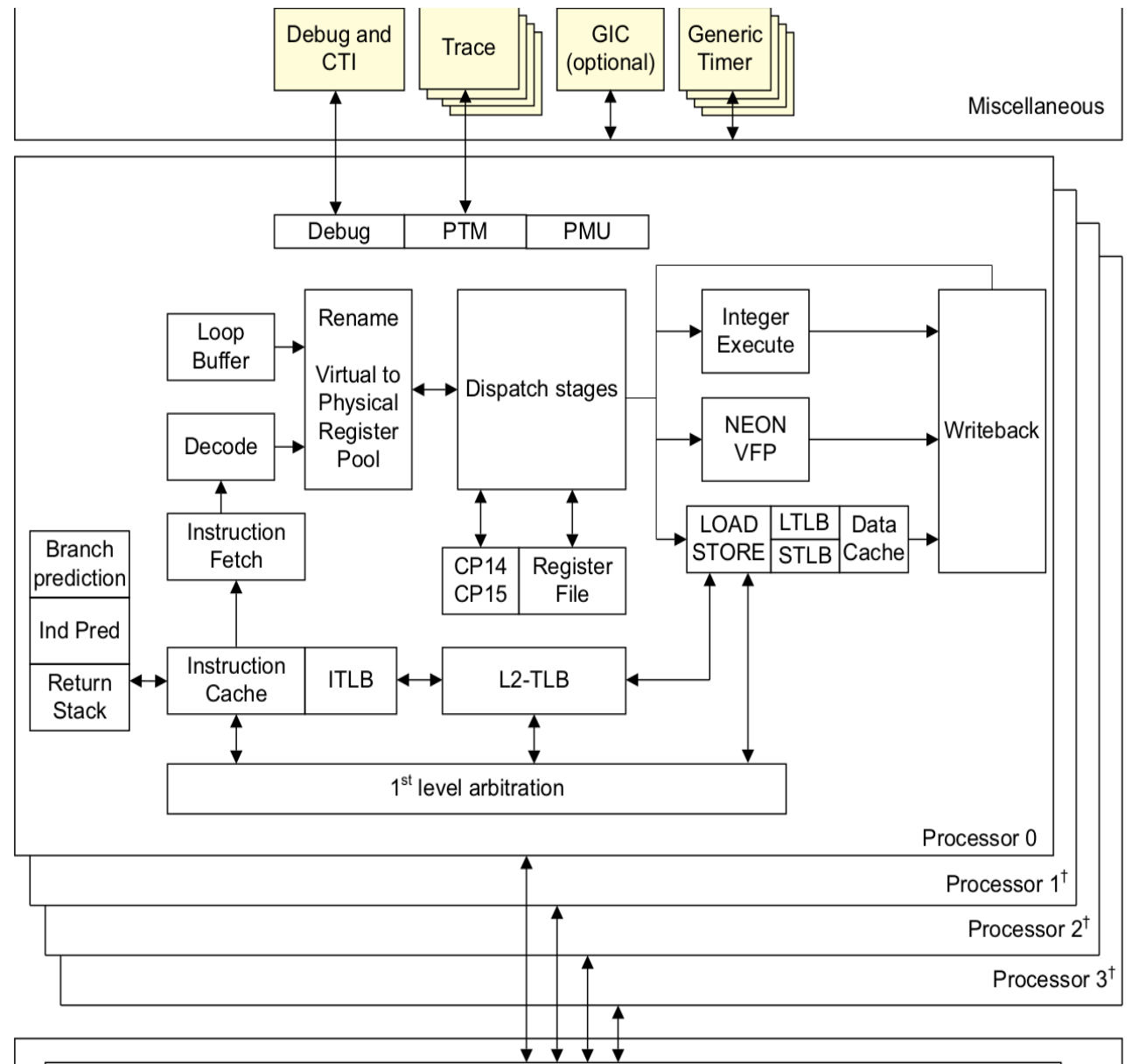
# Модули за числа с плаваща запетая (FPU)

\*FPU модульът е интегриран в суперскаларен  $\mu$ PU – тогава диспечерът на инструкцията може да пусне и микропроцесора, и FPU модула да работят в паралел.



# Модули за числа с плаваща запетая (FPU)

*Пример* – ARM Cortex-A15 е суперскаларен процесор [4] с out-of-order изпълнение на инструкцията. FPU модулът му се казва NEON (векторен FPU). Диспечерът на инструкцията позволява истинския паралелизъм.



# Модули за числа с плаваща запетая (FPU)

По подразбиране програмите за  $\mu$ PU се компилират със софтуерни библиотеки за FPU аритметиката.

Затова, ако трябва да се използва хардуерния модул FPU, **трябва да се направят две неща:**

\*да се укаже на **компилятора чрез аргумент**, че  $\mu$ PU има FPU (може една и съща архитектура да е имплементирана с или без FPU, например ARM Cortex-M4 и ARM Cortex-M4F);

\*във **фърмуера трябва да се включи FPU модула** преди да се използват float и double променливи.

# Модули за числа с плаваща запетая (FPU)

*Пример* – ARM Cortex-M7 имат FPU с двойна прецизност (64-bit double).

На компилатора се указва FPU:

```
arm-none-eabi-gcc -mcpu=cortex-m7 -mthumb -nostartfiles  
-specs=nosys.specs -mfloat-abi=hard -mfpu=fpv5-d16 -Tscript.ld  
./debug/main.o -o ./debug/main.axf
```

След това във фърмуера (в start-up кода):

```
#define SCB_CPACR_CP10_CP11_EN          0xF00000  
volatile uint32_t *scb_cpacr = (volatile uint32_t *)0xE000ED88;  
  
*scb_cpacr |= SCB_CPACR_CP10_CP11_EN; //Enable ARM's FPU
```

# Модули за числа с плаваща запетая (FPU)

След като се пусне FPU, може да се декларират променливи от вида **float** и **double**, и да се извършват операции с тях.

Оператори, които не съществуват в C (напр. корен квадратен) трябва да се заменят с извикване на **функция** от `math` или друга подобна библиотека.

# Модули за числа с плаваща запетая (FPU)

*Пример* – добавяне на стандартната math библиотека в C става чрез указване на линкера със специален аргумент:

```
arm-none-eabi-gcc -mcpu=cortex-m7 -nostartfiles --specs=nosys.specs -  
mfloat-abi=hard -mfpu=fpv5-d16 -Tscript.ld main.o -o main.axf -lm
```

и след това във фърмуера:

```
#include <math.h>
```

# Модули за числа с плаваща запетая (FPU)

Алтернатива на стандартните библиотеки са **вътрешните функции** (intrinsic functions) и **внедрен Асемблер** (inline Assembler), които обаче изискват добро познаване на FPU инструкциите.

*Пример* – използване на внедрен Асемблер е показано на следващия слайд.

# Модули за числа с плаваща запетая (FPU)

```
double my_fpu_array_1[10] =  
{4.00, 16.00, 3.504, 350.768, 1.14, 1256.13, 4096.00, 56.1212, 98.111,  
311.256};
```

```
double my_sqrt[10];
```

```
volatile int i;
```

```
*scb_cpacr |= SCB_CPACR_CP10_CP11_EN; //Enable ARM's FPU
```

```
for(i = 0; i < 10; i++){  
    asm volatile("vsqrt.f64 %P0,%P1"  
        : "=w" (my_sqrt[i])  
        : "w" (my_fpu_array_1[i]) );  
}
```



# Литература

- [1] Jason Albanus, “Coding Schemes Used with Data Converters”, SBAA042A, Texas Instruments, 2015.
- [2] <https://www.h-schmidt.net/FloatConverter/IEEE754.html>
- [3] “MSP430 IQmathLib Users Guide”, v1.10.00.05, 2015.
- [4] “ARM Cortex-A15 Technical Reference Manual”, r4p0, ARM Ltd, 2013.
- [5] M. Trevor, „The Designer’s Guide to the Cortex-M Processor Family – A Tutorial Approach“, Elsevier, 2013.