

# 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!***



# 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

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 :



$$\begin{array}{r} + \\ \hline \end{array}$$

1

$$0000\ 0100$$

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 (резолуцията) 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, представящо 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:

$$Q12(3.625) = 14848 * 2^{-12}$$

откъдето се вижда, че хардуерът предварително трябва да знае за  $2^{-12}$



# **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

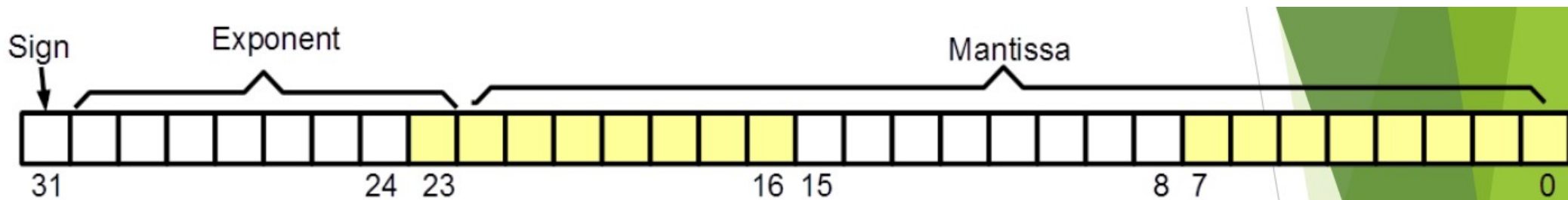
Числото  $+4.567$  може да се представи изцяло с  
целочислени стойности:  $+4567 \times 10^{-3}$

$+$   $\rightarrow$  знак

$4567$   $\rightarrow$  мантиа

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

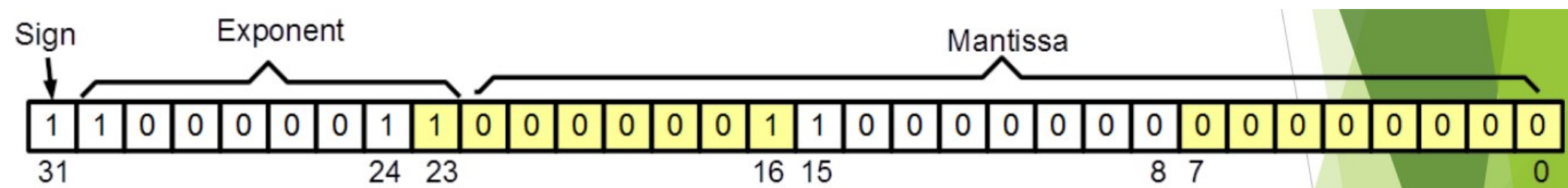
# Представяне на числа в двоичен ВИД



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

# 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] :-)

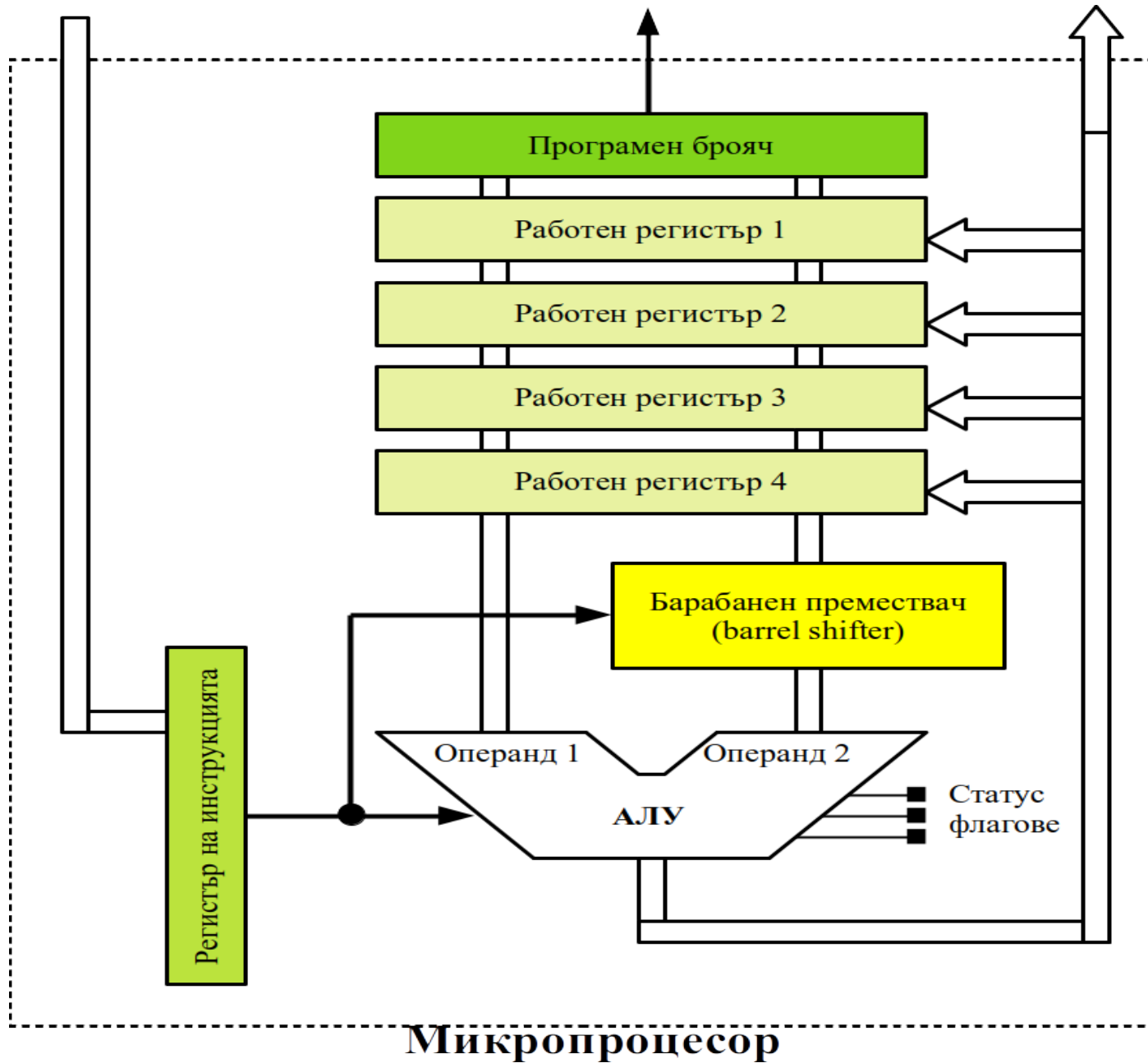
# Барабанен премествач

**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.



# Барабанен премествач



# Барабанен премествач

*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.

*Ако няма FPU, може да се използват софтуерни библиотеки.*

# Модули за числа с плаваща запетая (FPU)

\*FPU модульт 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>44/111</sup>

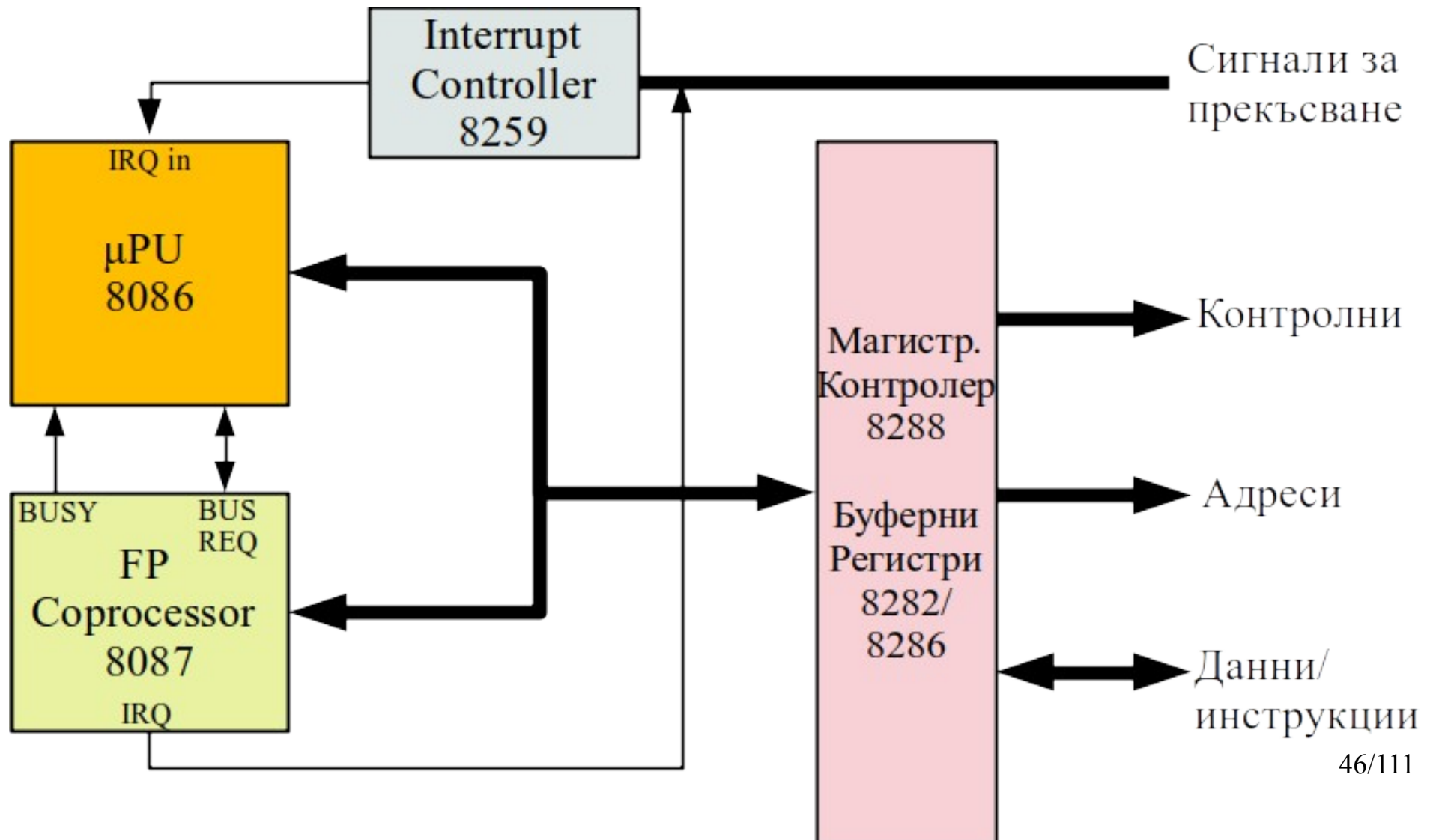
# **Модули за числа с плаваща запетая (FPU)**

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.

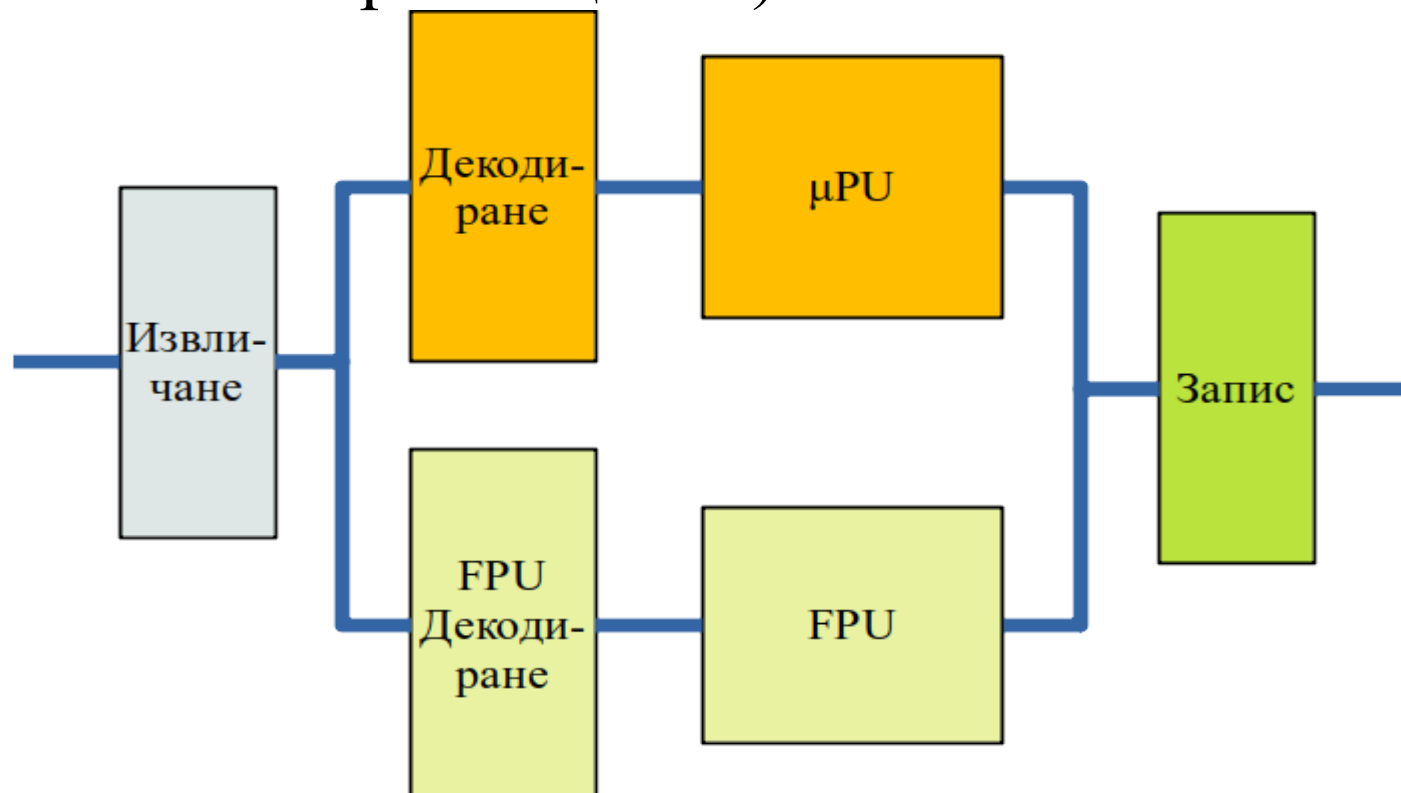
# Модули за числа с плаваща запетая (FPU)

*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)

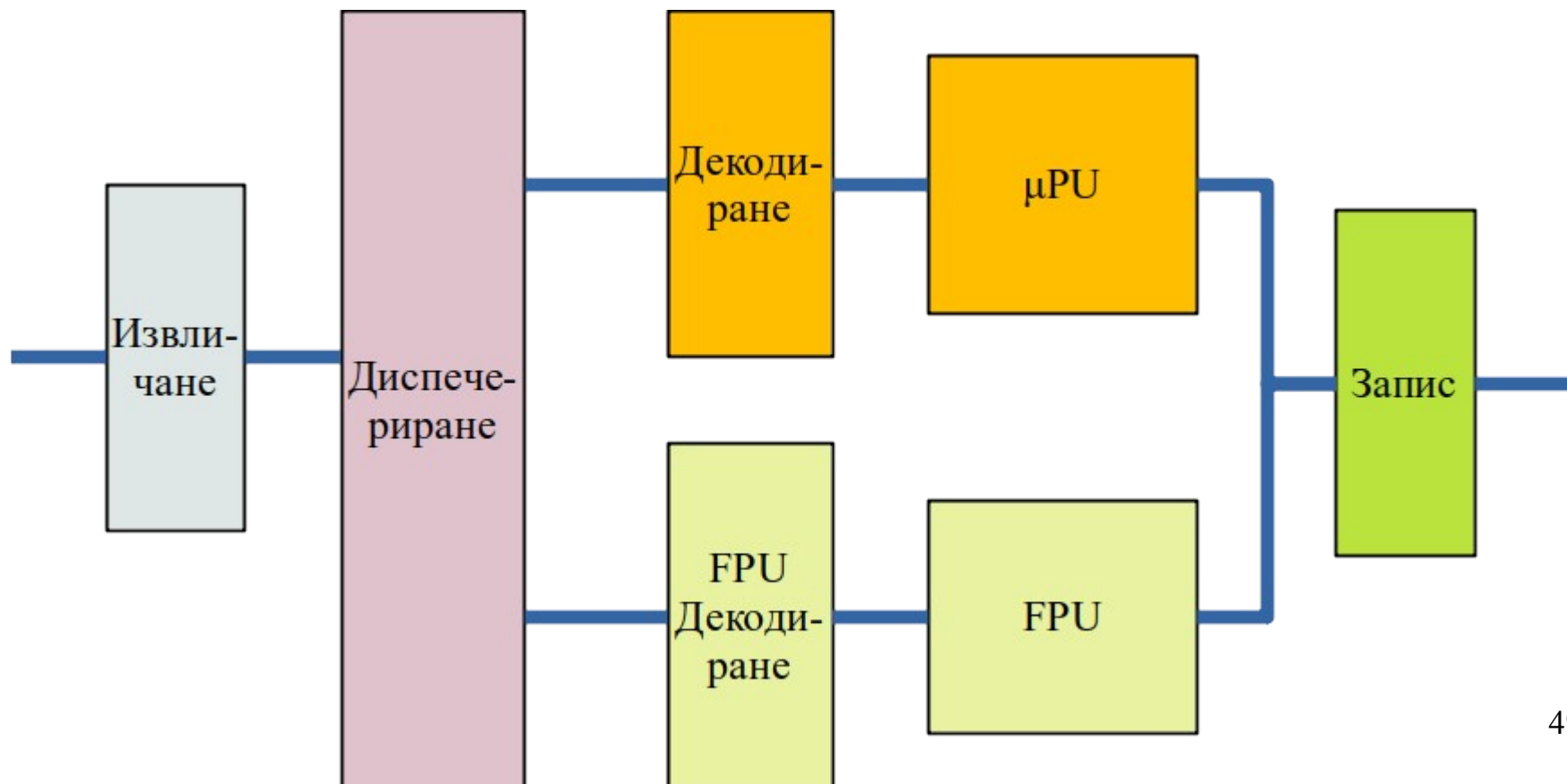
*Пример* – ARM Cortex-M4F е скаларен процесор с FPU [5]. Докато се изпълняват FPU инструкции, Cortex-M4F чака.

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



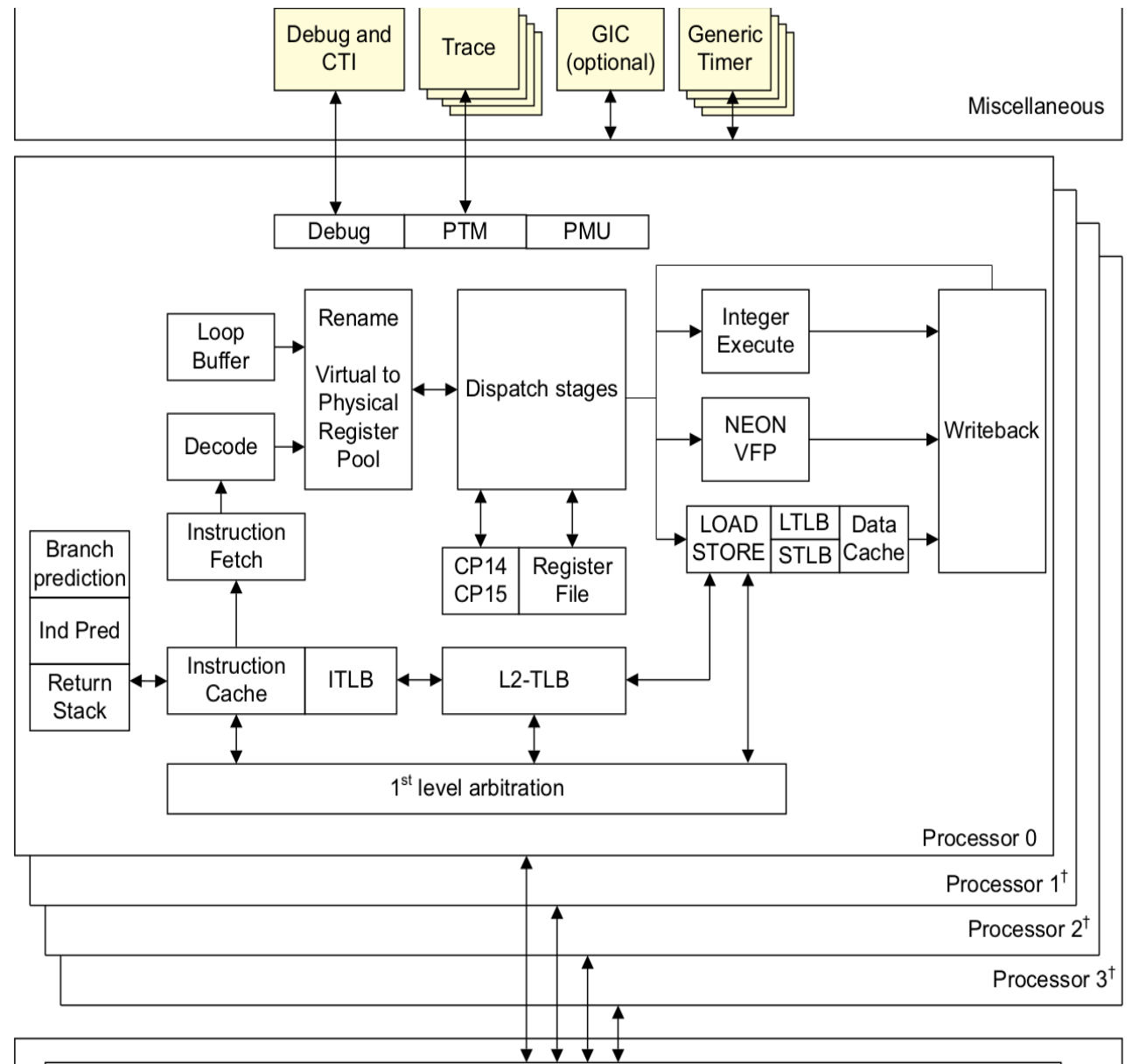
# Модули за числа с плаваща запетая (FPU)

\*FPU модульт 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 декларираат 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]) );  
}
```



# Умножители (MPY)

**Умножителен модул (MultiPLY, MPY)** – модул, реализиращ умножението на целочислени стойности в един микропроцесор.

В зависимост от връзката на MPY с  $\mu$ PU, има две възможни реализации:

- \*MPY е модул, видим в адресното поле на  $\mu$ PU (memory mapped multiplier);
- \*MPY е част от микропроцесорното ядро.

*Ако няма MPY, може да се използват софтуерни библиотеки [7].*

## Умножители (MPY)

\*MPY е модул, видим в адресното поле на  $\mu$ PU – използва се в случаите, когато  $\mu$ PU не поддържа инструкции за умножение.

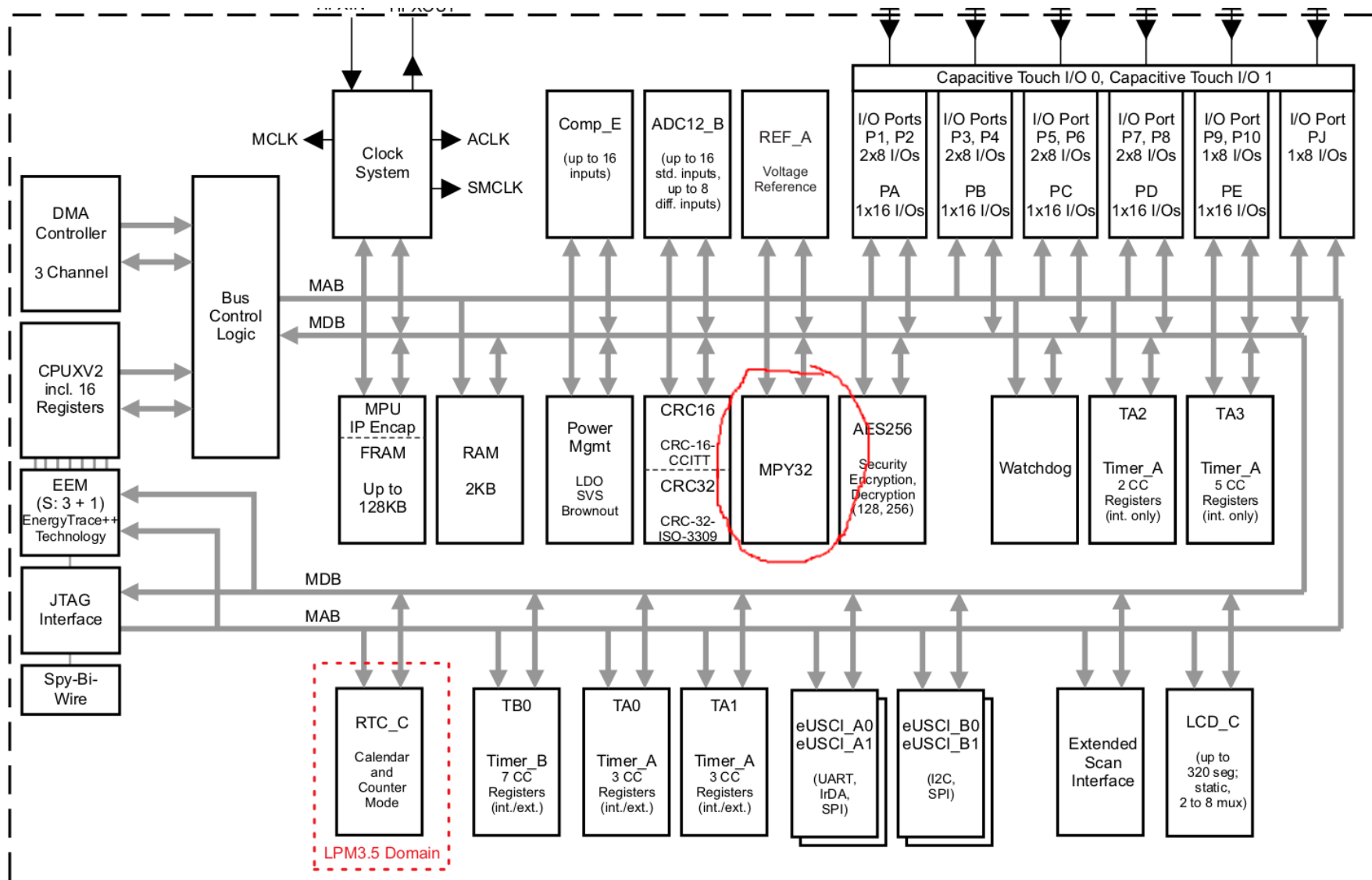
Тогава MPY модула се добавя като периферен модул (както е таймера, GPIO, АЦП, компаратора и т.н.) и може да се оприличи на хардуерен ускорител за съответните операции.

Той има регистри за входни данни и регистри за изходни данни.

$\mu$ PU записва в него числата, които трябва да се умножават, изчаква няколко такта и прочита резултата.

# Умножители (MPY)

Пример – MSP430FR6989 има умножител, ИЗВЪН микропроцесорното ядро.



# Умножители (MPY)

Поддържаните операции са:

- \*умножение на 8/16/24/32-битови числа с и без знак
- \*умножение с натрупване на 8/16/24/32-битови числа с и без знак
- \*умножение с насищане
- \*числа с фиксирана запетая**

Входните числа може да са до 32-бита (записват се в 2 отделни 16-битови регистъра).

Резултатът е 64-битов (записва се в 4 отделни 16-битови регистъра).

Умножението започва, веднага щом се запише второто число в съответния входен регистър.

# Умножители (MPY)

**Table 5-1. Result Availability (MPYFRAC = 0, MPYSAT = 0)**

Operation (OP1 × OP2)	Result Ready in MCLK Cycles					After
	RES0	RES1	RES2	RES3	MPYC Bit	
8/16 × 8/16	3	3	4	4	3	OP2 written
24/32 × 8/16	3	5	6	7	7	OP2 written
8/16 × 24/32	3	5	6	7	7	OP2L written
	N/A	3	4	4	4	OP2H written
24/32 × 24/32	3	8	10	11	11	OP2L written
	N/A	3	5	6	6	OP2H written

# Умножители (MPY)

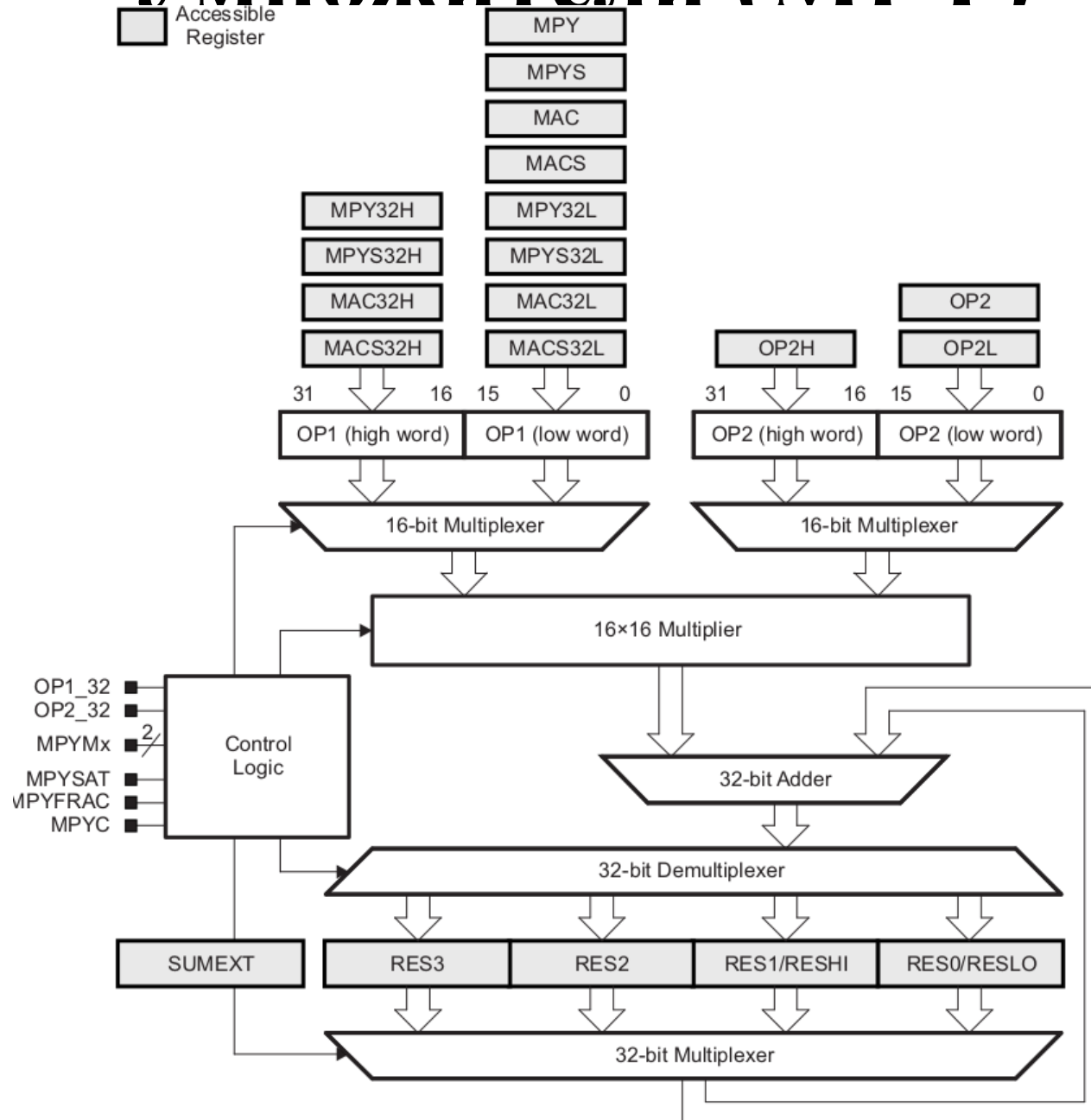


Figure 5-1. MPY32 Block Diagram

# Умножители (MPY)

Когато MPY модулът е външен за  $\mu$ PU, трябва да се **внимава с прекъсванията** и работата с този модул [6]. Ако се запише първото число и, преди да се запише второто число, настъпи прекъсване, и ако в прекъсването се използва умножителя, то конфигурацията на MPY ще се загуби и резултата ще е непредвидим.

*Пример* – MPY се приготвя за умножение на числа с фиксирана запетая (има един бит в контролния регистър MPY32CTL0), а в прекъсването се преконфигурира (отново с MPY32CTL0) за целочислени стойности. При връщането от IRQ, MPY ще вижда входните си числа като цели, а не с фиксирана запетая и операцията ще е грешна.

# Умножители (MPY)

*Пример* - контролен регистър MPY32CTL0 на MSP430FR6989. Ако дори един от битовете 0 ÷ 7 се промени в IRQ, умножението в основната програма ще е грешно.

**Table 5-9. MPY32CTL0 Register Description**

Bit	Field	Type	Reset	Description
15-10	Reserved	R	0h	Reserved. Always reads as 0.
9	MPYDLY32	RW	0h	Delayed write mode 0b = Writes are delayed until 64-bit result (RES0 to RES3) is available. 1b = Writes are delayed until 32-bit result (RES0 to RES1) is available.
8	MPYDLYWR TEN	RW	0h	Delayed write enable All writes to any MPY32 register are delayed until the 64-bit (MPYDLY32 = 0) or 32-bit (MPYDLY32 = 1) result is ready. 0b = Writes are not delayed. 1b = Writes are delayed.
7	MPYOP2_32	RW	0h	Multiplier bit width of operand 2 0b = 16 bits 1b = 32 bits
6	MPYOP1_32	RW	0h	Multiplier bit width of operand 1 0b = 16 bits 1b = 32 bits
5-4	MPYMx	RW	0h	Multiplier mode 00b = MPY – Multiply 01b = MPYS – Signed multiply 10b = MAC – Multiply accumulate 11b = MACS – Signed multiply accumulate
3	MPYSAT	RW	0h	Saturation mode 0b = Saturation mode disabled 1b = Saturation mode enabled
2	MPYFRAC	RW	0h	Fractional mode 0b = Fractional mode disabled 1b = Fractional mode enabled
1	Reserved	RW	0h	Reserved. Always reads as 0.
0	MPYC	RW	0h	Carry of the multiplier. It can be considered as 33rd or 65th bit of the result if fractional or saturation mode is not selected, because the MPYC bit does not change when switching to saturation or fractional mode. It is used to restore the SUMEXT content in MAC mode. 0b = No carry for result 1b = Result has a carry



# Умножители (MPY)

Затова този проблем се решава като временно се забраняват всички прекъсвания с псевдоинструкцията DINT. След завършване на операцията, инструкцията EINT ги разрешава отново.

Изчакването на резултата може да стане:

- \*с NOP инструкция, ако се използват инструкции с @R5 или @R5+ адресации за четене на резултата

- \*директно да се прочете, ако се използват инструкции с X(R5) или &MEM за четене на резултата

# Умножители (MPY)

От таблицата с тактовете – 32-битови резултати с цели числа отнемат 3 такта. За закъснение се използва 1 NOP(=mov r5, r5) инструкцията и първите две фази на инструкцията за достъп MOV, когато тя използва индиректна автоинкрементираща адресация за операнд-източник и абсолютна адресация за операнд-приемник (mov @r5+, &MEM0).

```
mov    #RESL0, r5
dint
nop
mov    r6, &MPY
mov    r7, &OP2
nop
mov    @r5+, &MEM0

mov    @r5, &MEM1
eint
```

;Изчакай 1 такт, заради изходния буфер  
;Стартира се умножението  
;Изчакай 1 такт (изпълнява се още при извличането)  
;Отнема 4 такта - 1 такт – извличане  
;1 такт – декодиране (MPY е готово)  
;1 такт – изпълнение (чете се регистър RESL0)  
;1 такт – записва се резултата

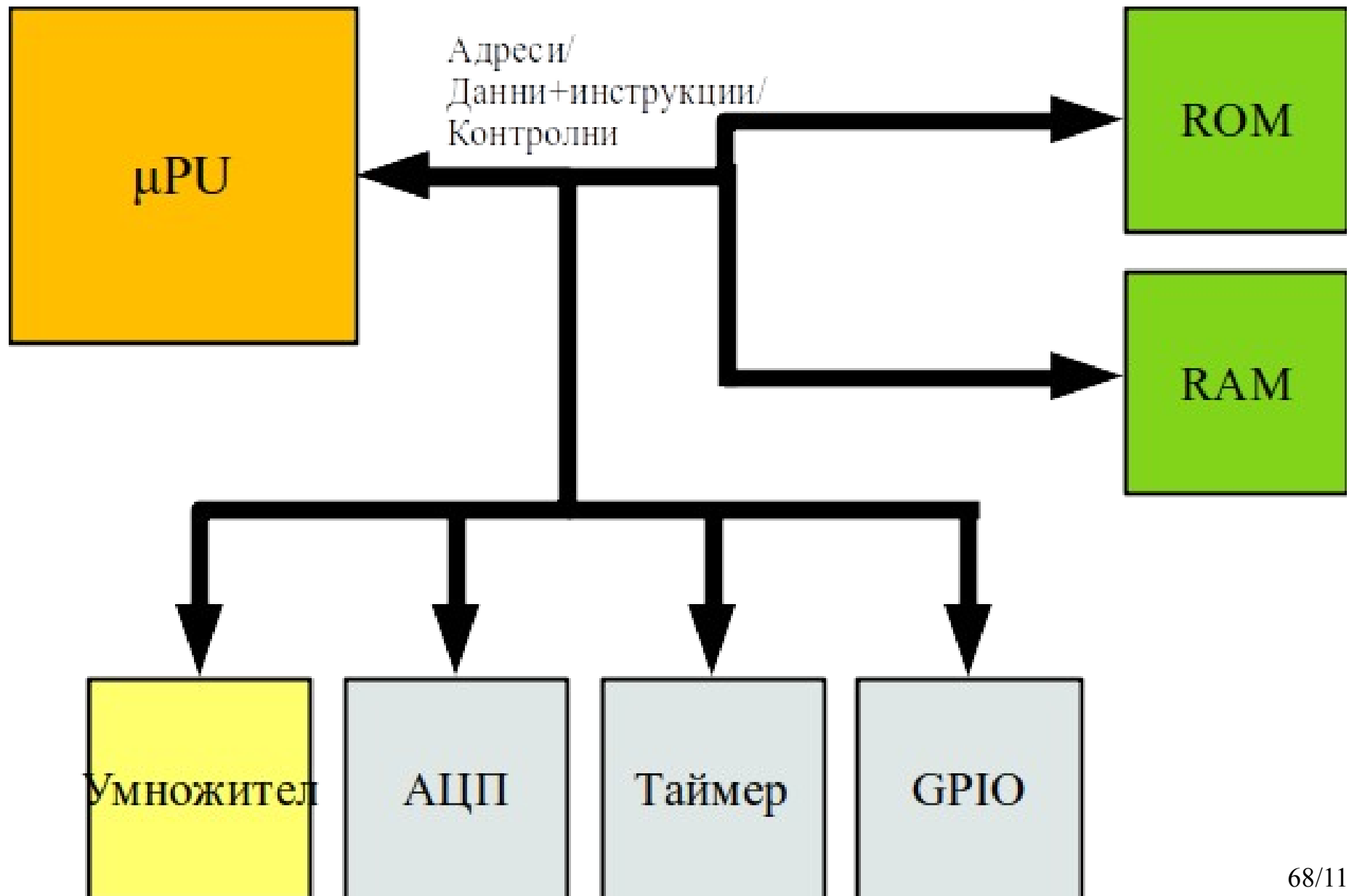
# Умножители (MPY)

По същата логика, ако резултатът се запише с x(R5), &MEM0 операнди, NOP (=mov r5, r5) е излишна, защото тази комбинация от адресации отнема 5 такта.

```
mov    #RESL0, r5
dint
nop                    ;Изчакай 1 такт, заради изходния буфер
mov    r5, &MPY
mov    r6, &OP2        ;Стартира се умножението
mov    0x00(r5), &MEM0 ;Отнема 5 такта - 1 такт — извличане инстр.
                        ;1 такт — извличане офсет
                        ;1 такт — декодиране (MPY е готово)
                        ;1 такт — изпълнение (чете се регистър RESL0)
                        ;1 такт — записва се резултат

mov    0x02(r5), &MEM1
eint
```

# Умножители (MPU)



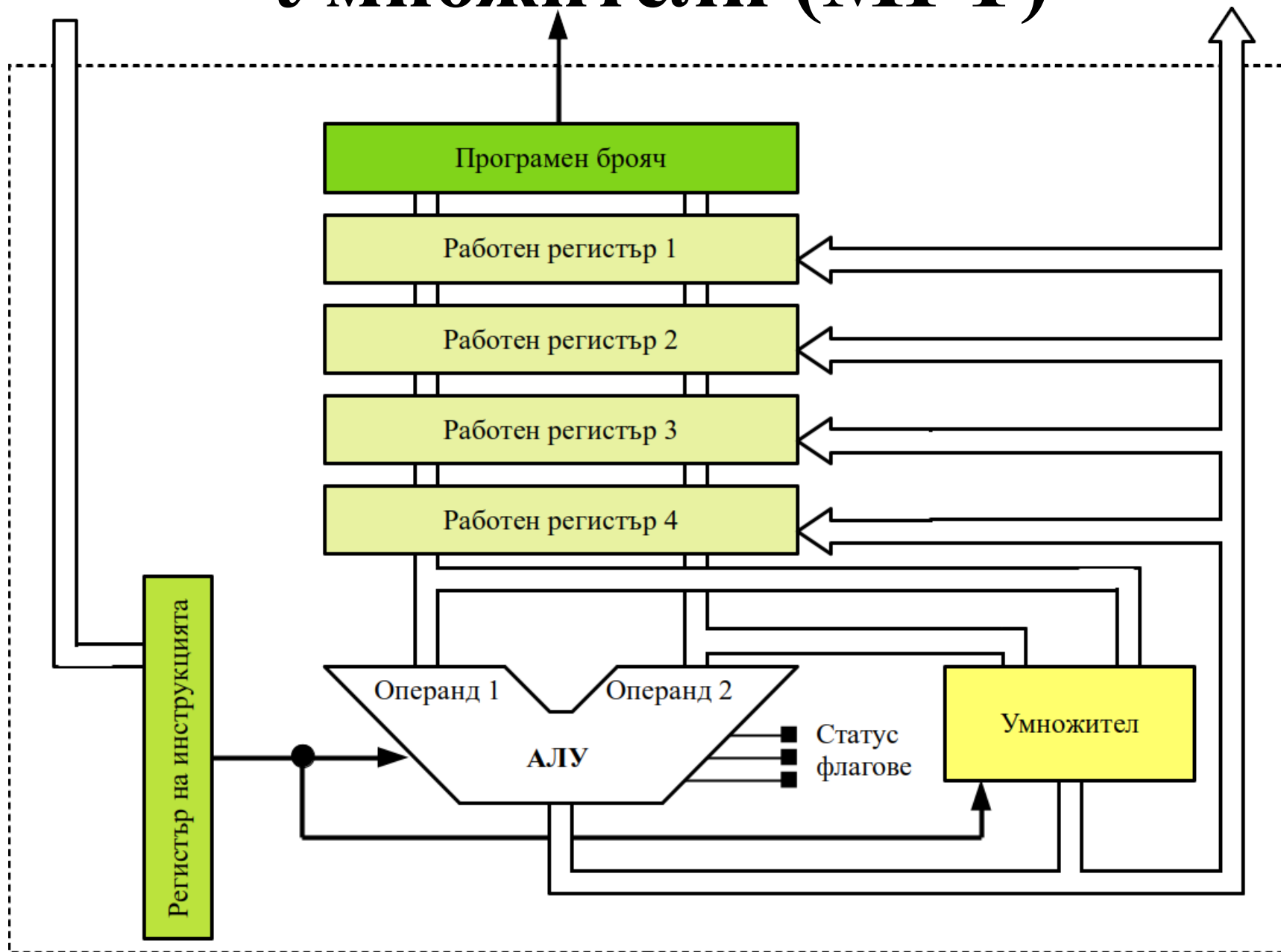
# Умножители (MPY)

\*MPY е част от микропроцесорното ядро – използва се, когато  $\mu$ PU има инструкции за умножение. Тогава той е функционален модул също като АЛУ и FPU.

**Използва регистрите от ядрото на  $\mu$ PU за входни данни и изходни данни.**

**В суперскаларните  $\mu$ PU може да работи в паралел с други функционални модули, благодарение на диспечера.**

# Умножители (MPY)



# Умножители (MPY)

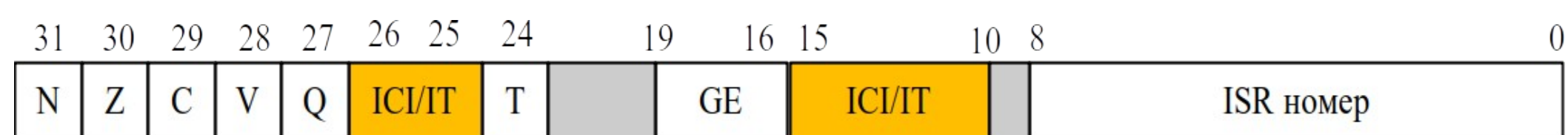
*Пример* – микропроцесорите ARM Cortex-Mx имат инструкция за умножение, отнемаща 1 или 2 такта [5]. Умножителя на ARM има възможност за **делене** на числа. **Умножението може да се използва с прекъсвания.** Входните и изходните числа са 32-битови (т.е. записва се само младшите 32-бита на 64-битовия резултат).

```
mul r9, r9, r10 ;умножи r9 и r10, и запиши резултата в r9
```

# Умножители (MPY)

Статус регистъра на ARM Cortex съдържа 8 бита, в които се запамятава служебна информация при прекъсване, ако ядрото изпълнява ITE, STM, LDM или MUL инструкции [5]. Това позволява при връщане от прекъсването съответните инструкции да продължат нормалната си работа.

Битовете са ICI/IT (Interruptible-Continuable Instruction/If-Then Instruction).



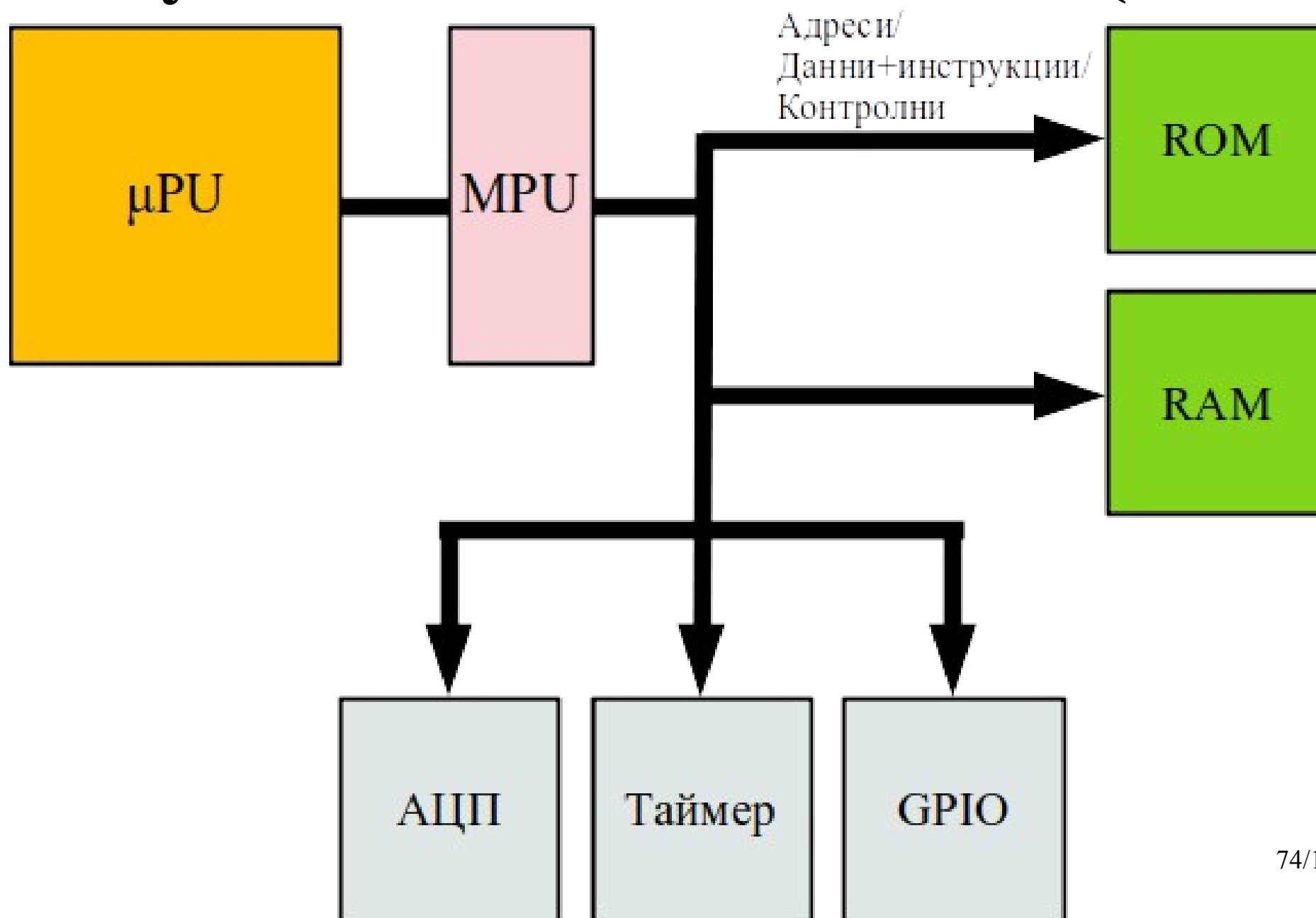


# Модул за защита на паметта (MPU)

Модул за защита на паметта (Memory Protection Unit, MPU) – разделя адресното поле на  $\mu$ PU на региони и подрегиони, и им присвоява права за достъп (четене/запис на данни, изпълнение на инструкции, read/write/execute). Използва се най-често от операционни системи за реално време (RTOS), за да раздели кода на ядрото им от потребителските програми (нишки/задачи). По този начин, ако една нишка сгреши и се опита да пише в системния регион, няма да успее и няма да повлияе на работата на RTOS.

MPU се използва също от библиотеки, които са сертифицирани и не се позволява на потребителския фърмуер да ги преконфигурира (напр. Bluetooth).

# Модул за защита на паметта (MPU)



# Модул за защита на паметта (MPU)

Регистри на MPU са:

- \***контролен** – съдържа битове за включване и конфигуриране на MPU;

- \***избор на регион** – записва се число, избиращо номер на регион, който ще се конфигурира с регистрите за базов адрес и атрибути със следващите инструкции в програмата.

- \***базов адрес** – начален адрес на региона, обикновено регионите имат ограничение за най-малък размер и най-младшите няколко бита от базовия адрес не се използват;

- \***атрибути** – съдържа няколко бита за размер на региона (задават размера по формула, напр.  $2^{[SZ+1]}$ ), както и права за достъп (четене/запис/изпълнение или изцяло забранен), и други.

# **Модул за защита на паметта (MPU)**

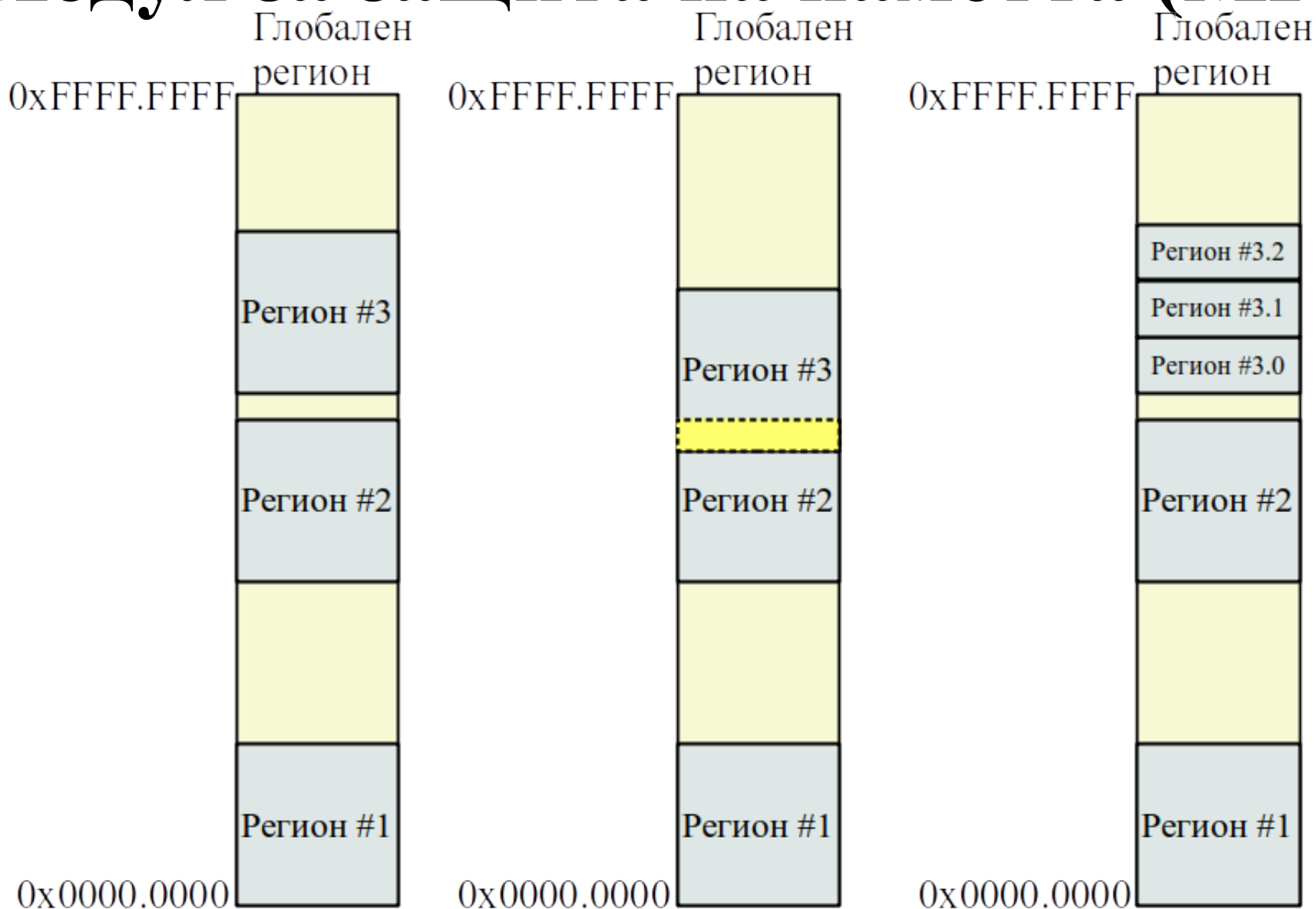
Ако код от един регион се опита да достъпи данни/код от друг регион, без да има съответните права, **достъпът ще се преустанови и ще се генерира прекъсване.**

В прекъсването на MPU се слага код на операционната система и тя трябва да реши как да се справи с проблема.

Възможно е регионите да се припокриват в някои от областите си. Тогава те си наследяват правата за достъп, като регионът с по-висок номер е с по-висок приоритет.

Регионите се разделят на равни подрегиони, които може да бъдат разрешавани/забранявани.

# Модул за защита на паметта (MPU)



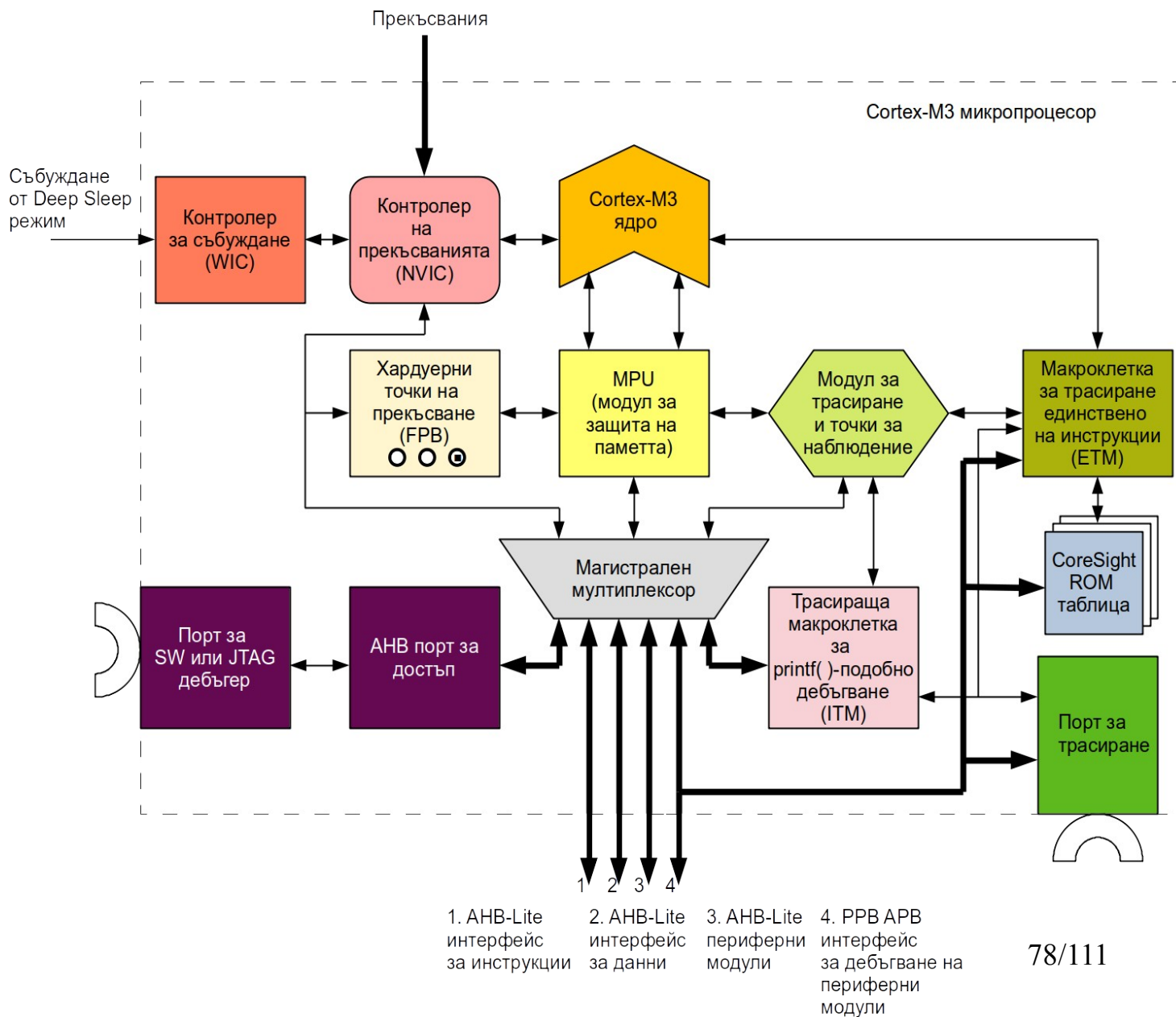
Без  
припокриване

С  
припокриване

С  
подрегиони

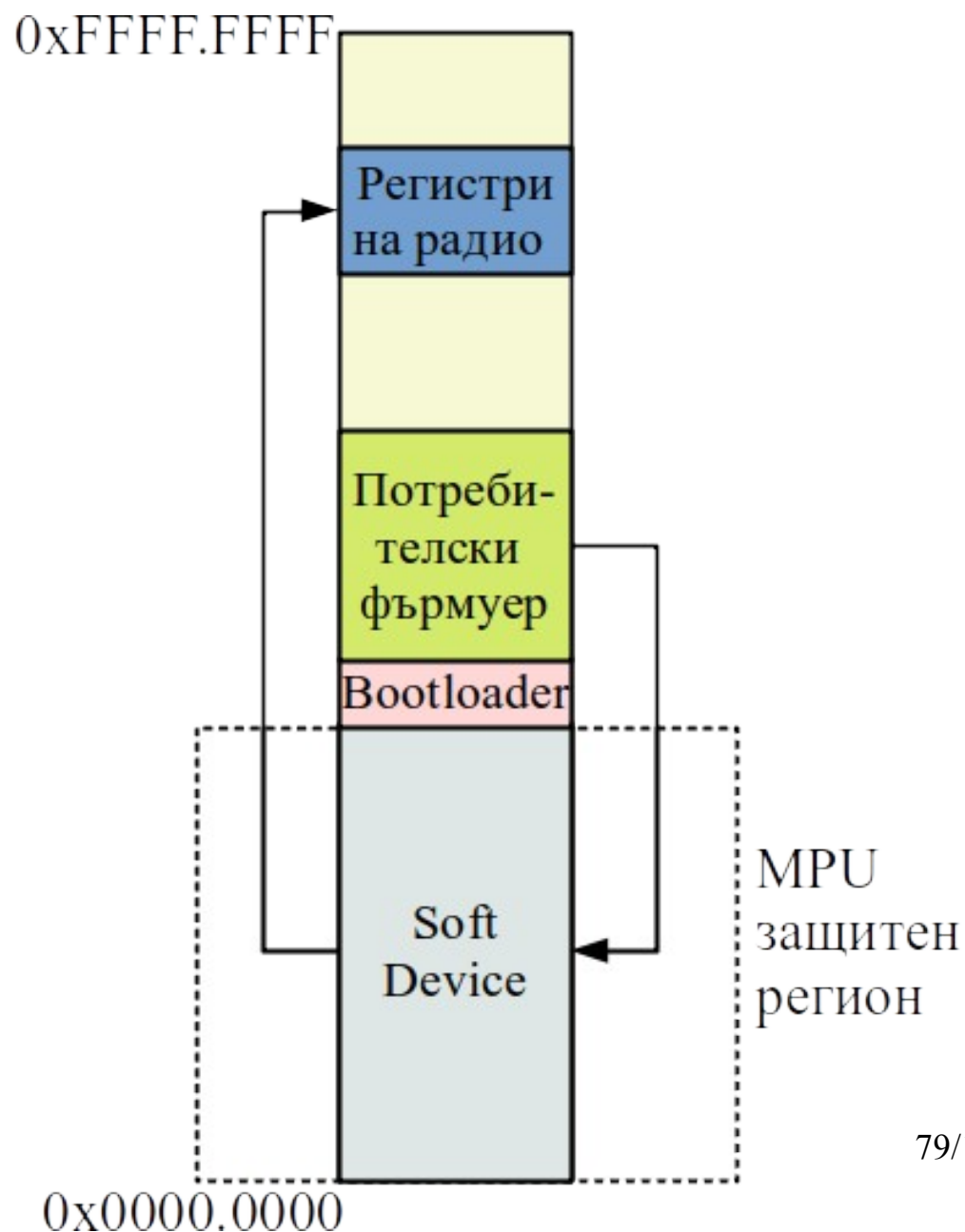
# Модул за защита на паметта (MPU)

*Пример* – ARM Cortex-M3 имат MPU, позволяващо да се раздели адресното поле на 8 региона, всеки от който е разделен на 8 равни подрегиона. Минималният размер е 32 байта, максималният 4 GB.



# Модул за защита на паметта (MPU)

*Пример* – Bluetooth микроконтролерите на Nordic от серията NRF51 и NRF52 използват библиотека (SoftDevice), която работи с радиото и потребителския фърмуер не може да достъпва директно регистрите му.



# Модул за организация на паметта (MMU)

**Модул за организация на паметта (Memory Management Unit, MMU)** – модул, който създава виртуално (реално несъществуващо) адресно поле за  $\mu$ PU и осъществява връзката между виртуални адреси и физически адреси (на регистри, реално съществуващи в системата) [9].

Целта на MMU е подобна на MPU – използва се от OS за контрол на потребителските програми. MMU позволява на OS да контролира кои региони от паметта да са видими за програмата, началните адреси на тези региони и права за достъп до регионите.



# Модул за организация на паметта (MMU)

Преобразуването на виртуални адреси във физически от MMU модул се нарича **транслиране на адреси**.

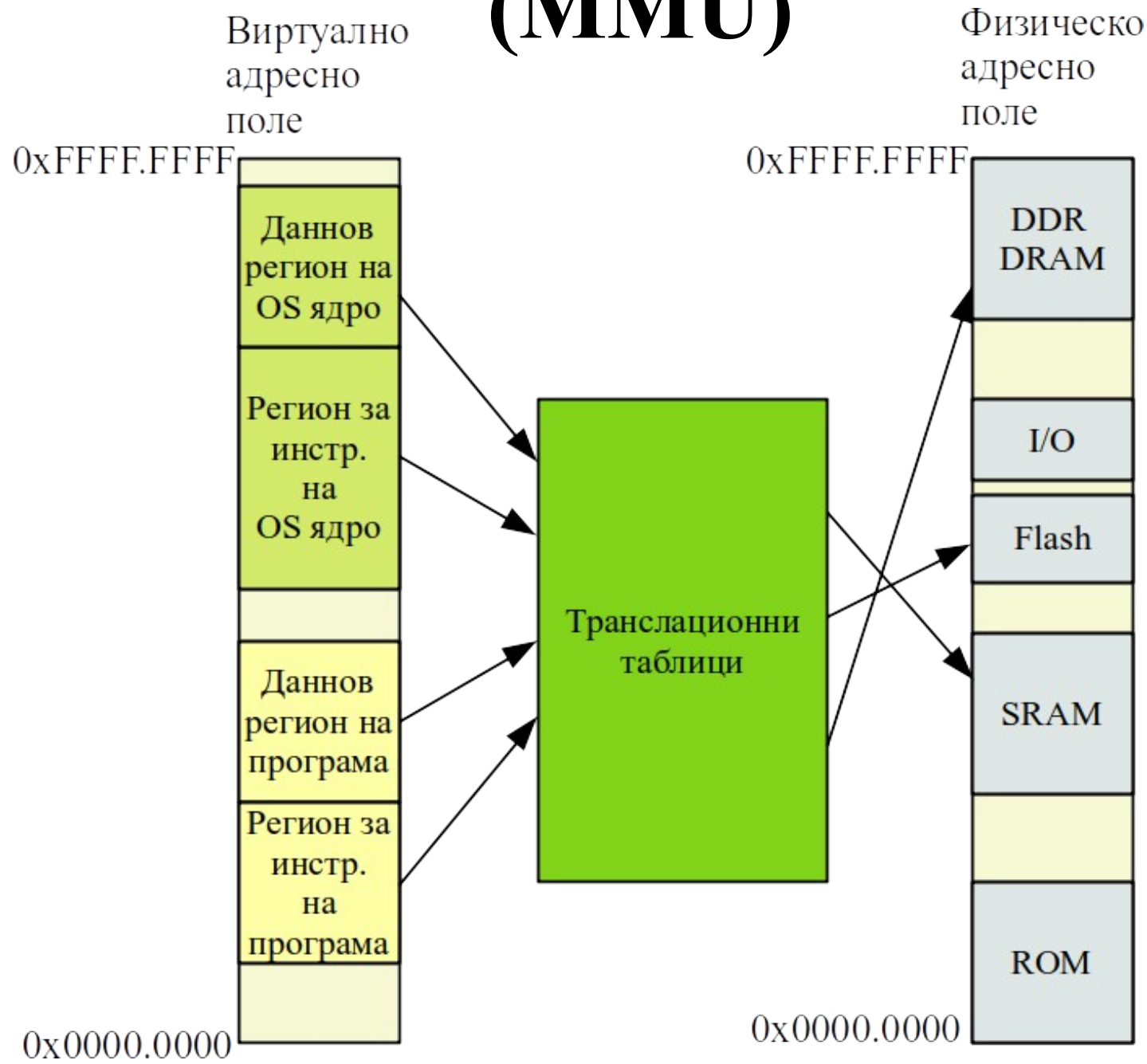
Освен контрол, създаването на виртуално адресно поле позволява и да се направи **абстракция от хардуера**.

*Пример* - няколко фрагментирани малки “парчета” от паметта да се представят като едно единствено адресно поле, което програмата вижда (важно за malloc).

*Пример* – програмистите не трябва да знаят точно на кои адреси се намира паметта, а само че има MMU, което ще направи трансляцията.

*Пример* – ако RAM свърши, може да започне да се използва хард диска като даннова памет.

# Модул за организация на паметта (MMU)



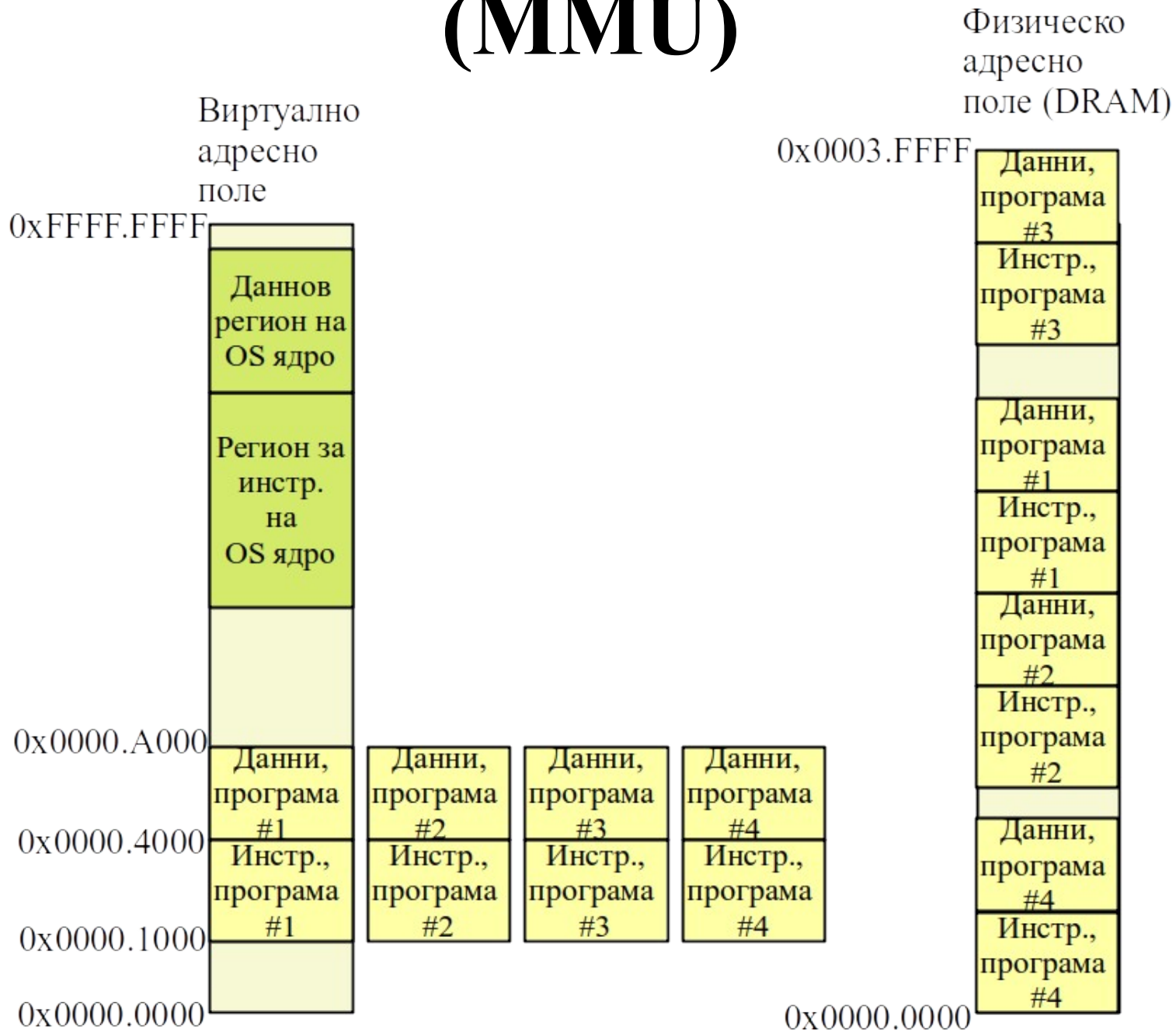
# Модул за организация на паметта (MMU)

При bare-metal фърмуера всички адреси трябва да са физически. Ако дадена програма трябва да се пусне на друга система със същия  $\mu$ PU, ще се наложи тя да се компилира наново, защото паметите ще са на други адреси.

Ако се използва MMU, същата програма трябва само да се копира в системата и тя ще работи без ре-компиляция.

И нещо повече – няколко програми в паралел може да работят на едни и същи адреси (виж ASID).

# Модул за организация на паметта (MMU)



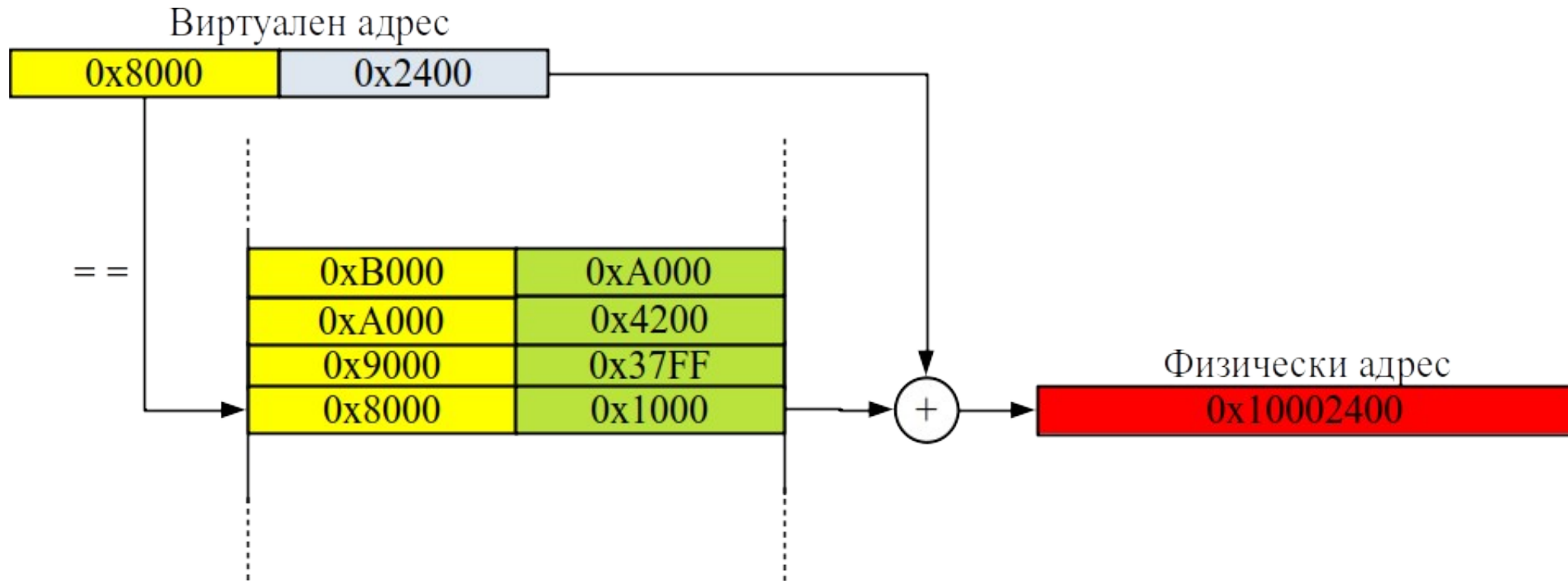
# Модул за организация на паметта (MMU)

Виртуалният адрес се разделя на две части:

**\*номер на страница (page number)** – старшата част от виртуалния адрес, която указва индекс на елемент от таблица на съответствието, съдържаща физическия адрес. Тази таблица се взима или от буфер (TLB), или от основната памет (RAM).

**\*отместване в страницата (page offset)** – отместване от базовия адрес на страницата, където се намират данните/инструкциите. Отместванията не се транслират, т.е. отместването във виртуалния адрес = отместването във физическия адрес.

# Модул за организация на паметта (MMU)



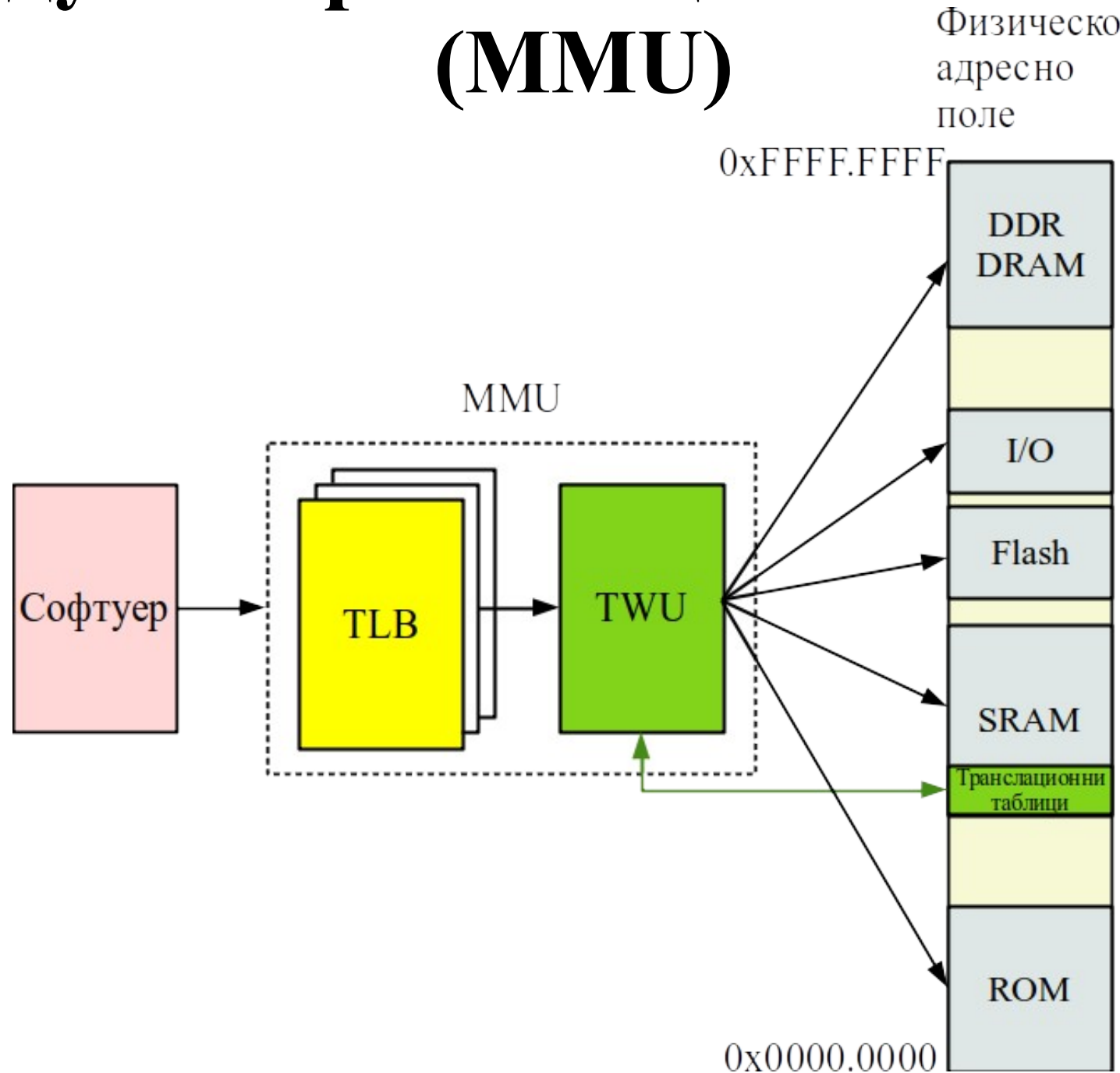
# Модул за организация на паметта (MMU)

MMU модула се състои от два подмодула:

**\*модул за търсене в таблица (Table Walk Unit, TWU)** – хардуерен модул, използван за намиране на съответствието между физически и виртуален адрес от транслационна таблица. Транслационната таблица се съхранява в RAM.

**\*транслационни буфери (Translation Lookaside Buffers, TLB)** – хардуерни буфери, съдържащи данни за скорошно извършени транслации, така че да не се налага отново да се търсят в таблиците. Типично се вграждат по  $16 \div 512$  буфера.

# Модул за организация на паметта (MMU)





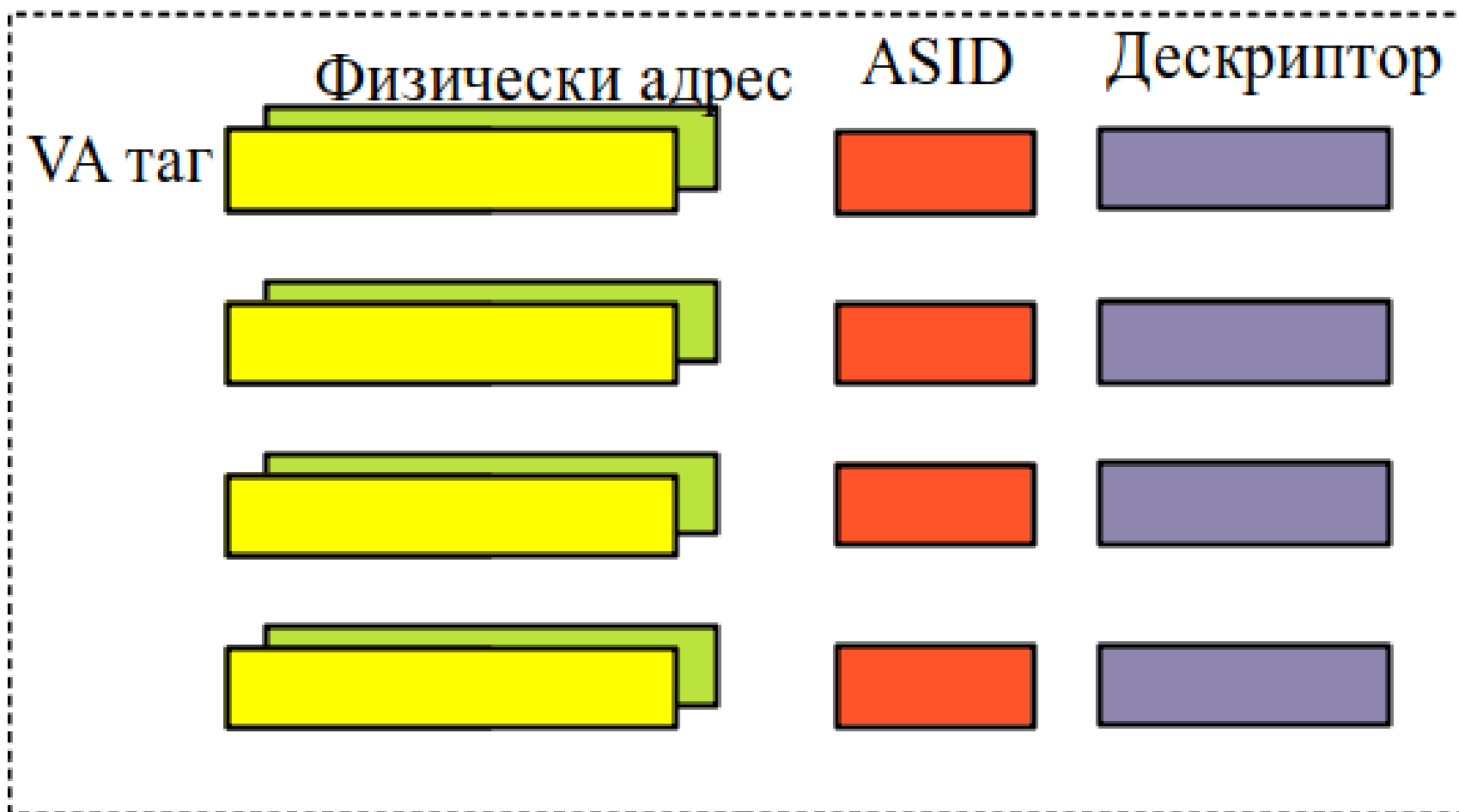
# Модул за организация на паметта (MMU)

Един TLB съдържа следната информация:

- \*VA таг (**V**irtual **A**ddress tag, **P**age **N**umber) – номер на страница, чийто физически адрес е буфериран
- \*Физически адрес – физическият адрес на страницата, отговарящ на номера на страницата от виртуалния адрес.
- \*ASID (**A**ddress **S**pace **I**dentifiers) – уникален номер, който се присвоява на всеки VA таг, за да се знае виртуалния адрес от точно коя нишка идва.
- \*Дескриптор – съдържа правата за достъп за съответния регион.

# Модул за организация на паметта (MMU)

TLB



# Модул за организация на паметта (MMU)

**TLB съвпадение (TLB hit)** – ако номера на страницата на виртуалния адрес съвпадне с номера на VA тагът и може да се намери физическия адрес от TLB модула, а не от транслационната таблица в основната (RAM) памет.

**TLB пропуск (TLB miss)** – ако номера на страницата на виртуалния адрес не съвпадне с номера на VA тагът и не може да се намери физическия адрес от TLB модула, той трябва да се извлече чрез TWU от транслационната таблица в основната (RAM) памет.

# Модул за организация на паметта (MMU)

Софтуерните драйвери (модули) обаче работят с конкретни физически адреси ...

Какво се прави тогава, ако е включено MMU-то?

# Модул за организация на паметта (MMU)

Под Линукс се използват две API функции за тази цел: `get_resource()` и `ioremap()`. Първата известява ядрото на OS, че хардуерният модул се използва от драйвер, втората връща указател към адреса на първия регистър на модула (в случая RCC).

```
uint32_t *reg_pointer;
struct resource *rcc_module;

rcc_module = platform_get_resource(pdev, IORESOURCE_MEM, 0);
if(unlikely(!rcc_module)){
    pr_err(" Specified Resource Not Available... 0\n");
    return -1;
}

//Converting the physical addresses to virtual for using in driver
rcc_base_virtual = ioremap_nocache(rcc_module->start, resource_size(rcc_module));

if(unlikely(!rcc_base_virtual)){
    printk(KERN_ALERT "(cannot map IO)\n");
    return -1;
}

//Modify register just as you would modify it in a bare-metal firmware
reg_pointer = (rcc_base_virtual + RCC_AHB1ENR_OFFSET);
*reg_pointer |= RCC_AHB1ENR_GPIOA_EN_MASK;
```

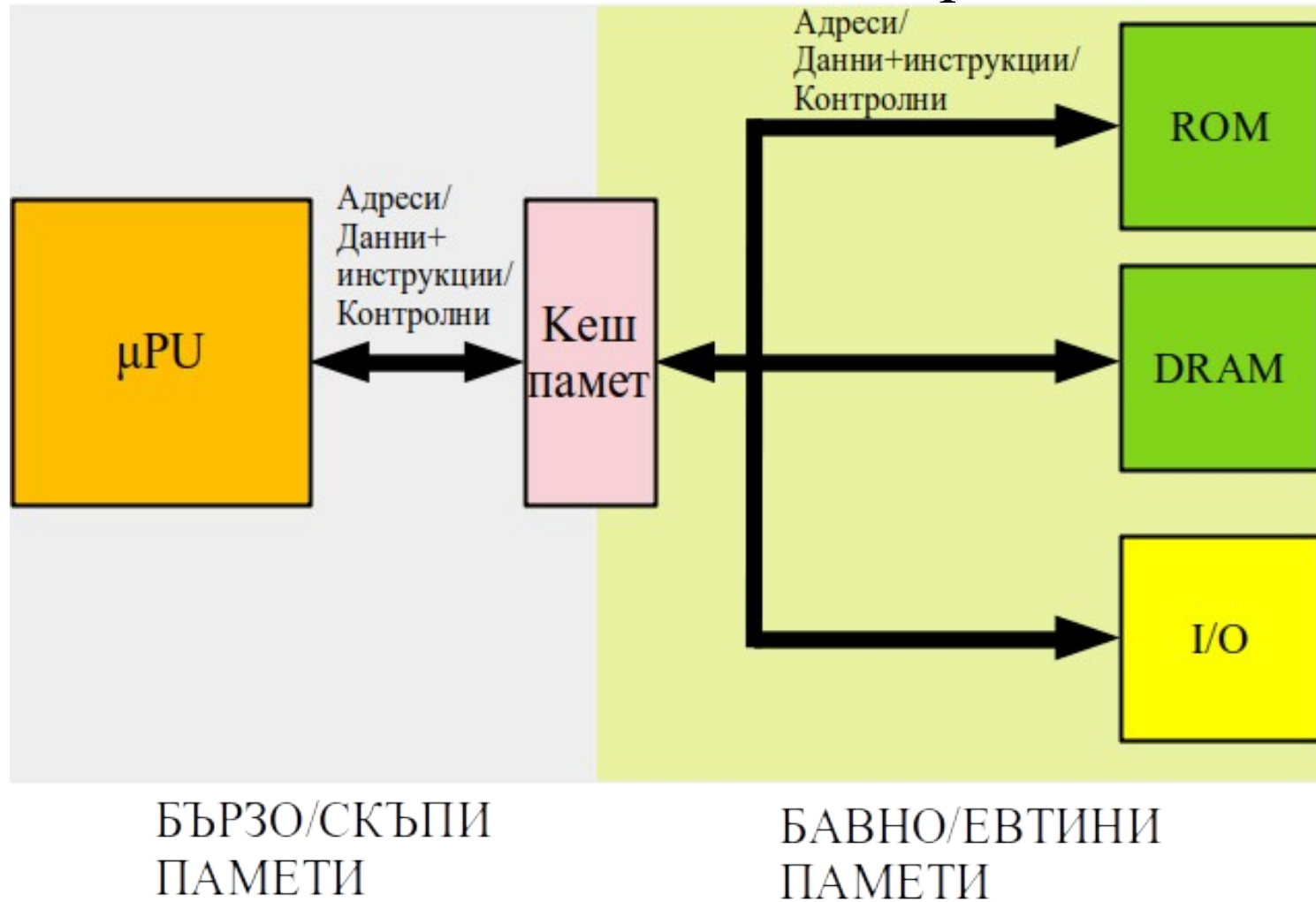
# Кеш памети

**Кеш памет** (cache memory) – буферна памет, в която се зареждат инструкции и данни от основната памет, с цел да се повиши бързодействието на връзката микропроцесор-памет. Клетките на тази памет се реализират със същата технология, с която се реализира SRAM.

Микропроцесорите са много по-бързи от паметите (DRAM, и всички видове ROM) и, в повечето случаи, бързодействието на една система се определя от по-бавния елемент (bottleneck). За да се преодолее този недостатък, най-често използваните части от програмата се извличат бавно от основната в буферната памет, а след това се достъпват бързо от  $\mu$ PU. За да работи този метод, **схемата по която се зареждат кеш паметите трябва да предскаже правилно** коя част от програмата ще се изпълни следваща.

# Кеш памети

Кеш паметите се зареждат от хардуер и програмистът не вижда този процес, освен че програмата му започва да се изпълнява по-бързо [11]. Кешът се пуска в началото от програмата, преди да се стартира изпълнението на основния алгоритъм.



# Кеш памети

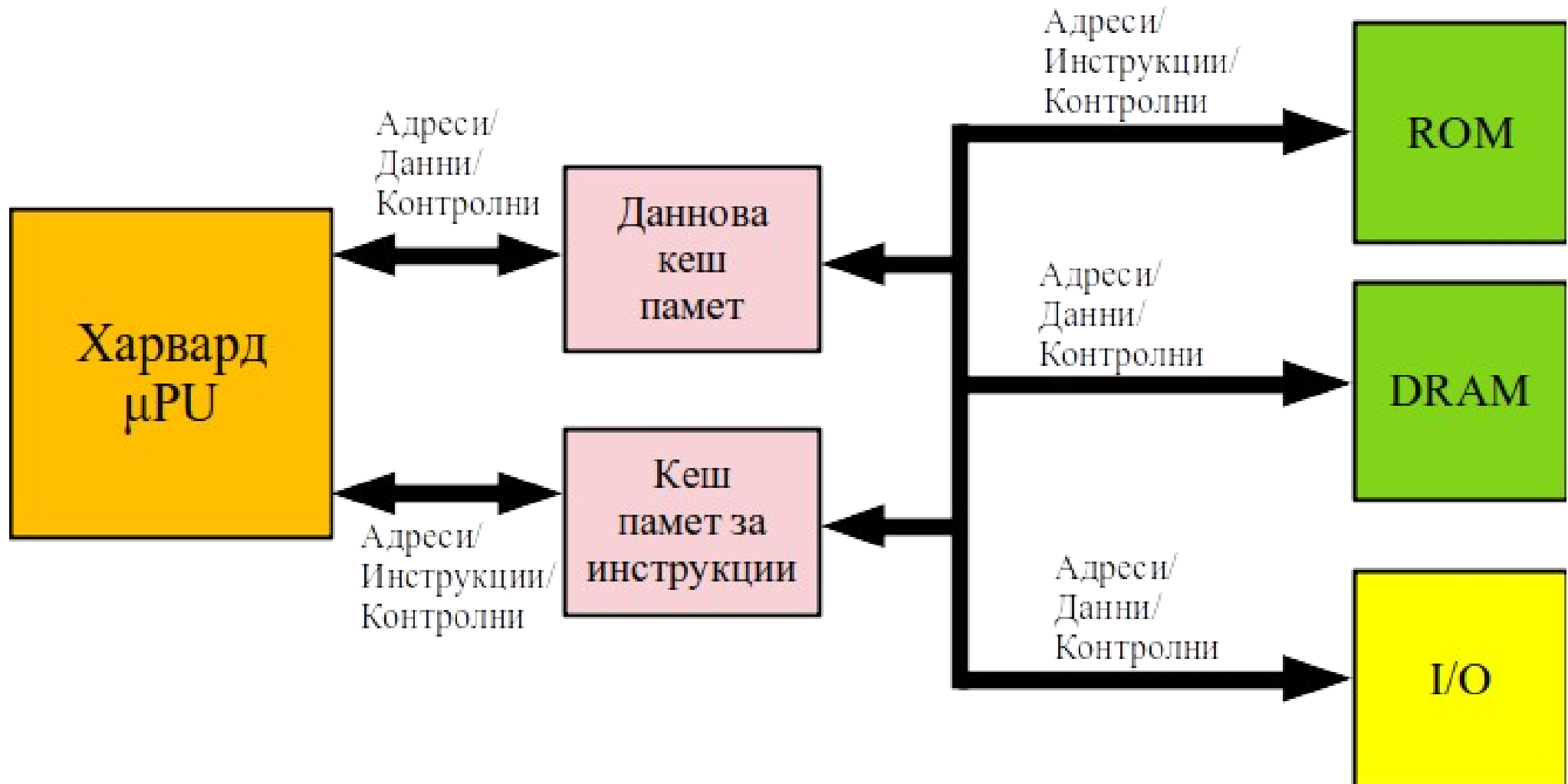
Харвард архитектурите изискват използването на **отделни кеш памети:**

- \*кеш памет за инструкции (instruction cache, icache)**
- \*кеш памет за данни (data cache, dcache)**

Така докато се извличат инструкции може в паралел да се извличат и данни. Ако кешът е общ (за данни и инструкции), това няма да е възможно, понеже магистралата към кеша е една.



# Кеш памети



# Кеш памети

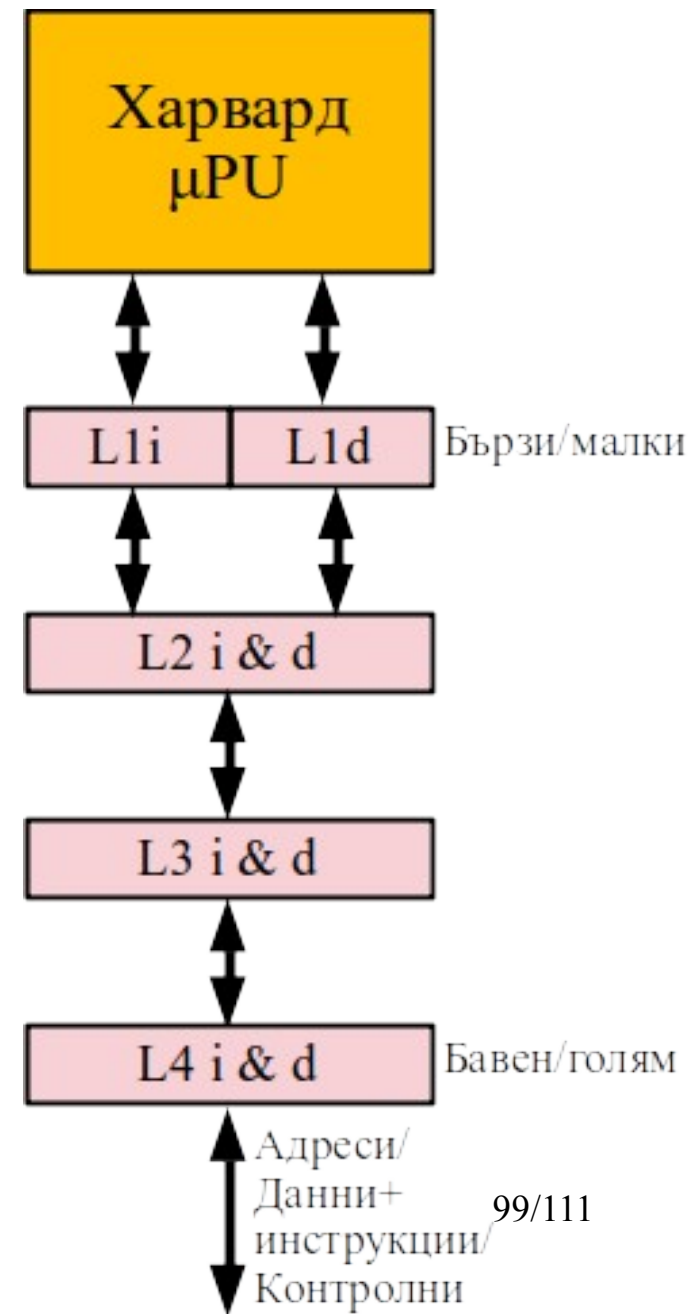
Кеш паметите имат **йерархия**. Регистрите на кеша, които се достъпват за няколко такта (най-бързи), но са малко на брой, са част от L1 кеш (level 1 cache). Всяко следващо ниво (към днешна дата L2, L3, L4) е с по-голям обем, но отнема повече тактове за достъп. Типични стойности на тактовете са дадени в [11].

Вид памет	Тактове
Регистри на ядрото	1
L1d кеш	~3
L2 кеш	~14
DRAM	~240

# Кеш памети

**Припокриващ кеш (inclusive cache)** – данните в L1 са “бързо” копие на данните от L2. Данните от L2 са “бързо” копие на данните от L3 и т.н. Ако L1=1kB, L2=2kB, L3=4kB, L4=16kB, то кешираният регион от RAM е 16 kB.

**Неприпокриващ кеш (exclusive cache)** – данните в L1, L2, L3 и L4 не се припокриват. Ако L1=1kB, L2=2kB, L3=4kB, L4=16kB, то кешираният регион от RAM е 23 kB [12].



# Кеш памети

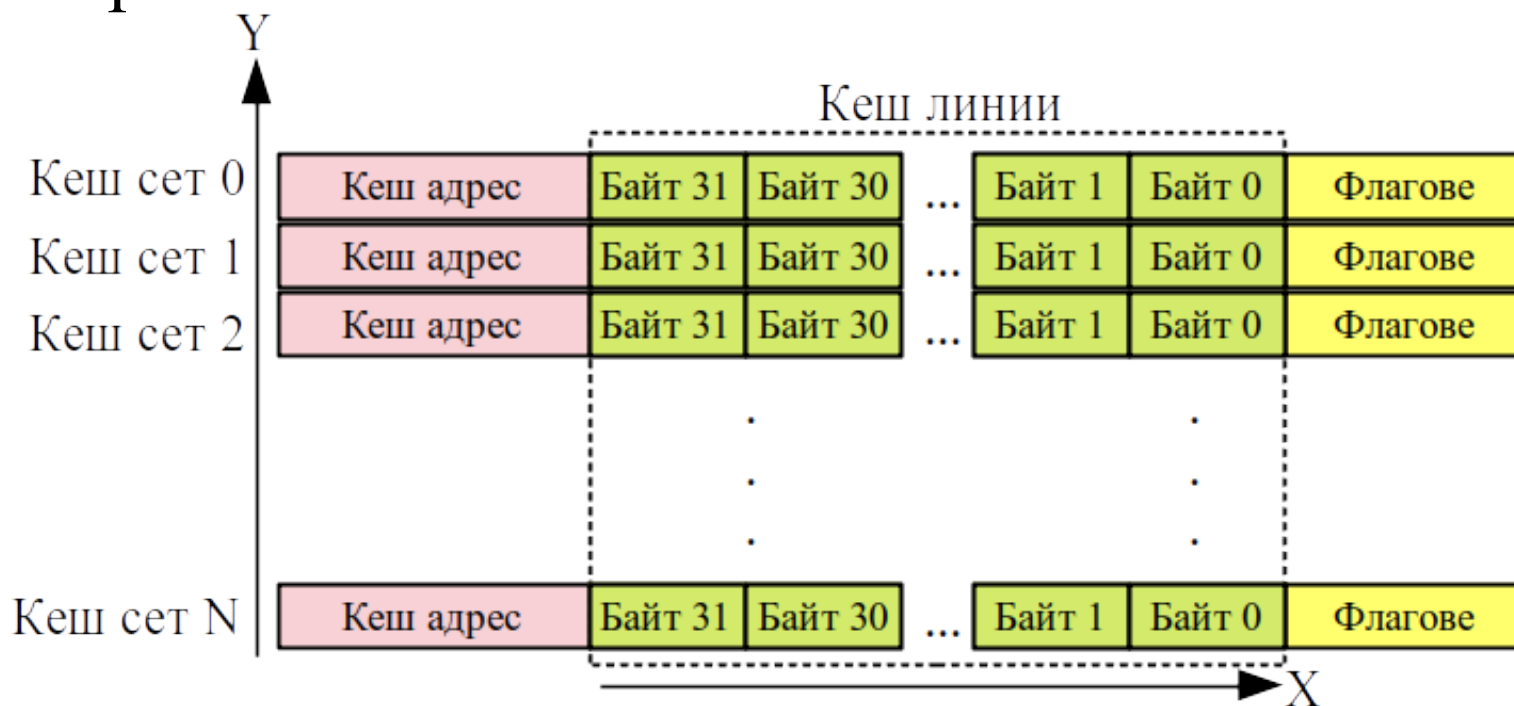
**Кеш линия** (cache line) — копие на няколко данни/инструкции от последователни адреси в основната памет, които са заредени в кеш модула. Понеже данните/инструкциите от DRAM се четат по няколко наведнъж (виж burst read на DRAM от лекцията за паралелните интерфейси), то и връзката DRAM-кеш ще е по-бърза, ако се буферират по няколко числа наведнъж. Типичен размер на линията — 32 или 64 байта.

**Кеш сет** (cache set) — кеш линия с добавени адрес и флагове (виж следващия слайд).

# Кеш памети

Всеки кеш сет се състои от:

- \***адрес** – уникален адрес на сета в кеша;
- \***даннов блок** – кеш линия с кеширани копия на байтове от основната памет;
- \***флагове** – 1 бит “валиден” указва дали данните/инструкциите в кеша са валидни и може да се използват, 1 бит “мръсен” указва дали данните на реда са променени от  $\mu$ PU, но промените все още не са отразени в основната памет.



# Кеш памети

Всеки кеш адрес се състои от:

\***кеш таг** (cache tag) – старшата част на физическия адрес на данните, които са кеширани. Кеш тагът се използва за **транслиране** на кеш адреси  $\leftrightarrow$  физически адреси.

\***кеш индекс** (cache set / cache index) – отместване по редове в кеш модула (Y)

\***кеш отместване** (cache offset) – отместване в линията на кеша (X)

# Кеш памети



Y  
↑

Кеш линии

Кеш сет 0

Кеш сет 1

Кеш сет 2

Кеш адрес	Байт 31	Байт 30	...	Байт 1	Байт 0	Флагове
Кеш адрес	Байт 31	Байт 30	...	Байт 1	Байт 0	Флагове
Кеш адрес	Байт 31	Байт 30	...	Байт 1	Байт 0	Флагове

·  
·  
·

·  
·  
·

Кеш сет N

Кеш адрес	Байт 31	Байт 30	...	Байт 1	Байт 0	Флагове
-----------	---------	---------	-----	--------	--------	---------

X  
→

# Кеш памети

**Кеш съвпадение (cache hit)** – събитие, при което  $\mu\text{PU}$  достъпва кеш паметта и там намира валидни инструкции/данни.

**Кеш пропуск (cache miss)** – събитие, при което  $\mu\text{PU}$  достъпва кеш паметта и там не намира валидни инструкции/данни. Тогава кеш модулът ще се обърне към основната памет, ще извлече необходимата информация и **ще я запише на мястото на някоя друга**. Пропуските водят до блокиране на ядрото (stall), докато се копират данните от основната памет.



# Кеш памети

**Мръсен кеш (dirty cache)** – кеш сет, който е променен от  $\mu\text{PU}$ , но промяната не е отразена в основната памет.

**Кеш невалидиране (cache invalidate)** – процес на изтриване на кеш сет и записване на  $\text{valid bit} = 0$ . Ако  $\mu\text{PU}$  прочете този кеш сет и  $\text{valid bit} = 0$ ,  $\mu\text{PU}$  ще бъде блокиран и нова информация ще бъде заредена в сета.

**Кеш изчистване (cache clean)** – отразяване на промяната от мръсен кеш в основната памет, така че и двете да съдържат едни и същи данни (т.е. кешът да стане **кохерентен**). Битът за мръсен кеш се записва  $\text{dirty bit} = 0$ .

# Кеш памети

Кеш паметите имат по няколко контролни регистъра, чрез които може да се **извършват някои основни операции** от потребителската програма:

- \*регистър за пускане/спиране (enable/disable)
- \*регистър за невалидиране (invalidate)
- \*регистър за изчистване (clean)

# Кеш памети

**Асоциативност на кеша** – ограничение в адресите от кеша, където може да бъдат кеширани данни/инструкции (cache placement policy).

**\*изцяло асоциативен кеш** (fully associative cache) – произволни региони от основната памет може да се записват на произволни места в кеша (изискват много хардуер за направа);

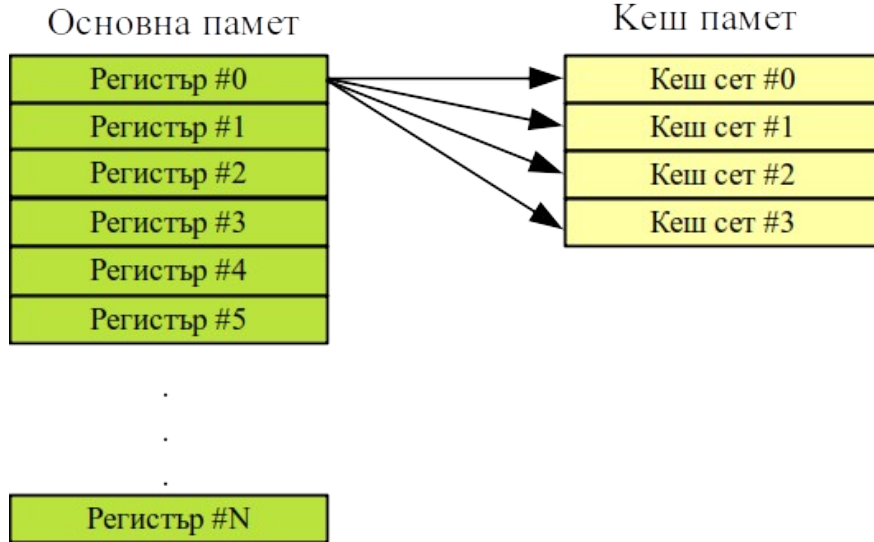
**\*директен кеш** (direct mapped) – даден регион от основната памет може да се запише само на едно място в кеша (малко хардуер, но много често има кеш пропуски);

**\*N-пътен сет-асоциативен** (N-way set associative) – даден регион от основната памет може да се запише само на N места от кеша.

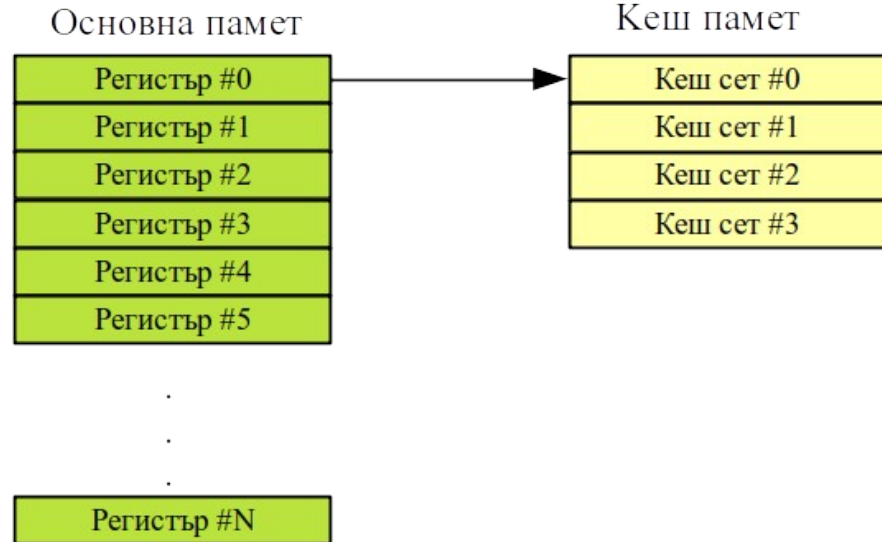
*Пример* – 2-пътен сет асоциативен кеш – позволява данните/инструкциите да се запишат на едно от две възможни места.

# Кеш памети

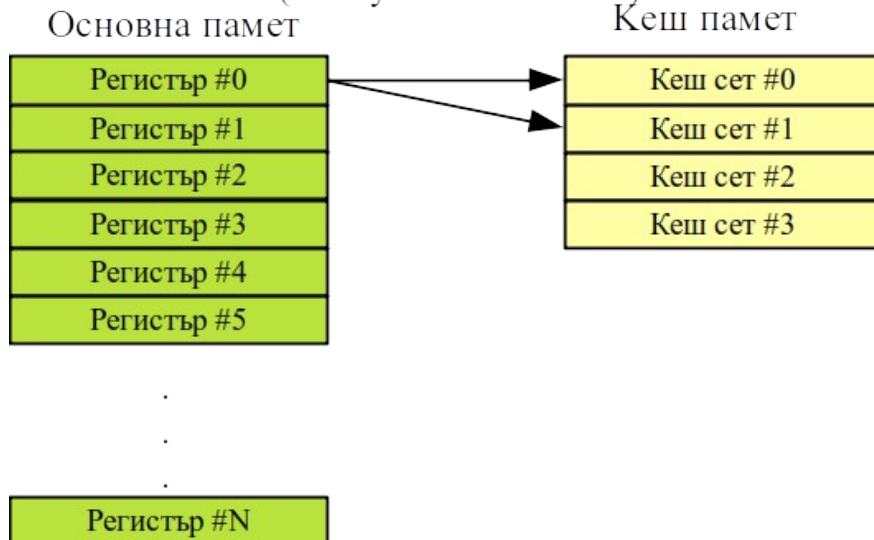
Изцяло асоциативен  
(fully associative)



Директен  
(direct mapped)



2-пътен сет асоциативен  
(2-way set associative)



# Кеш памети

**Схема за изтриване** (cache replacement policy) – решението коя информация да бъде изтрита при записване на нова в кеша. Когато кешът извлече данни/инструкции по някоя от 3-те възможни асоциативни схеми, новата информация трябва да се запише върху стара такава. Съществуват много алгоритми, всеки с предимства и недостатъци, съответно – дадени алгоритми са подходящи за дадено приложение.

Някои от алгоритмите са: алгоритъм на Беладис, FIFO, LIFO, FILO, LRU, TLRU, MRU, PLRU, RR, SLRU, LFU, LFRU, LFUDA, LIRS, ARC, AC, CAR, MQ и т.н.

# Кеш памети

Кеш памети и MMU – виртуалните адреси съдържат индекси на кеш сетове от различни нива кешове [11].

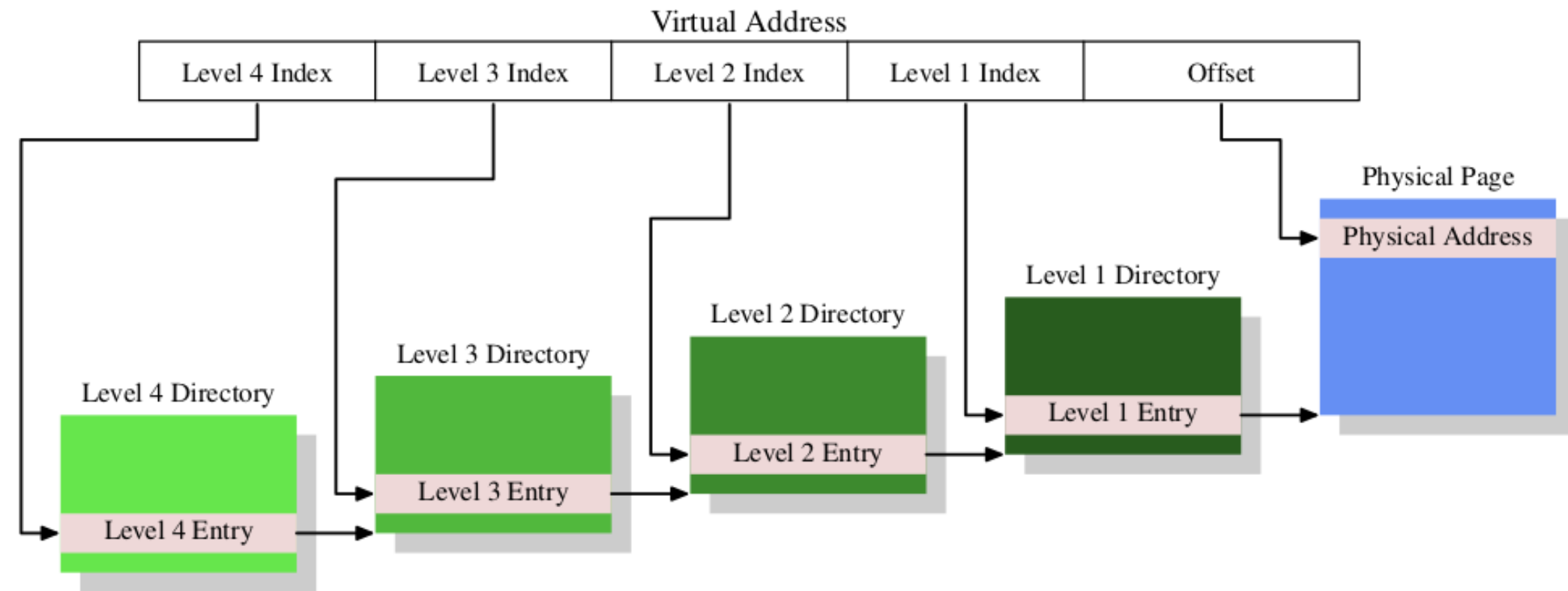


Figure 4.2: 4-Level Address Translation

# 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.
- [6] “MSP430FR6xx Family User’s Guide”, SLAU367N, Texas Instruments, 2017.
- [7] “Efficient Multiplication and Division Using MSP430 MCUs”, SLAA329A, Texas Instruments, 2018.
- [8] <http://msp gcc.sourceforge.net/manual/x613.html>
- [9] “Memory Management”, 101811\_0100\_00, ARM Ltd, 2019.
- [10] S. Harris, D. Harris, “Memory Systems”, Digital Design and Computer Architecture, 2016.
- [11] U. Drepper, “What Every Programmer Should Know About Memory”, online, 2007.
- [12] G. Cooperman, “Cache Basics”, online, 2003.  
<https://course.ccs.neu.edu/com3200/parent/NOTES/cache-basics.html>