Основи на микропроцесорите

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!

ЕСФ България
Европейски социален фонд

Съдържание

Разлика между µPU и MCU

Микропроцесорът (µPU – **micro P**rocessing **U**nit) е цифрова интегрална схема , която може да извършва аритметико-логически операции, умножение, деление, четене и запис на данни в паметта/периферията.

Микропроцесорните системи съдържат поне 4 интегрални схеми: микропроцесор, външна RAM памет, външна ROM памет и външна периферна интегрална схема (схеми).

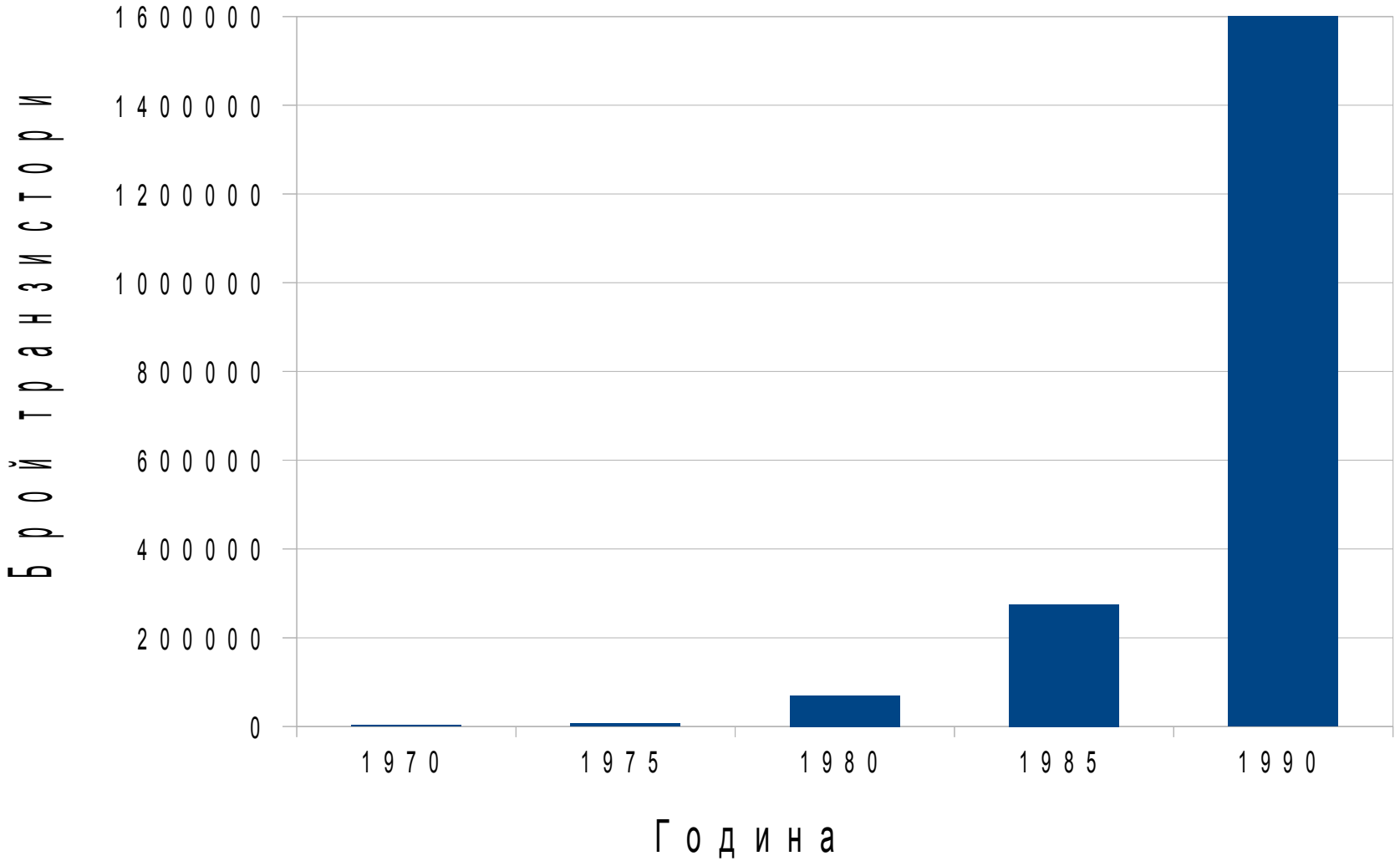
Една такава система наподобява структурата на персонален компютър, но разбира се – разполага с по-малки изчислителни ресурси.

Разлика между µPU и MCU

С напредването на микроелектрониката става възможна интеграцията на голям брой транзистори на един кристал [а]. От графиката на следващия слайд е видно, че максималният брой на транзисторите е:

* 2300 през 1975 г.
* 1,6 млн през 1990 г.
* 7 млрд. през 2012 г., на кристал с площ 0.5 cm^2.

Разлика между µPU и MCU



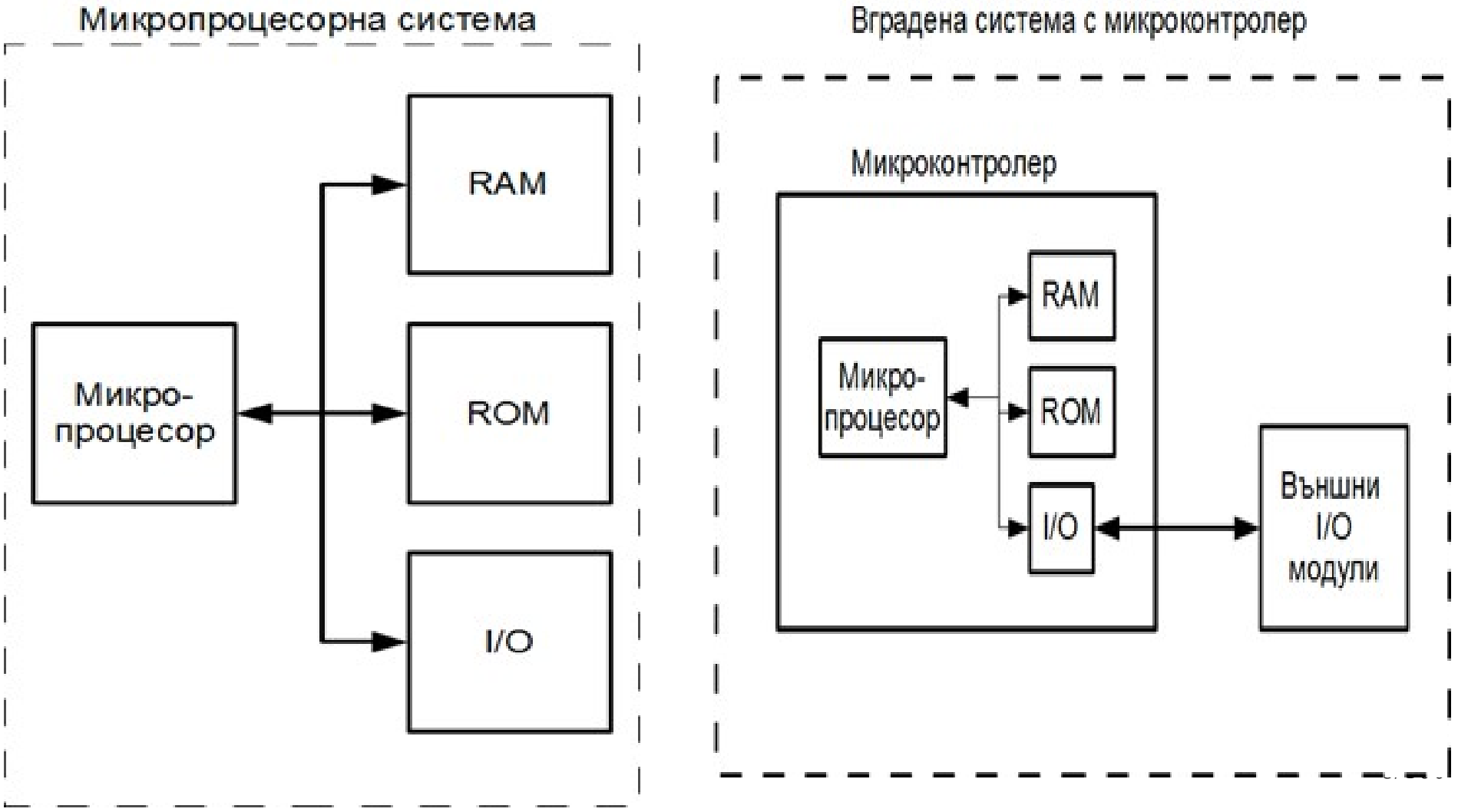
Разлика между µPU и MCU

В резултат на тези постижения **микропроцесор, RAM, ROM и I/O се интегрират на един единствен чип**. Първоначално тези ИС били наричани едночипови микрокомпютри, а днес – **микроконтролери (MCU – MicroController Unit)**.

Микроконтролерните системи съдържат поне един микроконтролер, към който обикновено се свързват сензори и актуатори. Те са аналогични на микропроцесорните системи, но обикновено постигат дадена функция с по-малък брой чипове (поради интегрираните модули).

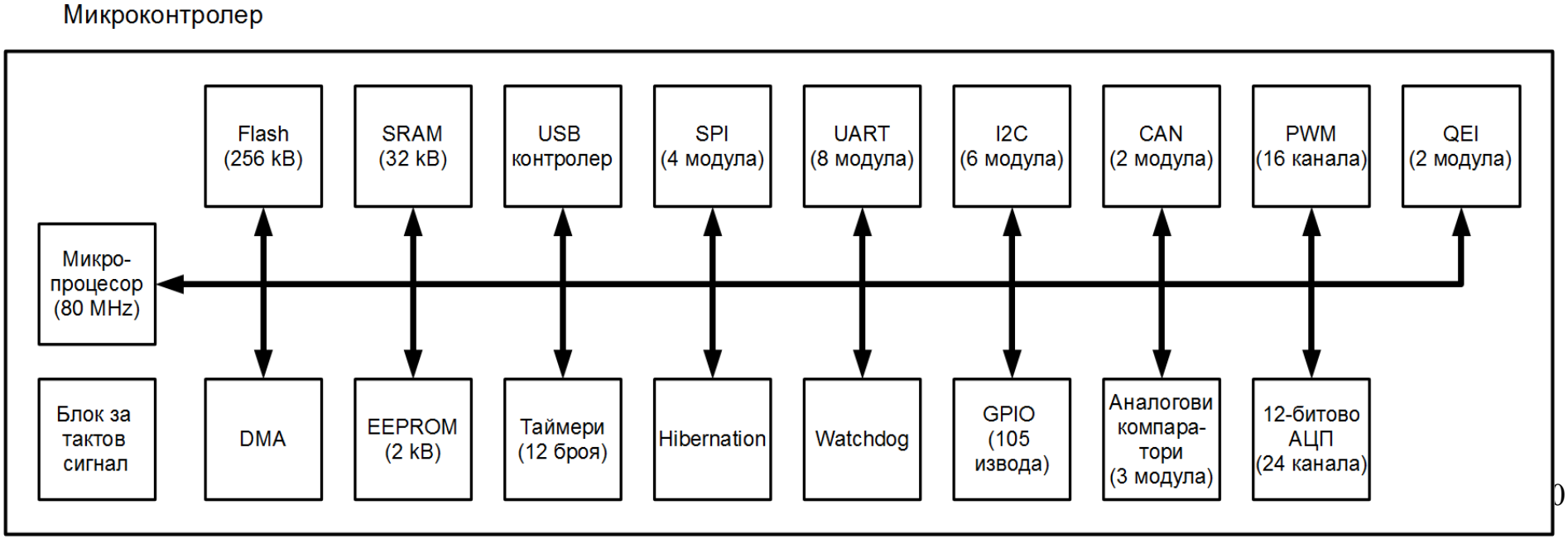
Разлика между µPU и MCU

На фигурата е дадена структурна схема на система, използваща микропроцесор и микроконтролер.



Разлика между µPU и MCU

На фигурата е дадена блокова схема на съществуващ микроконтролер. Вижда се, че това е цяла една микропроцесорна система интегрирана на един чип. Предимствата на този подход, пред реализирането с отделни ИС е, че цената е по-ниска, консумацията на статична мощност е по-малка и заеманата площ от печатната платка е по-малка.



Разлика между µPU и MCU

Трябва да се отбележи наличието на **аналогови модули** в микроконтролера – често се интегрират АЦП, ЦАП и/или аналогови компаратори, което по-рядко се среща във FPGA например.

Като недостатък на микроконтролерите може да се посочи фиксираната вътрешна архитектура.

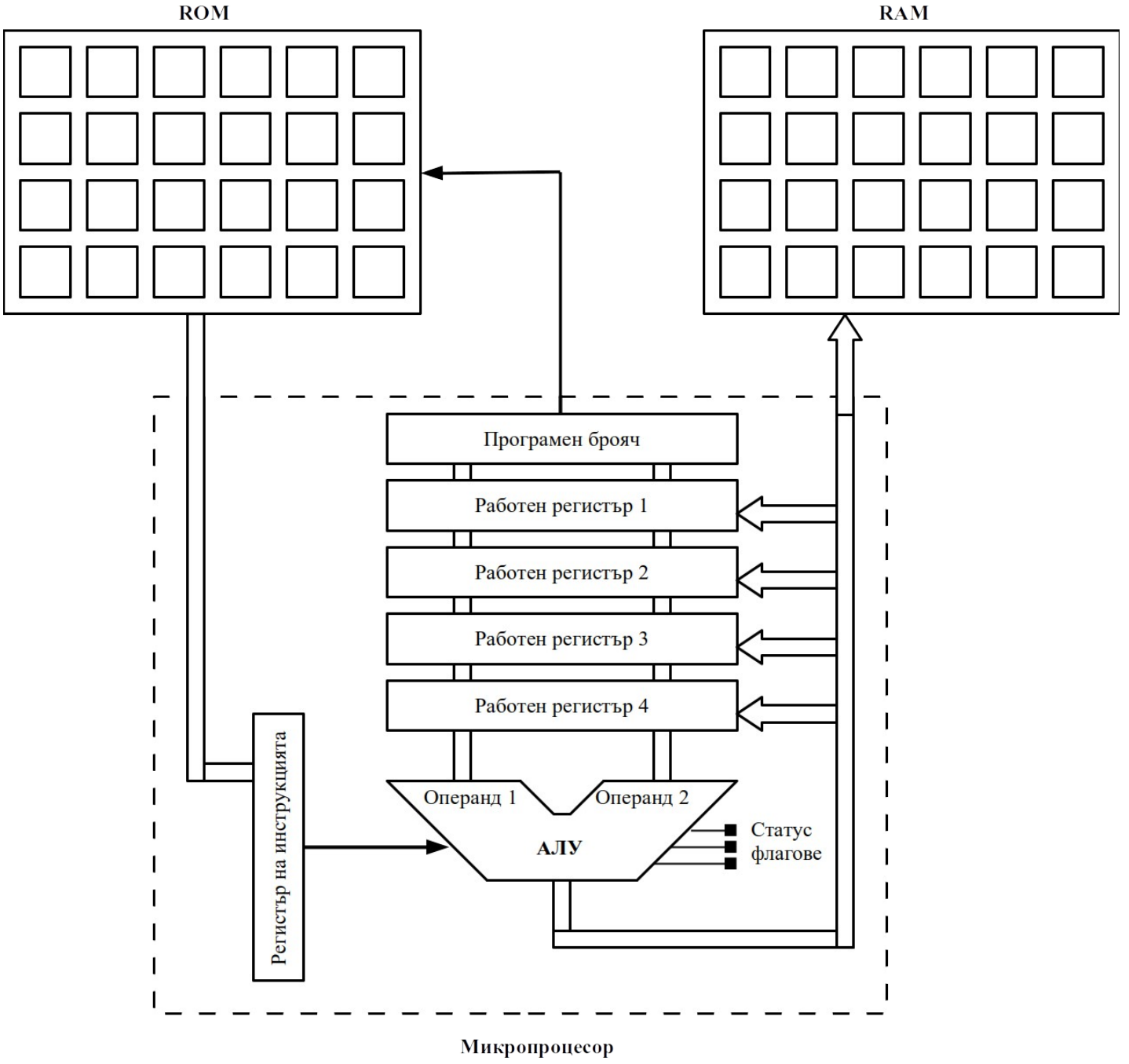
Например, размера на паметта е фиксиран – понеже тя е вградена в чипа, то инсталирането на повече памет е невъзможно. Частично решение е да се свърже допълнителна памет към някои от серийните интерфейси на чипа (SPI, I2C), но обменът на данни ще е значително по-бавен, отколкото с вътрешната памет.

Структурна схема на микропроцесор

Без значение дали са част от микропроцесорна система или част от микроконтролер, микропроцесорите съдържат някои основни блокове, като АЛУ, умножители, модули за числа с плаваща запетая, регистри с общо преназначение, специални регистри, буфери и др.

На следващия слайд е показана силно опростена блокова схема на микропроцесор и прилежащите му RAM и ROM.

Структурна схема на микропроцесор



Структурна схема на микропроцесор

Изпълнението на програмата започва с извличане (**fetch**) на първата инструкция от ROM паметта и записването ѝ в регистъра на инструкцията [1].

След това инструкцията се декодира (**decode**). В този етап се определя вида на операцията (събиране, изваждане, преместване, сравняване и т.н.) и се активират съответните модули.

Операндите (данните върху които ще се работи) се записват в някои от работните регистри 1 ÷ 4, или се взимат от паметта.

Структурна схема на микропроцесор

След това инструкцията се изпълнява (**execute**) от някои от следните модули:

* Аритметико-логическото (ALU – arithmetic logic unit) устройство извършва указаната операция и записва резултата или в работните регистри (процес наричан *write back*), или в RAM.

* Ако операцията е умножение, използва се отделен умножителен модул, (MPY – **M**ulti**P**l**Y**).

* Ако инструкцията е за работа с дробни числа, използва се отделен модул за числа с плаваща/фиксирана запетая (FPU – **F**loating **P**oint **U**nit).

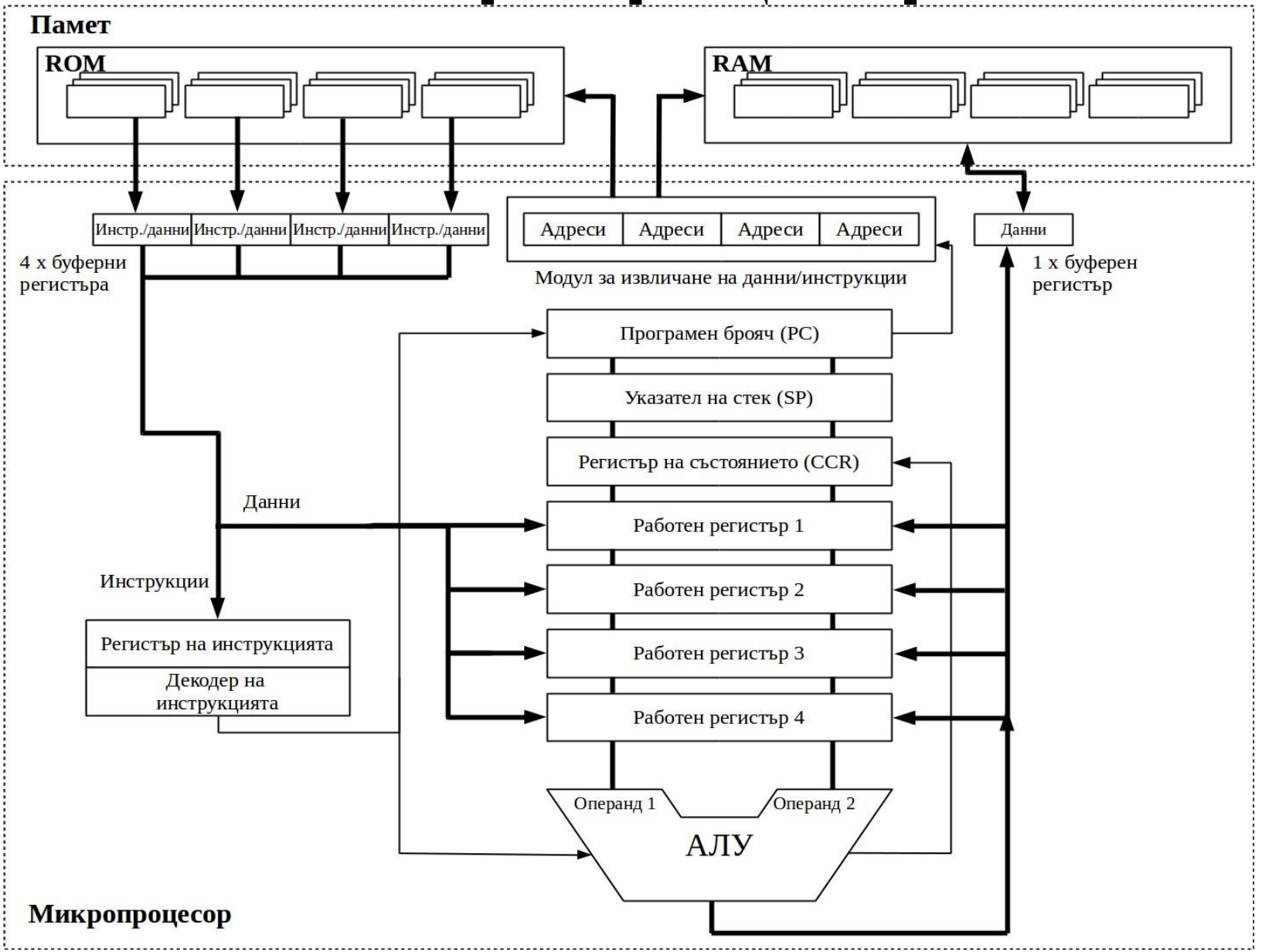
Структурна схема на микропроцесор

Изпълнението на инструкцията завършва със запис обратно в работните регистри (**write back**) и/или запис в паметта/периферията (**memory**).

Структурна схема на микропроцесор

Ако разгледаме структурната схема на по-ниско ниво, ще се получи конфигурацията показана на следващия слайд. Това е Харвард архитектура. Структурата на реалните микропроцесори варира, но принципът на работа е един и същ. Основните блокове са:

Структурна схема на микропроцесор



Структурна схема на микропроцесор

Буфери на данновите магистрали/магистралите за инструкции – те свързват вътрешните и външните за микропроцесора шини. Използването им се налага поради по-голямото бързодействие на микропроцесора спрямо външната памет [6]. Например, ако микропроцесора е 4 пъти по-бърз от ROM паметта, трябва да има поне 4 буфера, които да компенсират това несъответствие. ROM паметта ще запише в тях 4 думи на един такт, което е еквивалентно на 4 пъти по-бърза ROM памет.

Структурна схема на микропроцесор

Буфери на адресните магистрали – аналогични на буферите за данни.

Буферите на RAM паметта са двупосочни, а на ROM – еднопосочни.

Ако към буферите има свързани контролери за “интелигентно” извличане на инструкции/данни, те се наричат **кеш памет**. Размерите на кеш паметите варират и са от порядъка на kB – MB.

Скаларна и суперскаларна архитектура

Увеличаване на изчислителната мощ на микропроцесорите може да стане по два начина:

* чрез увеличаване на тактовата му честота;

* чрез добавяне на допълнителни логически блокове (например още едно ALU, MPY или FPU).

Скаларна и суперскаларна архитектура

Скаларен микропроцесор (scalar microprocessor) е този микропроцесор, който може да изпълнява само една инструкция в даден момент. Следващите инструкции от програмата ще бъдат изпълнени едва когато завърши изпълнението на настоящата инструкция. Инструкциите се изпълняват в реда, в който са записани в програмата.

Пример за скаларен микропроцесор: MSP430, PIC16, ARM Cortex-M4.

Скаларна и суперскаларна архитектура

Суперскаларен микропроцесор (superscalar microprocessor) е този микропроцесор, който може да изпълнява две или повече инструкции в паралел. За целта той трябва да съдържа два или повече функционални блока в ядрото си [4]. Типичен пример за функционален блок е АЛУ.

Пример за суперскаларен микропроцесор: Motorola MC88100, ARM Cortex M7, ARM Cortex A9.

Скаларна и суперскаларна архитектура

Скаларният микропроцесор може да обработва едно число за един или няколко такта. Числото е **скаларна величина**, т.е. целочислена стойност, число с фиксирана или плаваща запетая.

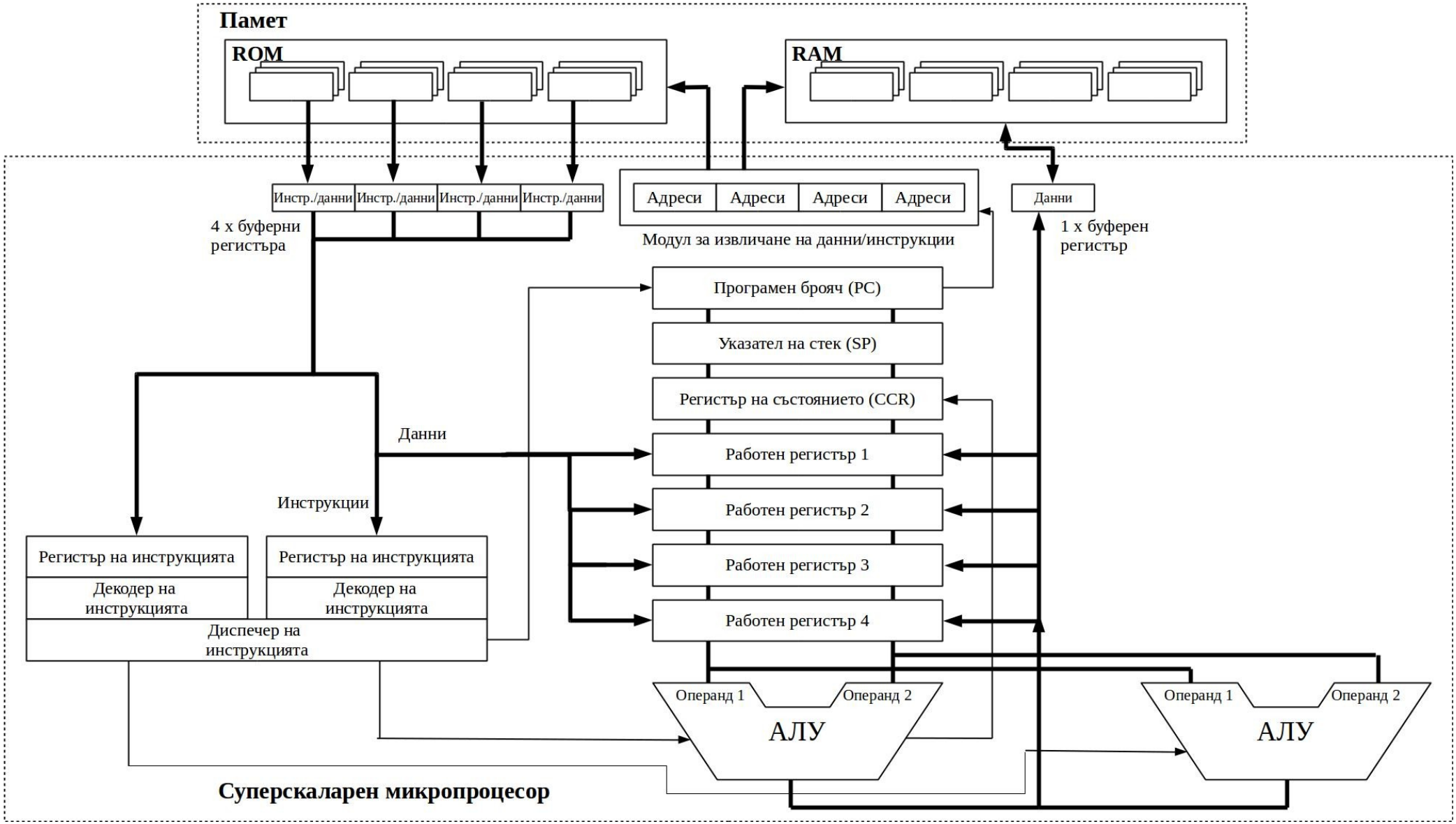
Суперскаларният микропроцесор може да обработва две или повече числа за един или няколко такта. Числата са скаларни величини, т.е. целочислени стойности, числа с фиксирана или плаваща запетая.

Скаларна и суперскаларна архитектура

Ако функционалните модули могат да обработват по няколко числа наведнъж, т.е. ако работят върху **масиви от данни**, микропроцесорът се нарича **векторен**.
Пример за векторен микропроцесор - Fujitsu FACOM VP-100, RISC V, графични процесори.

Ако в един микропроцесор са вградени няколко функционални векторни модула, той се нарича **супервекторен**.
Пример за супервекторен микропроцесор - все още липсват практически реализации [7].

Скаларна и суперскаларна архитектура



Скаларна и суперскаларна архитектура

Диспечер на инструкцията е контролен блок в скаларен микропроцесор, който проверява дали две (или повече) последователни инструкции могат да бъдат изпълнени в паралел на два (или повече) функционални блока.

Скаларна и суперскаларна архитектура

Диспечерът е необходим, защото невинаги две инструкции могат да бъдат изпълнени в паралел. Например инструкциите:

```
Routine1:
        xor.w R1, R2 ; пресметни R1 | R2 и запиши резултата в R1
        and.w R1, R3 ; пресметни R1 & R3 и запиши резултата в R1
```

не могат да се изпълнят в паралел, защото инструкцията and.w зависи от инструкцията xor.w (заради зависимостта от регистър R1).

Програмен модел

Програмен модел - това е наборът от регистри на даден микропроцесор, които са достъпни (четене/запис) за програмиста. Тяхното съдържание се променя от инструкциите в програмата.

Включват се *регистрите с общо предназначение* (GPR), *програмният брояч* (PC), *стековият указател* (SP), *регистърът на състоянието* (SR или CCR) и други регистри, специфични за дадения микропроцесор.

Програмен модел

Регистри с общо предназначение (**G**eneral **P**urpose **R**egisters, или още - Register file) – използват се за съхранение на данни по време на изпълнението на програмата. Това е най-бързата памет (от няколко клетки), до която микропроцесорът има достъп.

Програмен модел

Тези регистри не са част от адресното поле на микропроцесора. Те се достъпват директно чрез операндите на асемблерните инструкции и обикновено се бележат с “r” или “R”:

add r1, r2, r3

→ r1, r2 и r3 са регистри с общо предназначение

Програмен модел

Броят на регистрите с общо предназначение е важен, защото е един от факторите, от които зависи времето за изпълнение на програмата. Ако програмата е сложна, GPR няма да стигнат и ще трябва да се използва външната RAM, която обикновено изисква десетки и стотици тактове за достъп.

* MC6800 на Motorola има 2 GPR
* MSP430FR5739 на Texas Instruments има 12 GPR

Програмен модел

Програмен брояч (**P**rogram **C**ounter) – регистър, съдържащ адреса на следващата† инструкция, чието изпълнение предстои. PC се увеличава автоматично по време на изпълнението на настоящата инструкция. Числото, с което се увеличава зависи от микропроцесора:

* при ARM Cortex се увеличава винаги с 2 или 4
* при MSP430 се увеличава винаги с 2, 4 или 6

† - има изключения, виж по-напред за конвейери

Програмен модел

Защо е необходим PC?

Запис на число в PC ще доведе до промяна в хода на програмата – преход. Така се извикват подпрограми и реализират цикли.

Стойността на PC може да се използва като относителна точка в адресното поле, спрямо която да се направи преход (**P**osition **I**ndependent **C**ode).

Стойността на PC може да се използва като относителна точка в адресното поле, спрямо която да се разположат данни (които да бъдат достъпени).

Програмен модел

Началната стойност на PC (след подаване на захранването) е фиксирана и зависи от конкретния микропроцесор.

* При PIC18 това е адрес 0x0000.

* При MSP430 това е адрес 0xFFFE.

* При ARM Cortex това е адрес 0x0004.

Програмен модел

ВНИМАНИЕ! При конвейерни микропроцесори (ще се изучават по-късно) PC съдържа адрес на инструкция, която е няколко нива по-напред от следващата. Колкото повече степени има конвейера, толкова по-напред ще сочи PC.

* При ARM Cortex-M0 (**три**степенен конвейер) PC се увеличава така:

→ PC = PC + 4 (байта), което е еквивалентно на PC = PC + **две** 16-битови инструкции

или

→ PC = PC + 8 (байта), което е еквивалентно на PC = PC + **две** 32-битови инструкции

Програмен модел

Извадка от асемблерна програма за ARM Cortex-M0. Използват се 16-битови инструкции. PC в инструкцията на адрес 0x168 ще сочи към 0x16c (две по-напред), а не към 0x16A (една по-напред) заради конвейера.

```
0x164       mov  r2, #8
0x166       str  r0, [r1, r2]
0x168       ldr  r0, [PC, #8] ; PC →0x168 + 4 = 0x16c
                                ; 0x16c + 8 = 0x174
0x16a       ldr  r0, [r1, #8]
…
0x170     0xabcdefef          ; Данни
0x174     0xdeadbeef          ; Данни
0x178     0x12345678          ; Данни
```

Програмен модел

Стеков указател (**S**tack **P**ointer) – регистър, съдържащ адреса на клетка памет, където се намира последното записано число служебна информация по време на изпълнението на програмата. Тази клетка памет, заедно с няколко съседни такива, формират специален регион в RAM паметта, наречен стек (стекова памет).

Стекът се достъпва като LIFO (**L**ast **I**n **F**irst **O**ut) буфер. Повечето микропроцесори поддържат специални инструкции, с които се достъпва стекът – това са PUSH и POP.

Програмен модел

Организацията на стека е два вида:

* намаляващ стек (descending stack) – всеки следващ елемент се записва на по-малък адрес

* растящ стек (ascending stack) – всеки следващ елемент се записва на по-голям адрес

Програмен модел

При намаляващ стек:

PUSH – инструкция, която автоматично намалява SP и след това записва един елемент в стека.

POP – инструкция, която чете един елемент от стека и автоматично увеличава SP.

При растящ стек:

PUSH – инструкция, която автоматично увеличава SP и след това записва един елемент в стека.

POP – инструкция, която чете един елемент от стека и автоматично намалява SP.

Програмен модел

Увеличаването и намаляването на адресите става с числа, пропорционални на разредността на микропроцесора:

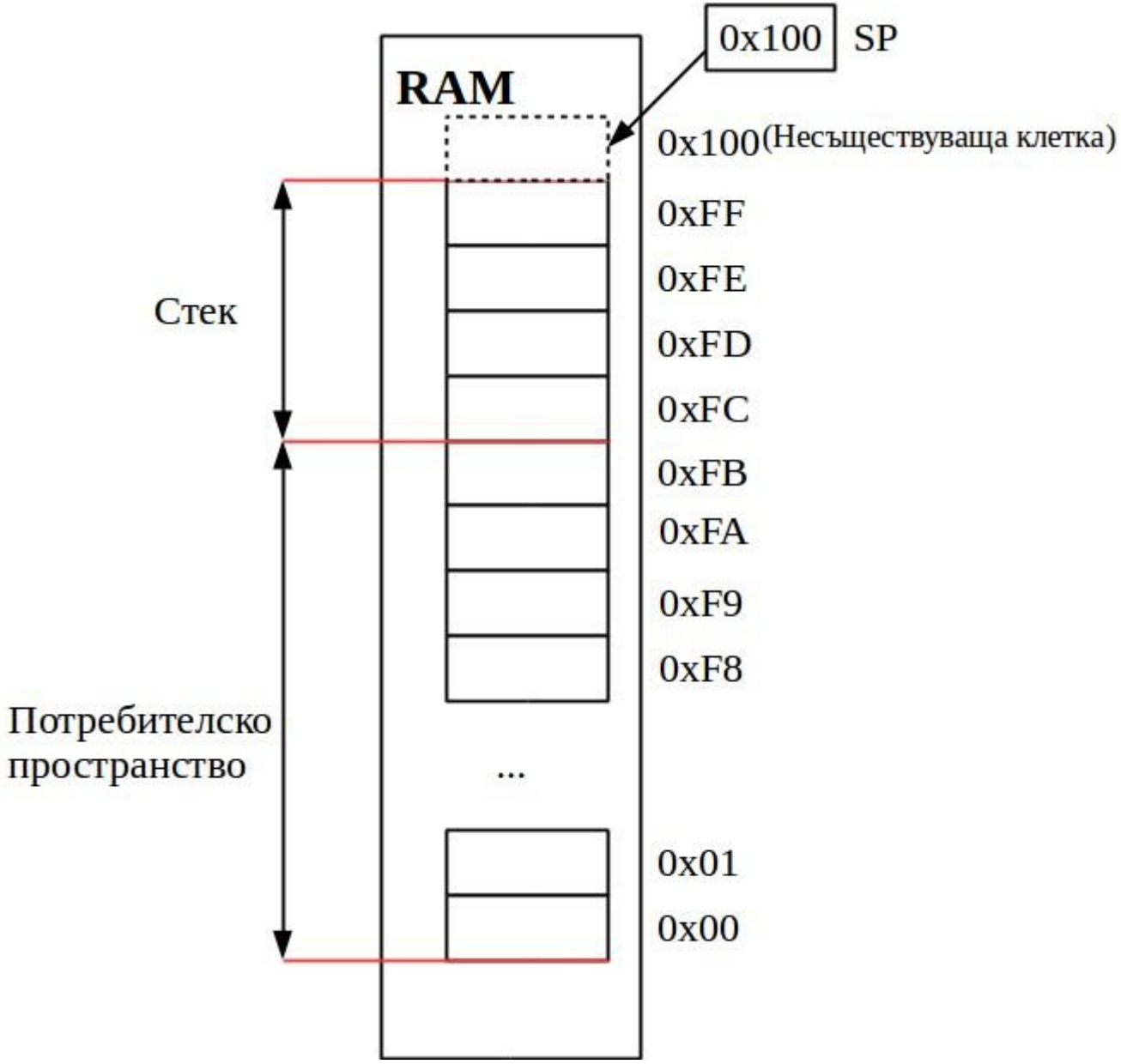
* за 8-битови μPU SP се увеличава с 1
* за 16-битови μPU SP се увеличава с 2
* за 32-битови μPU SP се увеличава с 4

и т.н.

Програмен модел

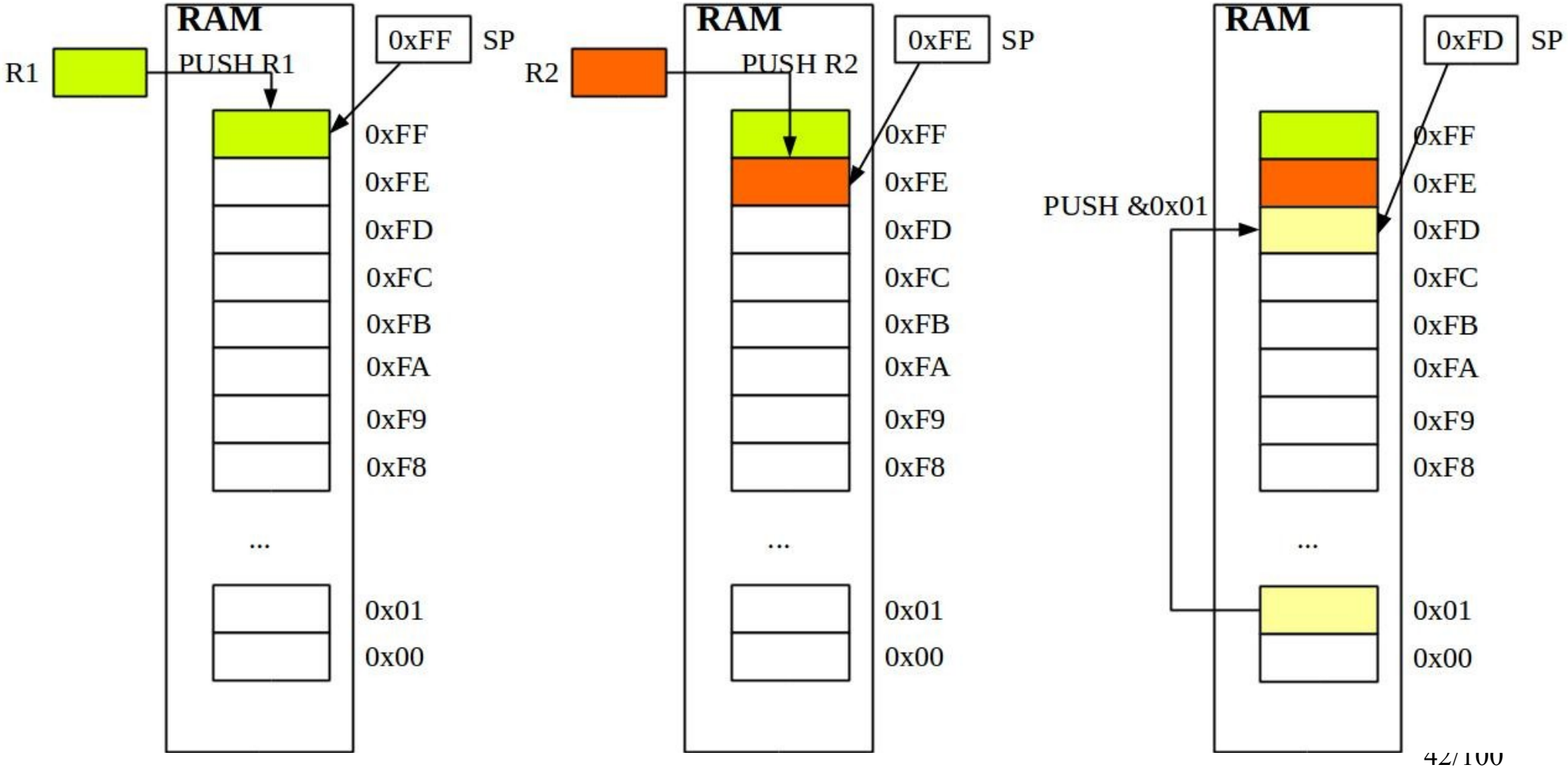
Пример с намаляващ стек.

Инициализация



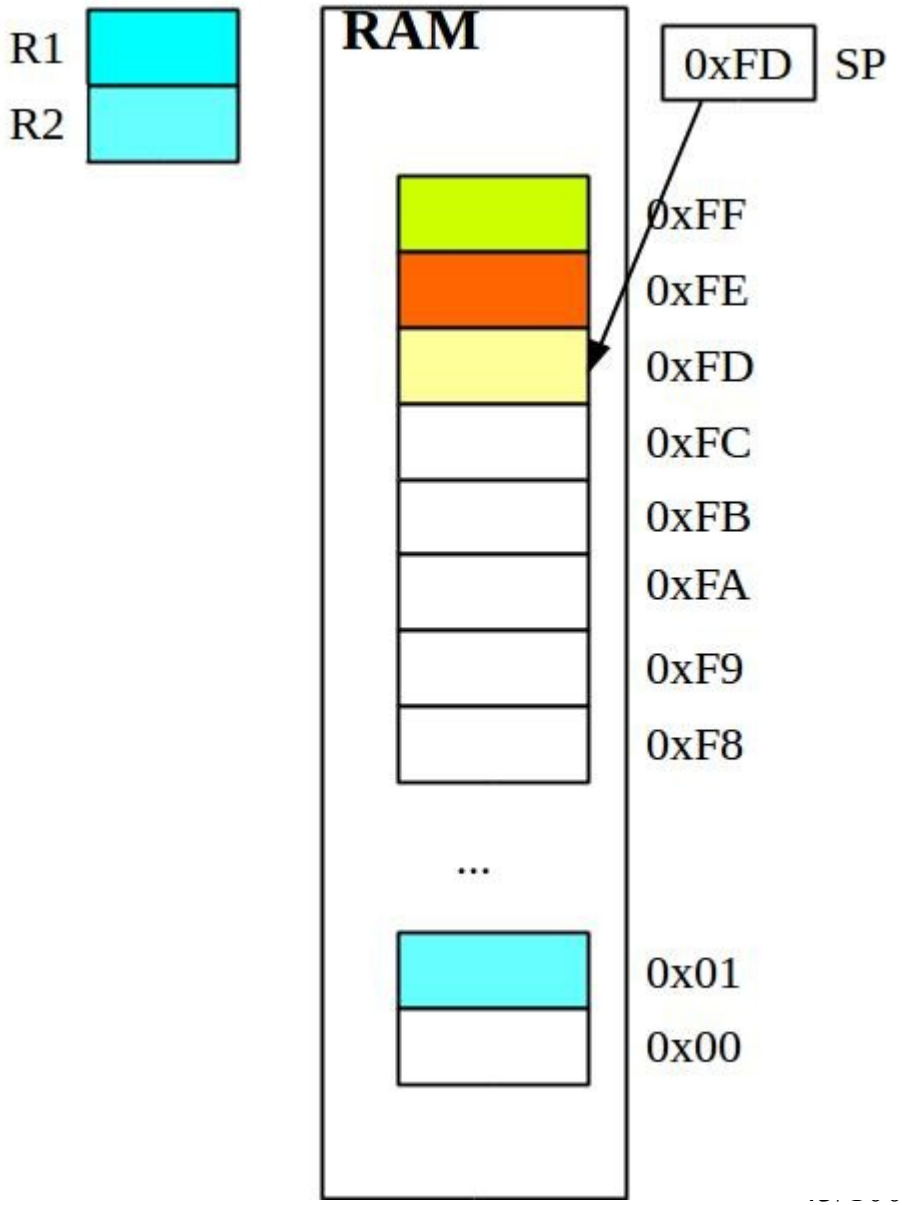
Програмен модел

Пример с намаляващ стек. Запис в стека.



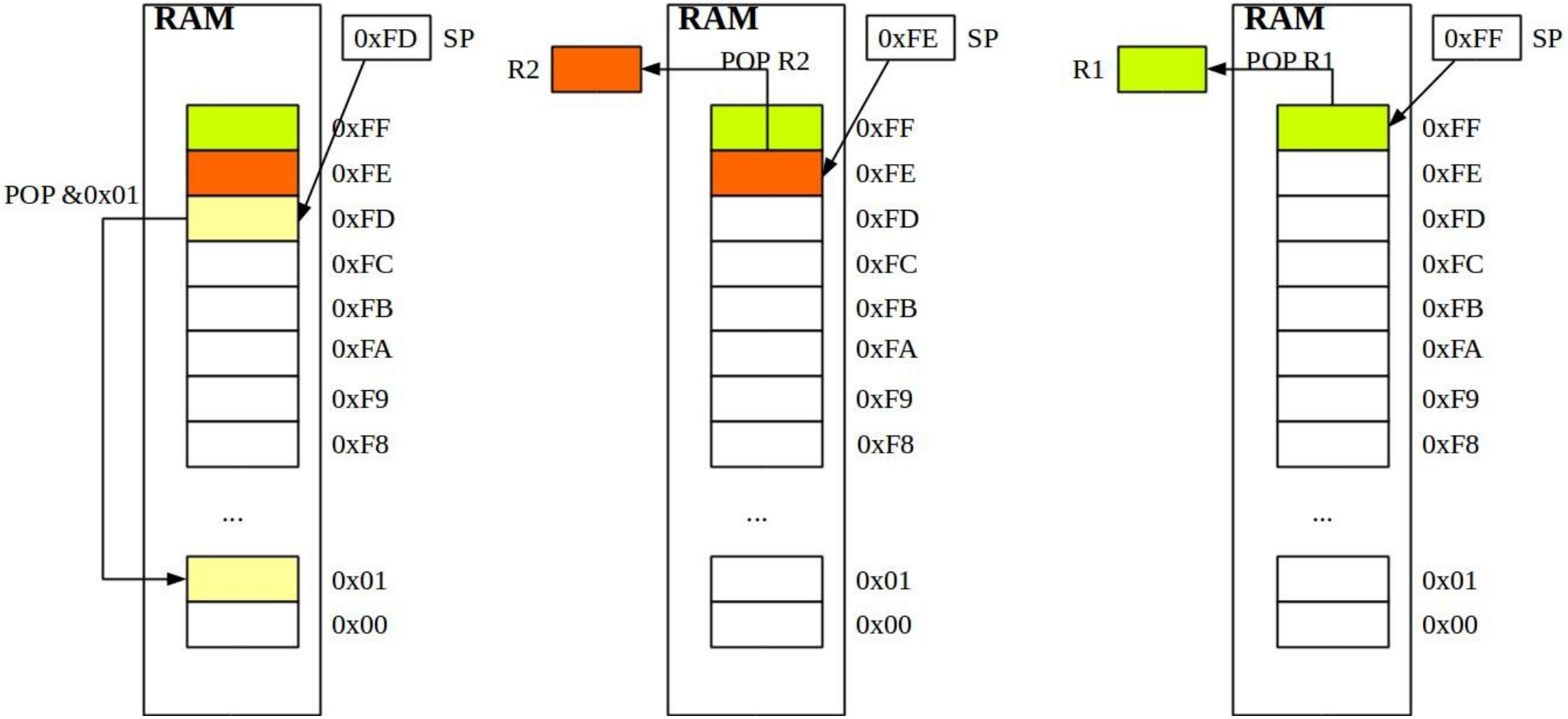
Програмен модел

Пример с намаляващ стек. Работа с освободените регистри – R1, R2, RAM-регистър на адрес 0x01.



Програмен модел

Пример с намаляващ стек. Четене на стека.



Програмен модел

Записът и четенето на данни в стека **винаги** става в **обратен ред**. Асемблерната програма, съответстваща на предишните слайдове би изглеждала така:

```
    ...
PUSH   R1
PUSH   R2
PUSH   &0x01
MOV    #0x55, R1
MOV    #0x33, R2
MOV    #0x12, &0x01
AND    R1, R10
XOR    R2, R11
OR     R11, &0x01
```

```
POP   &0x01
POP R2
POP   R1
...
```

Програмен модел

Служебната информация, която трябва да се помни в стека бива два вида:

* регистри от ядрото и оперативната памет, когато се извикват функции

* регистри от ядрото и оперативната памет, когато диспечер на операционна система превключва изпълнението на задачи

* регистри от ядрото и оперативната памет, когато се получи прекъсване

Програмен модел

Пример - регистри от ядрото и оперативната памет, когато се извикват функции. ABI стандарт на компилатора гласи – първите 4 регистъра са за предаване на параметри.

```
uint32_t my_func(uint32_t a, uint32_t b,
                 uint32_t c, uint32_t d){
    return (a + b) * (c + d);
}

int main(void){
    uint32_t a = 355, b = 743, c = 291, d = 8096;
    uint32_t e;

    e = my_func(a, b, c, d);

    while (1){ }
}
```

```
08000220 <main>:
 8000220:   b580        push   {r7, lr}
 8000222:   b086        sub    sp, #24
 8000224:   af00        add    r7, sp, #0
 8000226:   f240 1363   movw   r3, #355
 800022a:   617b        str    r3, [r7, #20]
 800022c:   f240 23e7   movw   r3, #743
 8000230:   613b        str    r3, [r7, #16]
 8000232:   f240 1323   movw   r3, #291
 8000236:   60fb        str    r3, [r7, #12]
 8000238:   f44f 53fd   mov.w  r3, #8096
 800023c:   60bb        str    r3, [r7, #8]
 800023e:   68bb        ldr    r3, [r7, #8]
 8000240:   68fa        ldr    r2, [r7, #12]
 8000242:   6939        ldr    r1, [r7, #16]
 8000244:   6978        ldr    r0, [r7, #20]
 8000246:   f7ff ffd7   bl     80001f8 <my_func>
 800024a:   6078        str    r0, [r7, #4]
 800024c:   e7fe        b.n    800024c <main+0x2c>
```

Програмен модел

Пример - регистри от ядрото и оперативната памет, когато се извикват функции. ABI стандарт на компилатора гласи – при повече от 4 параметъра – използвай стека.

```
uint32_t my_func(uint32_t a, uint32_t b,
uint32_t c, uint32_t d, uint32_t e){
        return (a + b) * (c + d) + e;
}

int main(void){
        uint32_t a = 355, b = 743, c = 291, d = 8096,
e = 3131;
        uint32_t f;

        f = my_func(a, b, c, d, e);

        while (1){ }
}
```

```
08000230 <main>:
 8000230:   b580        push   {r7, lr}
 8000232:   b088        sub    sp, #32
 8000234:   af02        add    r7, sp, #8
 8000236:   f240 1363   movwr3, #355
 800023a:   617b        str    r3, [r7, #20]
 800023c:   f240 23e7   movwr3, #743
 8000240:   613b        str    r3, [r7, #16]
 8000242:   f240 1323   movwr3, #291
 8000246:   60fb        str    r3, [r7, #12]
 8000248:   f44f 53fd   mov.w        r3, #8096
 800024c:   60bb        str    r3, [r7, #8]
 800024e:   f640 433b   movwr3, #3131
 8000252:   607b        str    r3, [r7, #4]
 8000254:   687b        ldr    r3, [r7, #4]
 8000256:   9300        str    r3, [sp, #0] ; == PUSH
 8000258:   68bb        ldr    r3, [r7, #8]
 800025a:   68fa        ldr    r2, [r7, #12]
 800025c:   6939        ldr    r1, [r7, #16]
 800025e:   6978        ldr    r0, [r7, #20]
 8000260:   f7ff ffd0   bl     8000204 <my_func>
 8000264:   6038        str    r0, [r7, #0]
 8000266:   e7fe        b.n    8000266 <main+0x36>
```

Програмен модел

Пример - регистри от ядрото и оперативната памет, когато се получи прекъсване. Регистрите, използвани в хендлера се копират на стека.

```
void exti0_irq_handler(void){
	uint32_t i = 0;
	uint32_t arr[8] = { 55, 12, 89012,
65390, 838107, 785, 912, 1024};
	uint32_t a = 355, b = 743, c = 291,
d = 8096, e = 3131;

	for(i = 0; i < 8; i++){
		a = ((a + b) * (c + d) + e)/arr[i];
	}

	a = a; //Do nothing
}

int main(void){
	while (1){ }
}
```

```
080001f8 <exti0_irq_handler>:
 80001f8:	b4b0      	push	{r4, r5, r7}
 80001fa:	b08f      	sub	sp, #60
 80001fc:	af00      	add	r7, sp, #0
 80001fe:	2300      	movs	r3, #0
 8000200:	637b      	str	r3, [r7, #52]
 8000202:	4b1d      	ldr	r3, [pc, #116]
 8000204:	463c      	mov	r4, r7
 8000206:	461d      	mov	r5, r3
 8000208:	cd0f      	ldmia	r5!, {r0, r1, r2, r3}
 800020a:	c40f      	stmia	r4!, {r0, r1, r2, r3}
 800020c:	e895 000f 	ldmia.w	r5, {r0, r1, r2, r3}
 8000210:	e884 000f 	stmia.w	r4, {r0, r1, r2, r3}
 8000214:	f240 1363 	movwr3, #355
 ……..
 800026a:	d9e5      	bls.n	8000238
 800026c:	bf00      	nop
 800026e:	373c      	adds	r7, #60
 8000270:	46bd      	mov	sp, r7
 8000272:	bcb0      	pop	{r4, r5, r7}
 8000274:	4770      	bx	lr
 8000276:	bf00      	nop
```

Програмен модел

ВНИМАНИЕ! При влизане във функция, обслужваща прекъсване има регистри, които се копират на стека автоматично (хардуерно). Множеството от тези регистри се наричат **стекова група** (**stack frame**).

При MSP430 стековата група е съставена от PC, SR.
При ARM Cortex-M стековата група е съставена от PSR, PC, R14, R12, R3, R2, R1, R0.

Виж миналия слайд – дисасемблерът е за ARM Cortex-M7. Липсват инструкции за копиране на регистри R0 – R3, защото те са копирани ***хардуерно****.*

Програмен модел

Регистър на състоянието (**S**tatus **R**egister или още **C**ondition **C**ode **R**egister) – съдържа битове, които отразяват състоянието на АЛУ (и други функционални блокове) след изпълнението на всяка една инструкция.

V (oVerflow) – число със знак е обърнало знакът си. (-128 ←→ 127, -32768 ←→ 32767 и т.н.)

N (Negative) – ако резултатът от операцията е отрицателно число, то N = 1; ако е положително, N = 0.

Z (Zero) - ако резултатът от операцията е нула, то Z = 1; ако е ненулева стойност, то Z = 0;

C (Carry) – ако резултатът от операцията е предизвикал пренос, то C = 1; ако няма пренос, то C = 0. (255 + 1 = 0 (1), и т.н.)

Програмен модел

ВНИМАНИЕ! Регистрите PC, SP и SR могат да повлияят на хода на изпълнението на програмата. Въпреки, че са част от регистровия файл, те не бива да бъдат използвани като GPR. Допълнително объркване може да се получи и от алтернативните им имена, които имат аналогични обозначения (с R или r) с тези на GPR.

* При MSP430 – PC = R0, SP = R1, SR = R2

* При ARM Cortex – PC = R15, SP = R13

Видове инструкции

Всяка инструкция се състои от един или няколко байта.

В тези байтове има групи от битове (битови полета), които имат различно значение за микропроцесора. Най-общо казано инструкцията може да се раздели на две части:

***код на операцията** (КОП, на англ. - opcode) – операцията, която трябва да извърши микропроцесора върху операндите

***служебни полета** – вид на достъп до паметта (напр. 8- или 16-битов), номер на регистър от ядрото, вид на адресацията, бит за опресняване на статус регистъра и др.

Видове инструкции

Пример с инструкция на MSP430:

mov.w r5, r4

Двоичен вид: 0100 0101 0000 0100

КОП	**Номер на регистър-източник**	**Вид адресация на регистъра-приемник**	**Вид достъп до паметта**	**Вид адресация на регистъра-източник**	**Номер на регистър-приемник**
0100	0101	0	0	00	0100
mov	r5	регистрова адресация	.w	регистрова адресация	r4

Видове инструкции

1. Според дължината си, т.е. броя байтове, заемани в паметта на системата, инструкциите биват два вида:

*с **фиксирана дължина** → GPP, DSP, μPU (например 8-, 16- или 32-битови инструкции)

*с **променлива дължина** → VLIW процесори (дължината зависи от конкретната програма)

ВНИМАНИЕ! При някои микропроцесори операндите могат да се намират между инструкциите (MSP430), а при други – след тях (ARM Cortex).

Видове инструкции


```
26276: 480d      ldr     r0, [pc, #52]
26278: 2800      cmp     r0, #0
2627a: d002      beq.n   26282
2627c: 480c      ldr     r0, [pc, #48]
2627e: f3af 8000 nop.w
26282: f012 fe4b bl      38f1c
26286: 0020      movs    r0, r4
26288: 0029      movs    r1, r5
2628a: f00e f8c1 bl      34410
2628e: f012 fe31 bl      38ef4
26292: bf00      nop
26294: 00080000
26298: 20010000
262a4: 20002c28
262a8: 20004a00
```

Регион за константи (literal pool)

ARM Cortex-M0

```
c40e: 0c 43          clr   r12
c410: b2 d0 03 00    bis   #3,   &0x01b0
c414: b0 01
c416: 32 d0 18 00    bis   #24, r2
c41a: 1c 52 b4 01    add   &0x01b4,r12
c41e: 37 53          add   #-1,  r7
c420: 07 93          cmp #0,   r7
c422: f6 23          jnz   $-18
```

MSP430

Видове инструкции

2. Според вида на изпълняваната операция инструкциите биват:

* за достъп до паметта (=)
* за аритметични операции (+, -, *, /)
* за логически операции (&, |, ~, ^)
* за условни и безусловни преходи (if, goto)
* за аритметика с насищане (if(i > N){i = N;})
* разни (nop, wfe, clra)

Режими на адресация

Съществуват няколко различни режима, с които дадена инструкция може да извлече данните за операндите си. В зависимост от тези режими се казва, че инструкцията използва една или друга адресация (addressing mode).

Режимите на адресации е възможно да са различни за всеки един операнд в рамките на една инструкция.

Видовете адресации зависят от набора инструкции и следователно са различни за различните микропроцесори.

Режими на адресация

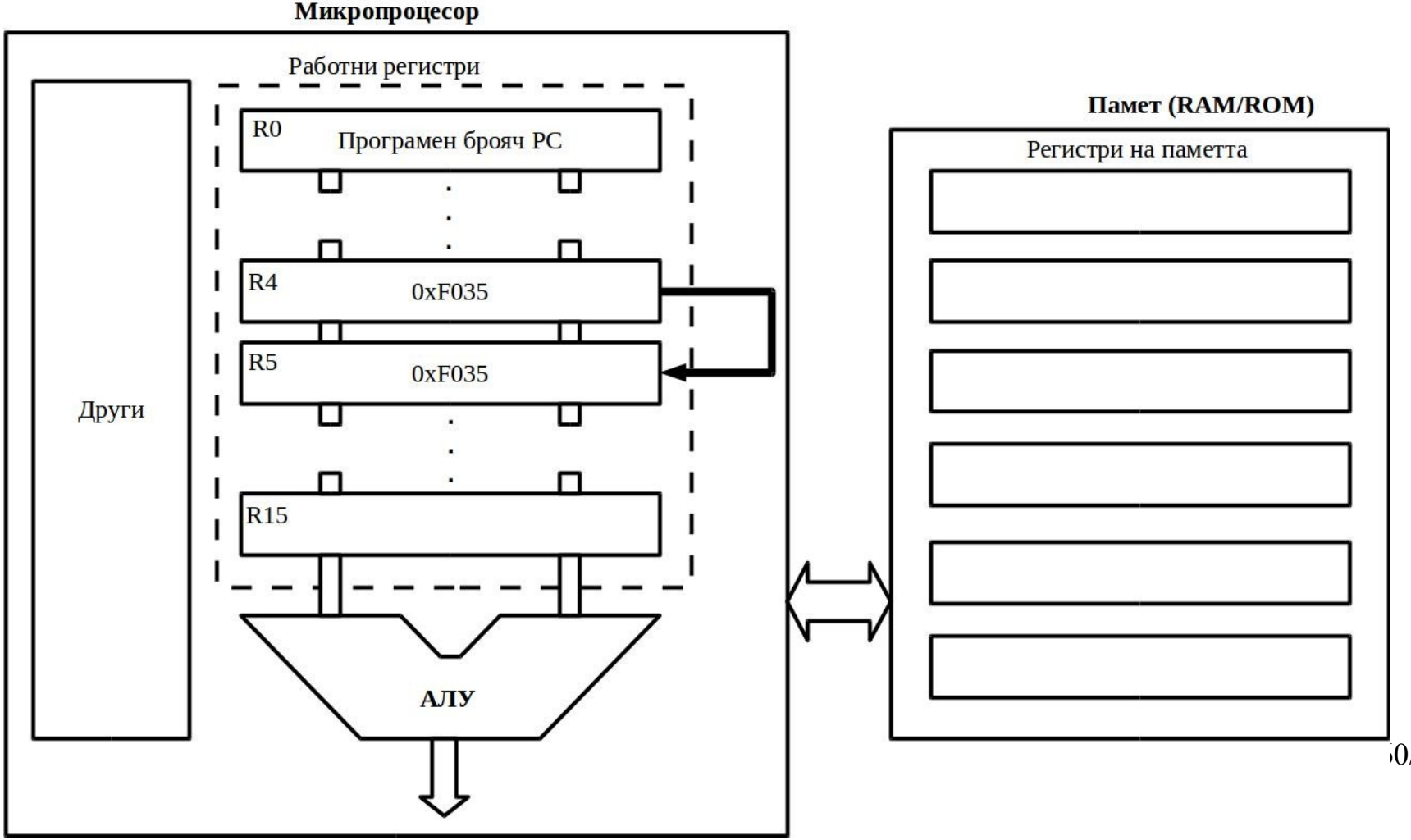
На следващите слайдове са показани адресациите за за MSP430, но те дават обща представа за механизма на “извличане” на операндите и в много други микропроцесори се използват същите и/или подобни адресации (имената им може да варират при различните фирми производители).

За MSP430 това са:

1. Регистрова
2. Индексна
3. Относителна
4. Абсолютна
5. Индиректна
6. Индиректна автоинкрементираща
7. Непосредствена

Режими на адресация

1. **Регистрова адресация** – съдържанието на регистър/регистри от ядрото са операнд/операндите на инструкцията. (mov.w R4, R5)



Режими на адресация

1. Регистрова адресация – пример с инструкцията mov.w R4,R5.

Преди:
R4 = 0xF035
R5 = 0x5555

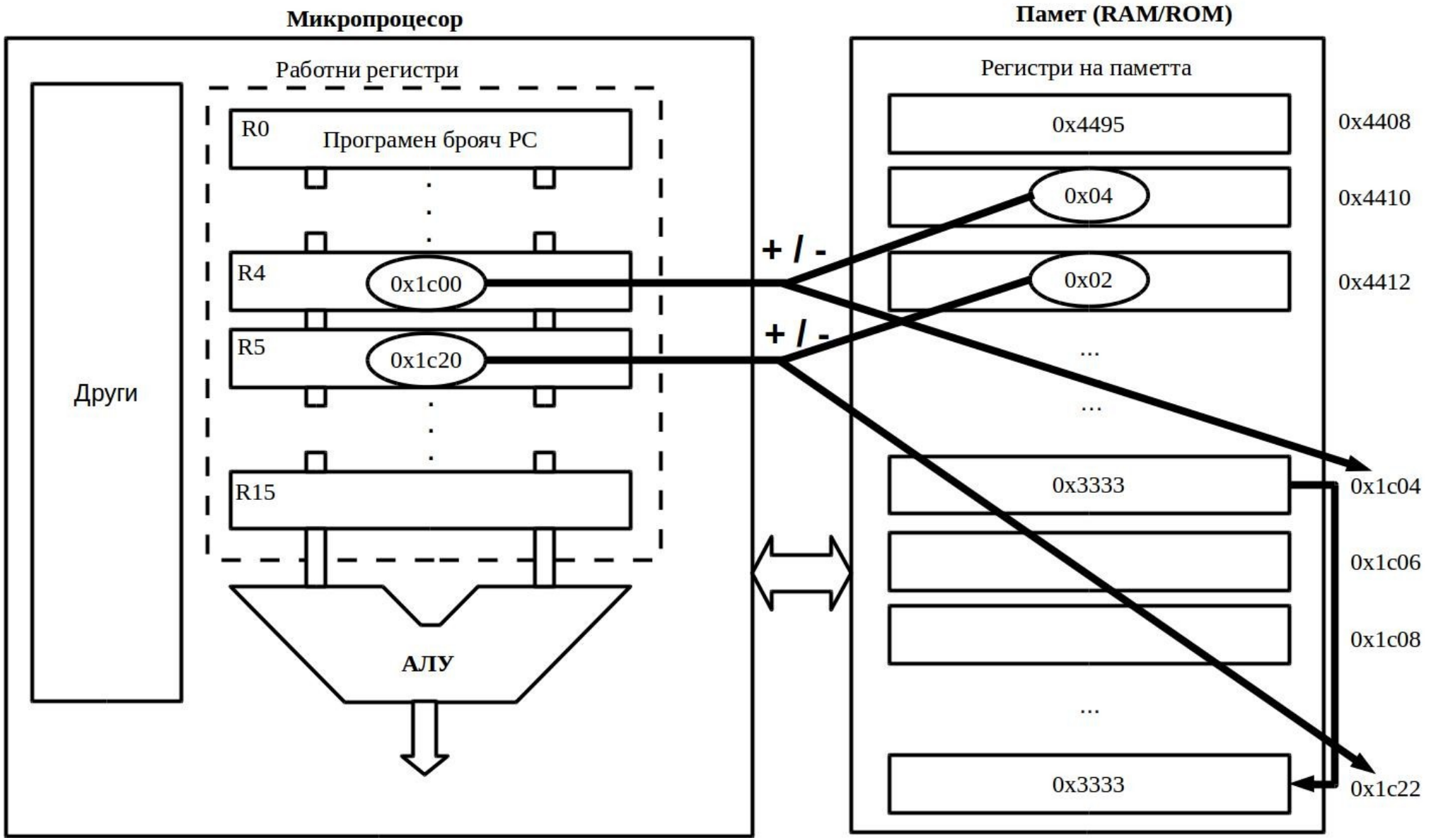
След:
R4 = 0xF035
R5 = 0xF035

Режими на адресация

2. **Индексна адресация** (Indexed mode) – съдържанието на работен регистър от ядрото + съдържанието на регистър от програмната памет (отместване) указват адреса (някъде в паметта на микропроцесора) на операнда.

Отместването е число със знак и може да бъде както положително, така и отрицателно.

Режими на адресация



Режими на адресация

2. Индексна адресация – пример с инструкцията mov.w 0x04(R4), 0x02(R5). Машинен код:
0x4495 0x0004 0x0002

Преди:
R4 = 0x1c00
R5 = 0x1c20
MEM(0x1c04) = 0x3333
MEM(0x1c22) = 0x5555

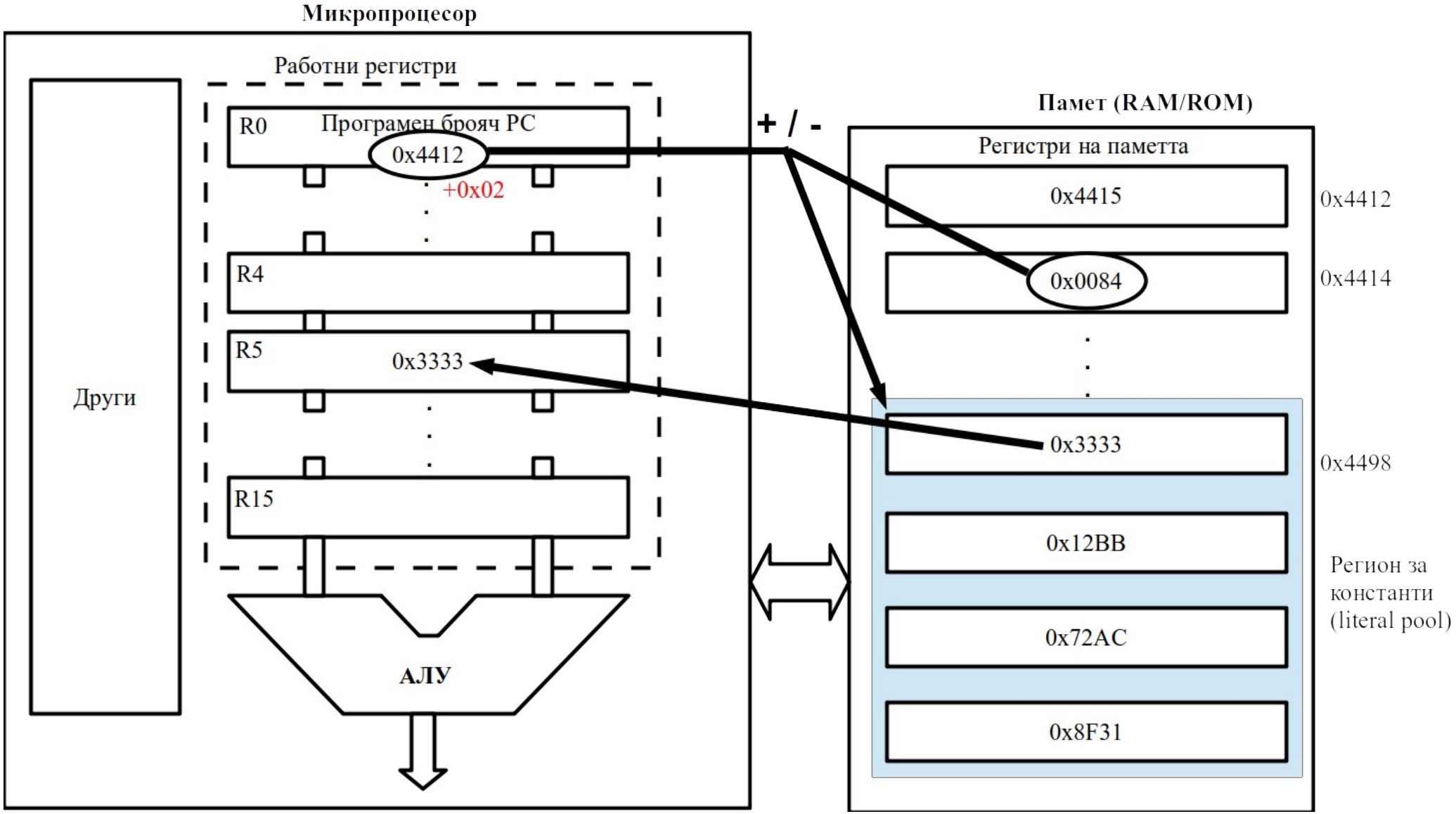
След:
R4 = 0x1c00
R5 = 0x1c20
MEM(0x1c04) = 0x3333
MEM(0x1c22) = 0x3333

Режими на адресация

3. **Относителна адресация** (Symbolic mode, PC-relative mode) – съдържанието на регистър от програмната памет (отместване) + стойността на програмния брояч PC указват адреса на операнда. Тази адресация е аналогична на индексната, с тази разлика, че се използва не-кой да е регистър от ядрото, а точно програмния брояч.

Регион за константи (literal pool) - региони от паметта, запълнени с константи, които се достъпват чрез символна адресация. Обикновено се поместват след края на подпрограмата (функцията), в която се използват.

Режими на адресация



Режими на адресация

3. Относителна адресация – пример с инструкцията

```
mov     my_const_0, R5.
…
my_const_0:
    DW 0xaaaa
```

Машинен код:

0x4415 0x0084

Режими на адресация

Преди:
R5 = 0x0000

mov my_const_0, R5 = 0x4412: 0x4415 0x0084
(PC -> 0x4412)
...
my_const_0:
DW 0xaaaa = 0x4498: 0xaaaa

0x4412(настоящата стойност PC) + 0x84 (офсет спрямо PC) + 2 (конвейер) = 0x4498

След:
R5 = 0xaaaa

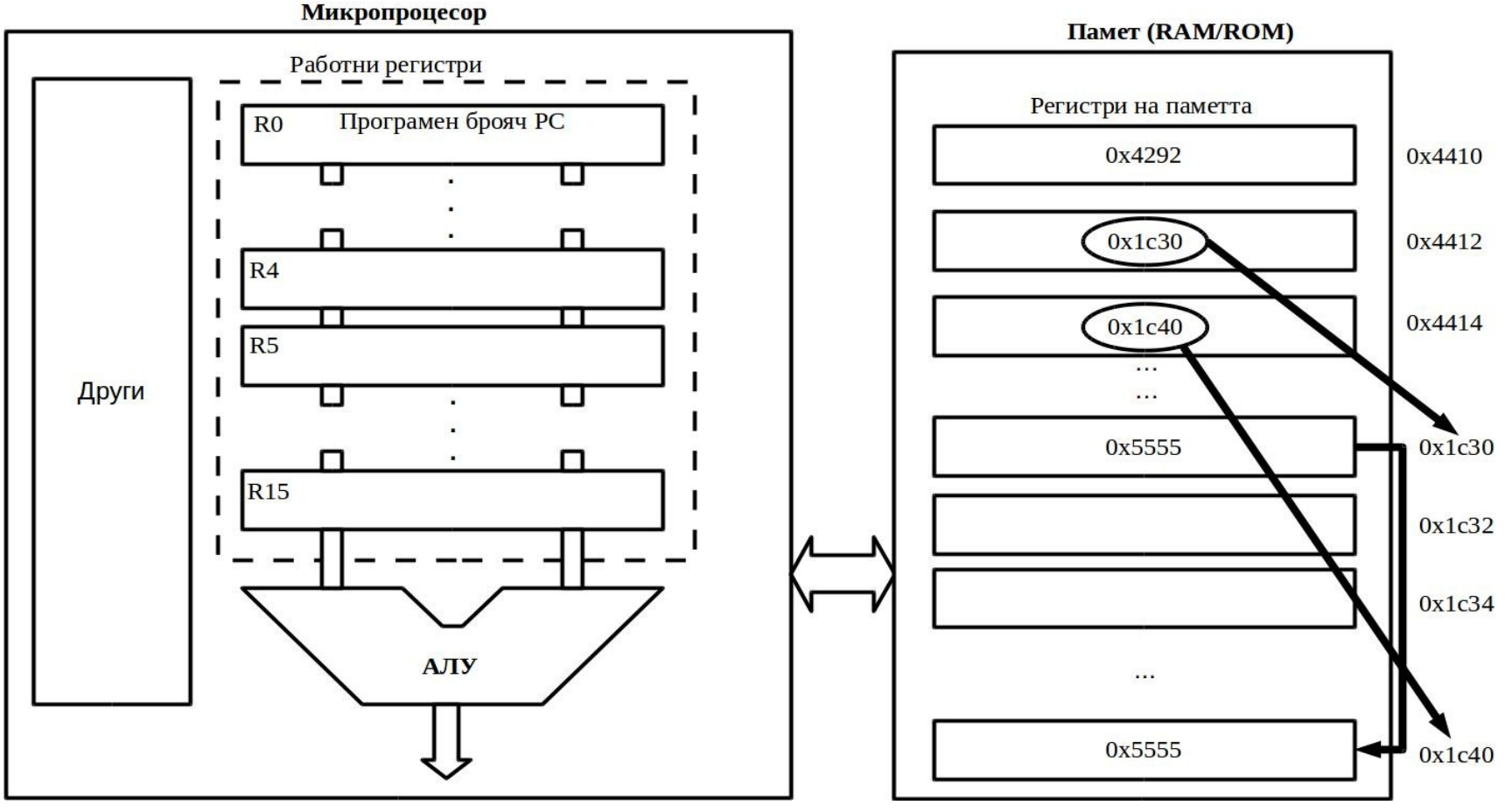
Режими на адресация

3. Относителна адресация – пример с няколко последователни инструкции

```
mov     my_const_0, R5          =       4412: 4415 0084
mov     my_const_1, R5          =       4416: 4415 0082
mov     my_const_2, R5          =       441a: 4415 0080
mov     my_const_3, R5          =       441e: 4415 007e
        …                                       …
my_const_0:
        DW 0xaaaa               =       4498: aaaa
my_const_1:
        DW 0xbbbb               =       449a: bbbb
my_const_2:
        DW 0xcccc               =       449c: cccc
my_const_3:
        DW 0xdddd               =       449e: dddd
```

Режими на адресация

4. **Абсолютна адресация** (Absolute mode) - съдържанието на регистър от програмната памет указва абсолютен адрес от паметта, на който се намира операнда.



Режими на адресация

4. Абсолютна адресация – пример с инструкцията mov &0x1c30, &0x1c40. Машинен код:
0x4292 0x1c30 0x1c40

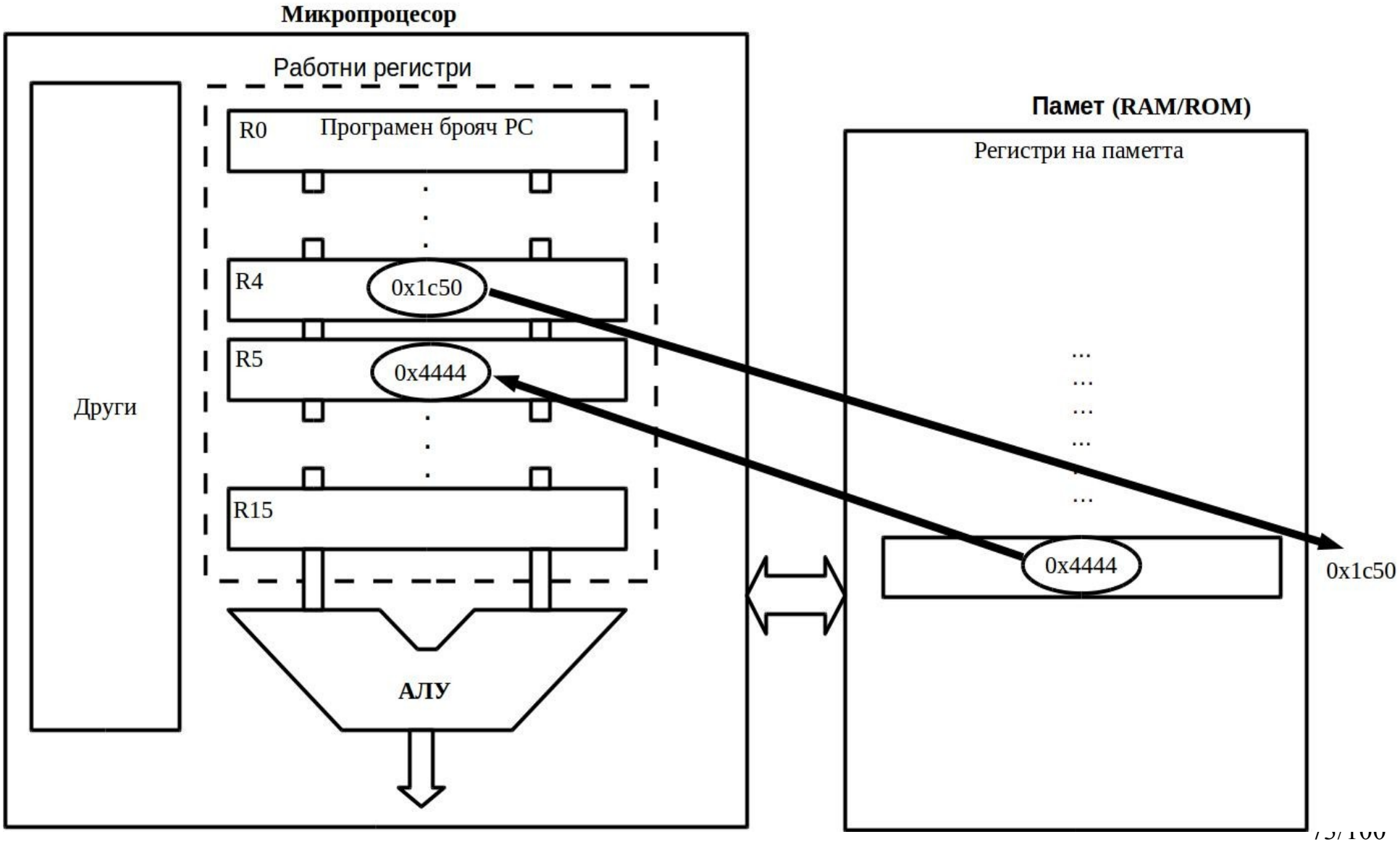
Преди:
MEM(0x1c30) = 0x5555
MEM(0x1c40) = 0xf035

След:
MEM(0x1c30) = 0x5555
MEM(0x1c40) = 0x5555

Режими на адресация

5. Индиректна адресация (Indirect mode) – аналогична на абсолютната. Разликата – съдържанието на работен регистър от ядрото указва абсолютен адрес от паметта, на който се намира операнда.

Режими на адресация



Режими на адресация

5. Индиректна адресация – пример с инструкцията mov @R4, R5. Машинен код:

0x4425

Преди:

R4 = 0x1c50

R5 = 0xfefe

MEM(0x1c50) = 0x4444

След:

R4 = 0x1c50

R5 = 0x4444

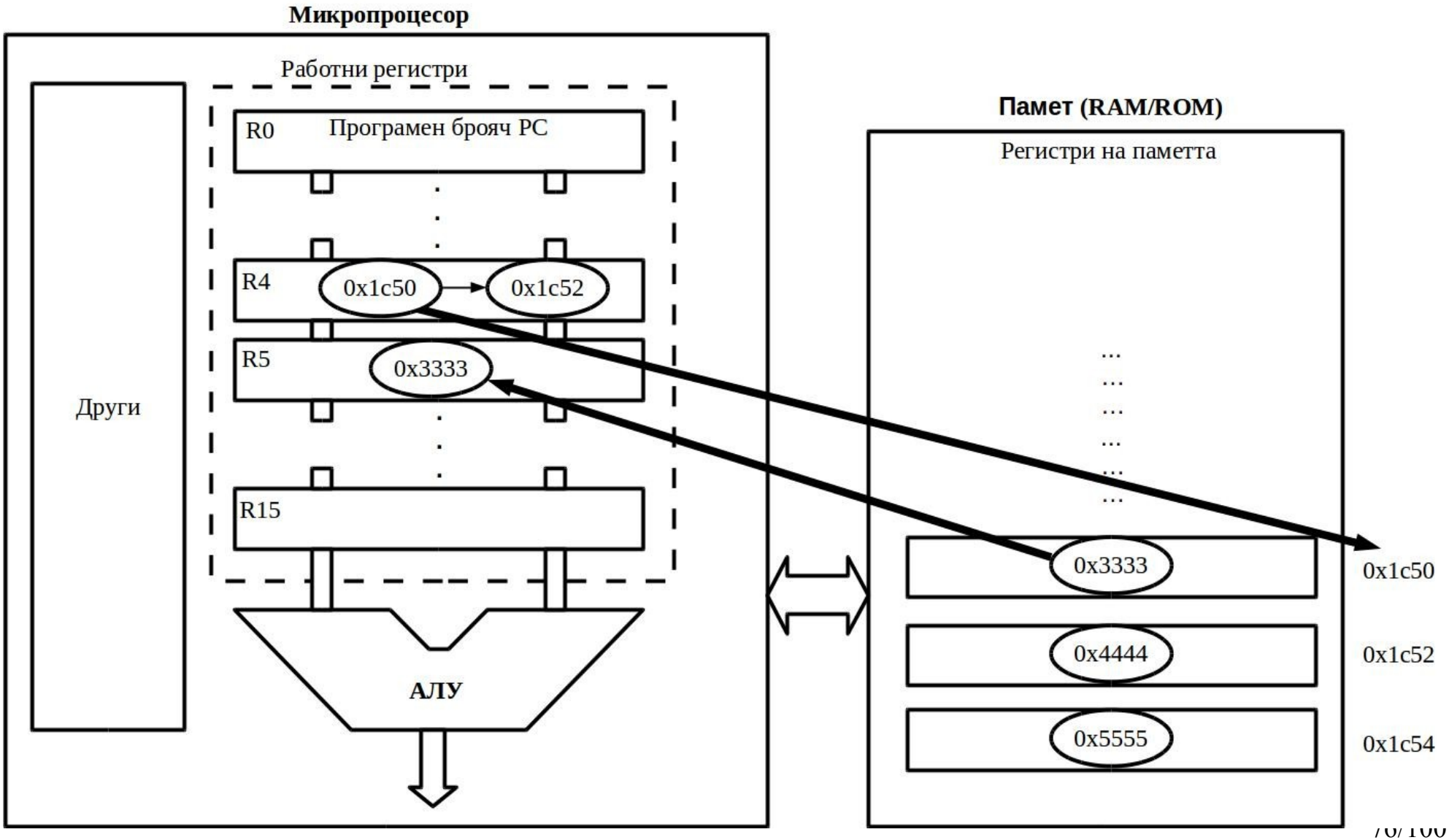
MEM(0x1c50) = 0x4444

Режими на адресация

6. Индиректна автоинкрементираща адресация (Indirect autoincrement mode) – съдържанието на работен регистър от ядрото указва абсолютен адрес от паметта, на който се намира операнда. След изпълнението на инструкцията, съдържанието на работния регистър се увеличава автоматично.

* Ако инструкцията е с наставка “.b”, адресът се увеличава с 1.
* Ако инструкцията е с наставка “.w”, адресът се увеличава с 2.

Режими на адресация



Режими на адресация

6. Индиректна автоинкрементираща адресация – пример с инструкцията mov.w @R4+, R5. Машинен код:

0x4435

Преди:

R4 = 0x1c50

R5 = 0xfefe

MEM(0x1c50) = 0x4444

След:

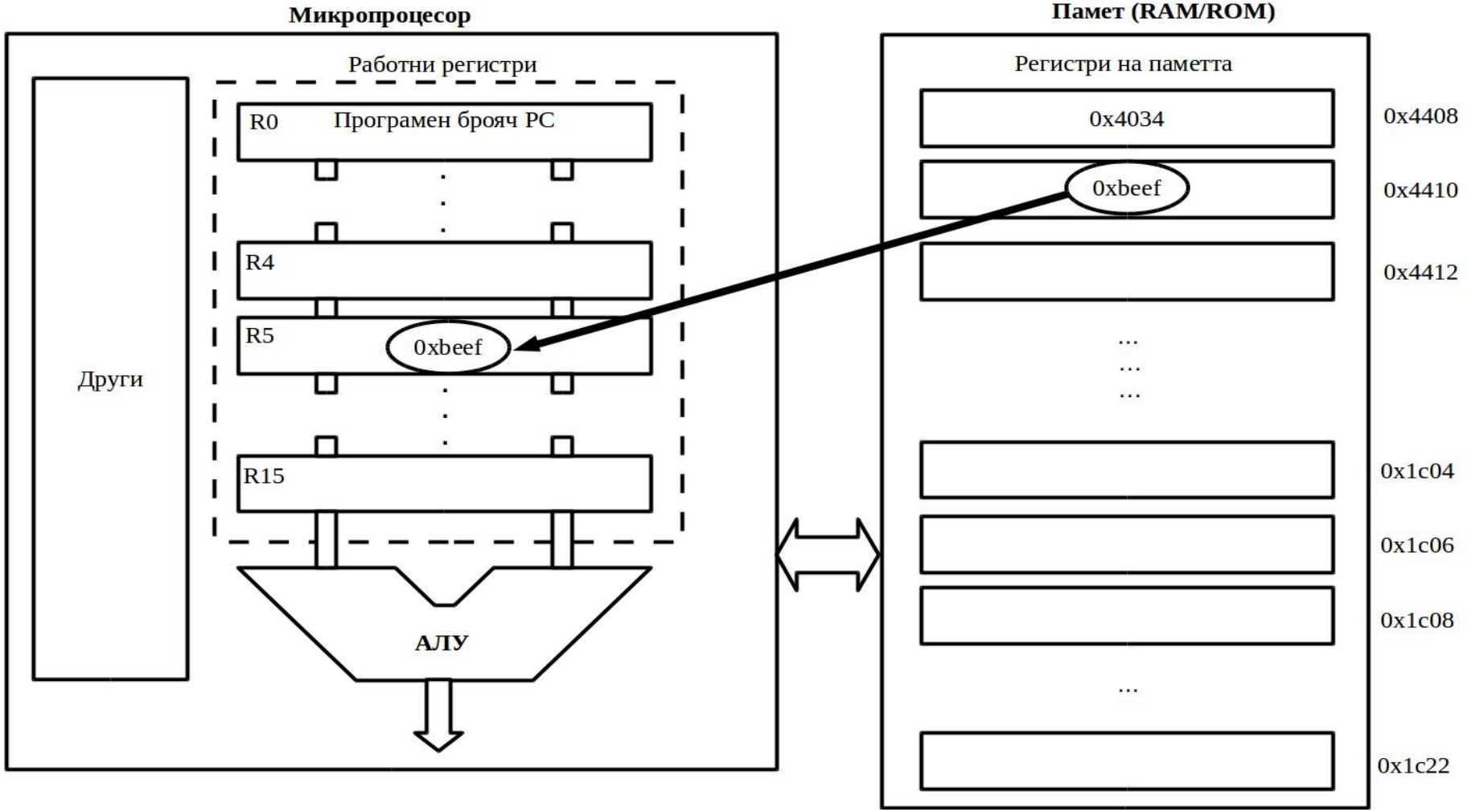
R4 = 0x1c52

R5 = 0x4444

MEM(0x1c50) = 0x4444

Режими на адресация

7. Непосредствена адресация (Immediate mode) – съдържанието на регистър от програмната памет (константа) е операнд на инструкцията.



Режими на адресация

7. Непосредствена адресация – пример с инструкцията mov #0xbeef, R4.

Преди:
R4 = 0xbebe

След:
R4 = 0xbeef

Режими на адресация

В зависимост от **поддържаните режими на адресация**, **архитектурите** на микропроцесорите могат да се категоризират като [8]:

* Регистър-регистър
* Регистър-памет
* Регистър-плюс-памет

Режими на адресация

Архитектурата от вида **регистър-регистър** (от англ. ез. register-register architecture) може да извършва операции само с регистри от ядрото. Известна е още като запиши-прочети, от англ. ез. Load-store architecture.

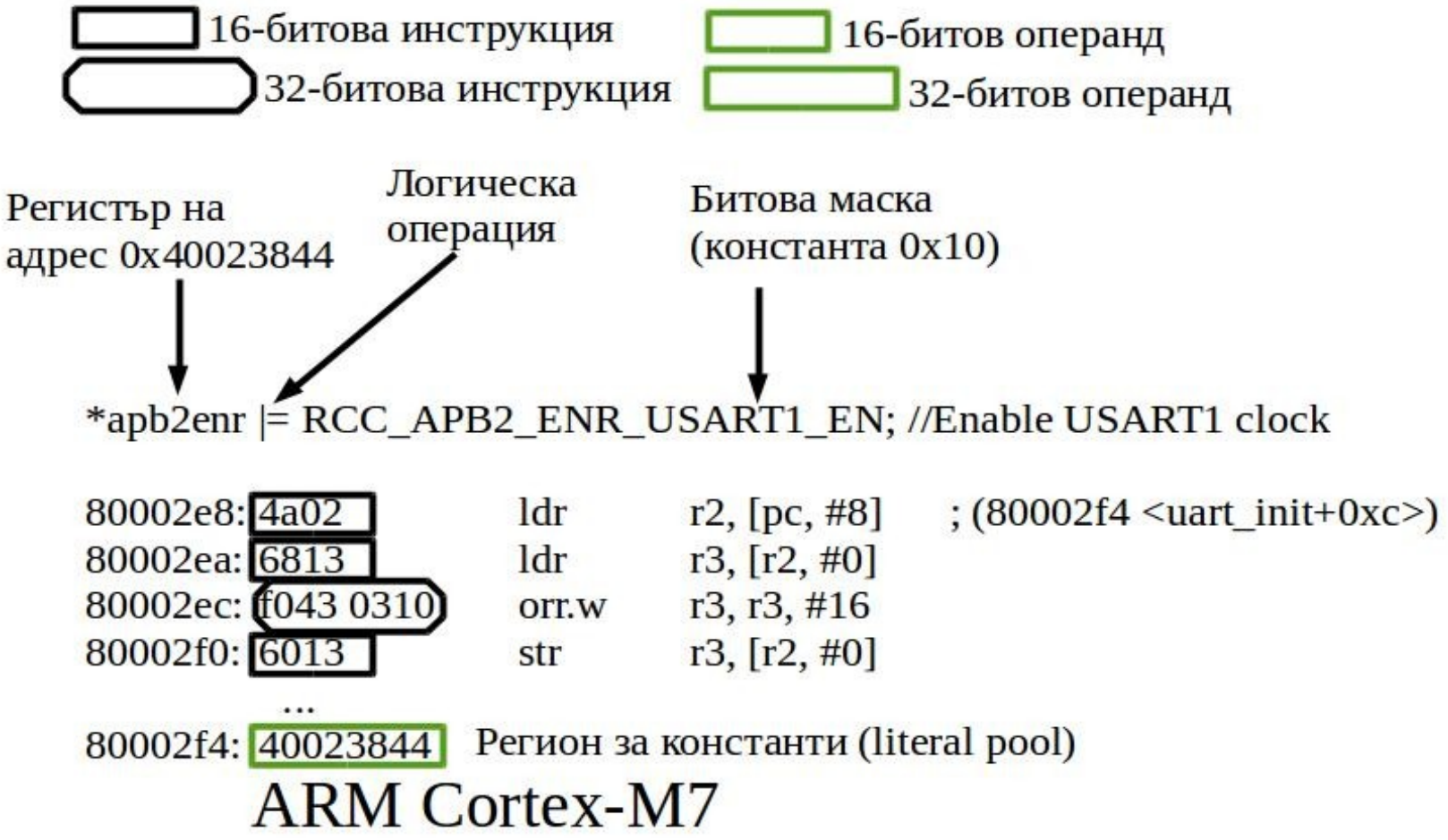
Това означава, че инструкциите могат да извършват операции само върху числа, записани в регистри на ядрото.

От друга страна, въпросните числа се записват в регистрите посредством две специални инструкции –

* **load** (копиране на числа от паметта в регистрите на ядрото)
* **store** (копиране на числа от регистрите на ядрото в паметта).

Режими на адресация

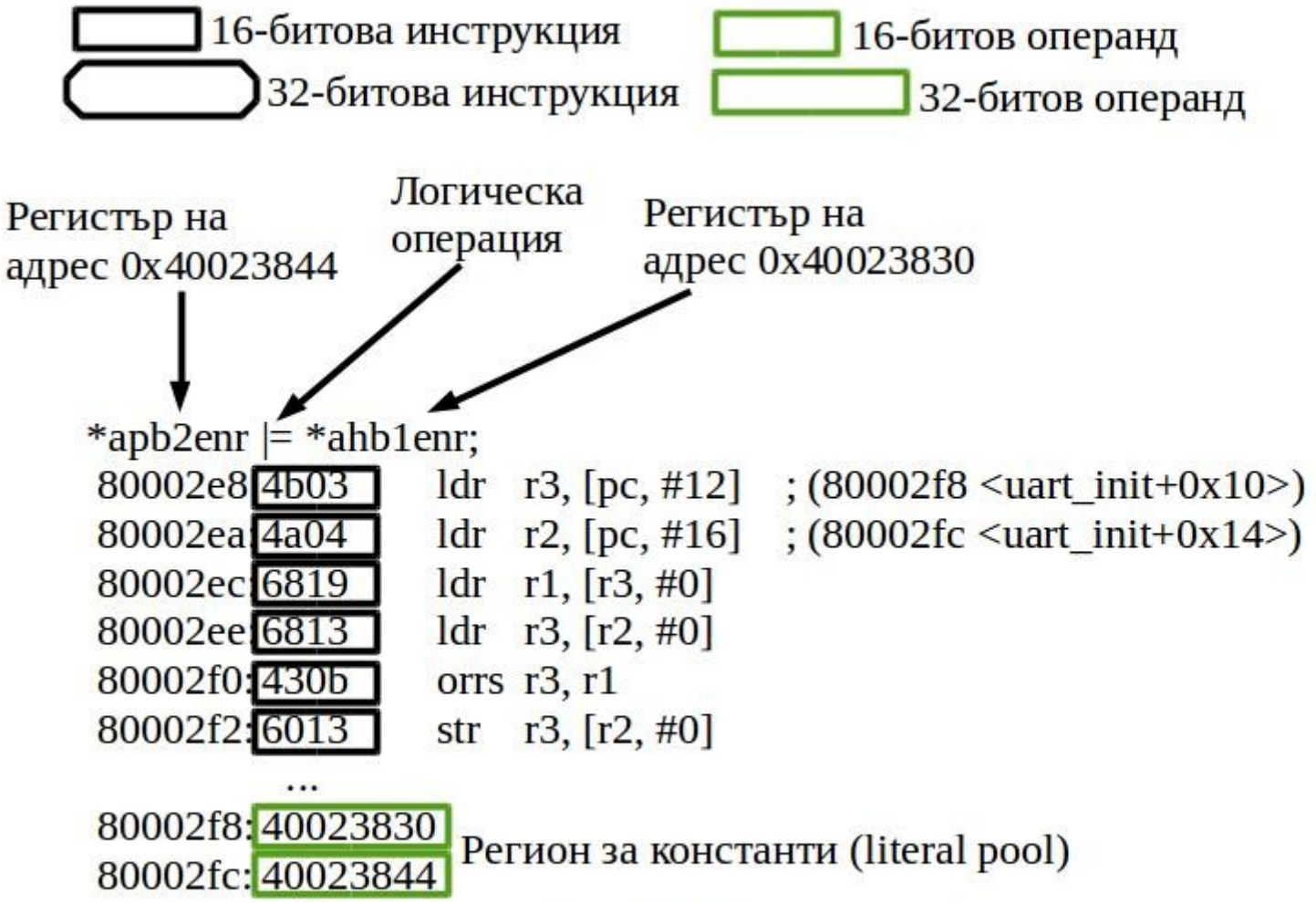
Пример – микропроцесорите ARM Cortex-M са с load-store архитектура. Ето как един ARM Cortex-M7 ще приложи логическо ИЛИ на един регистър от паметта с една константа:



ARM Cortex-M7

Режими на адресация

Пример – ето как един ARM Cortex-M7 ще приложи логическо ИЛИ на два регистъра от паметта:



Режими на адресация

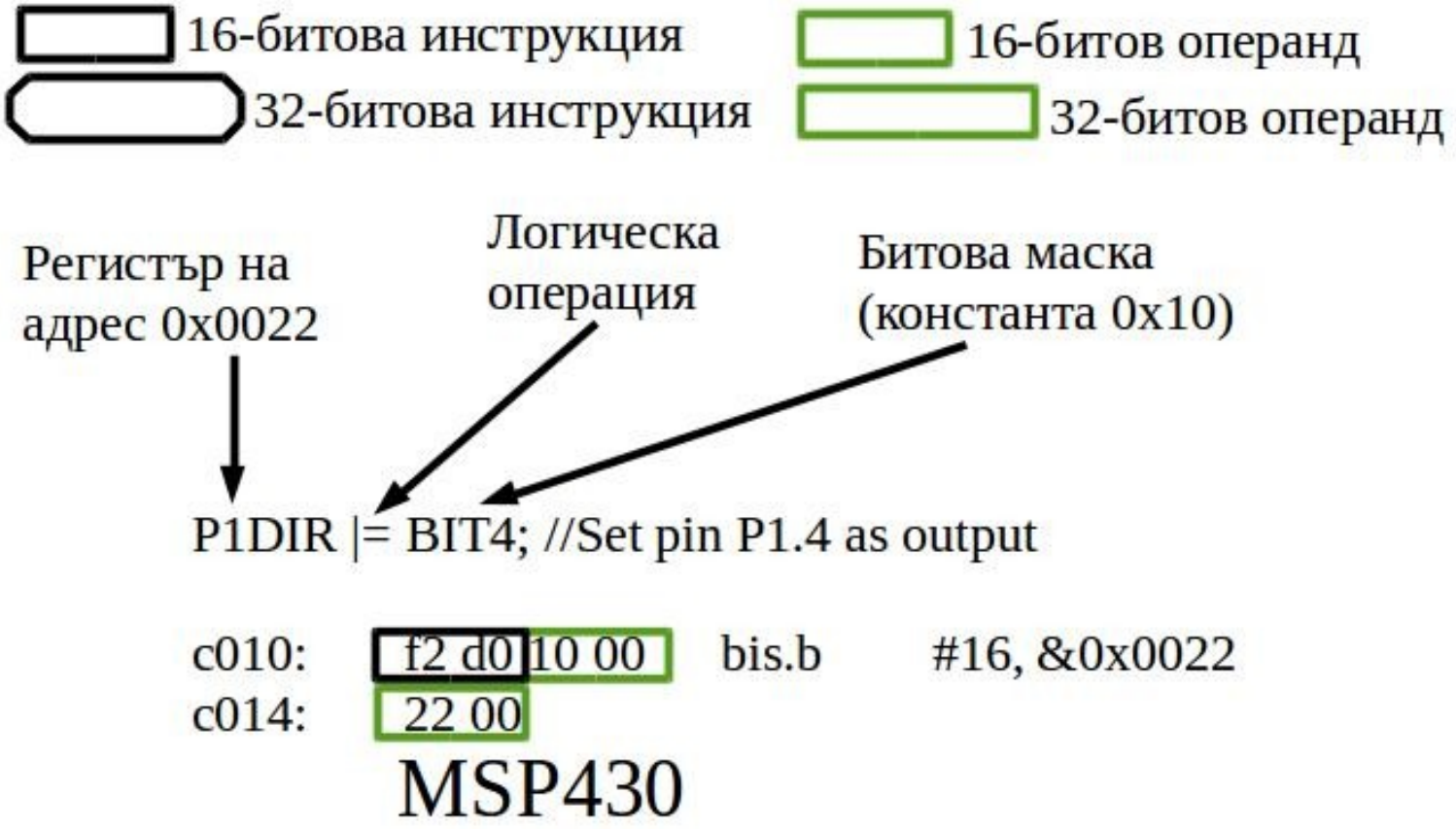
Архитектурата от вида **регистър-плюс-памет** (от англ. ез. register plus memory architecture) може да извършва операции с

* регистри от ядрото
* регистри от паметта
* комбинация регистри от ядро+памет .

Това означава, че инструкциите могат да извършват операции върху числа, записани само в регистри на ядрото, или само в регистри на паметта, или в комбинация от двете.

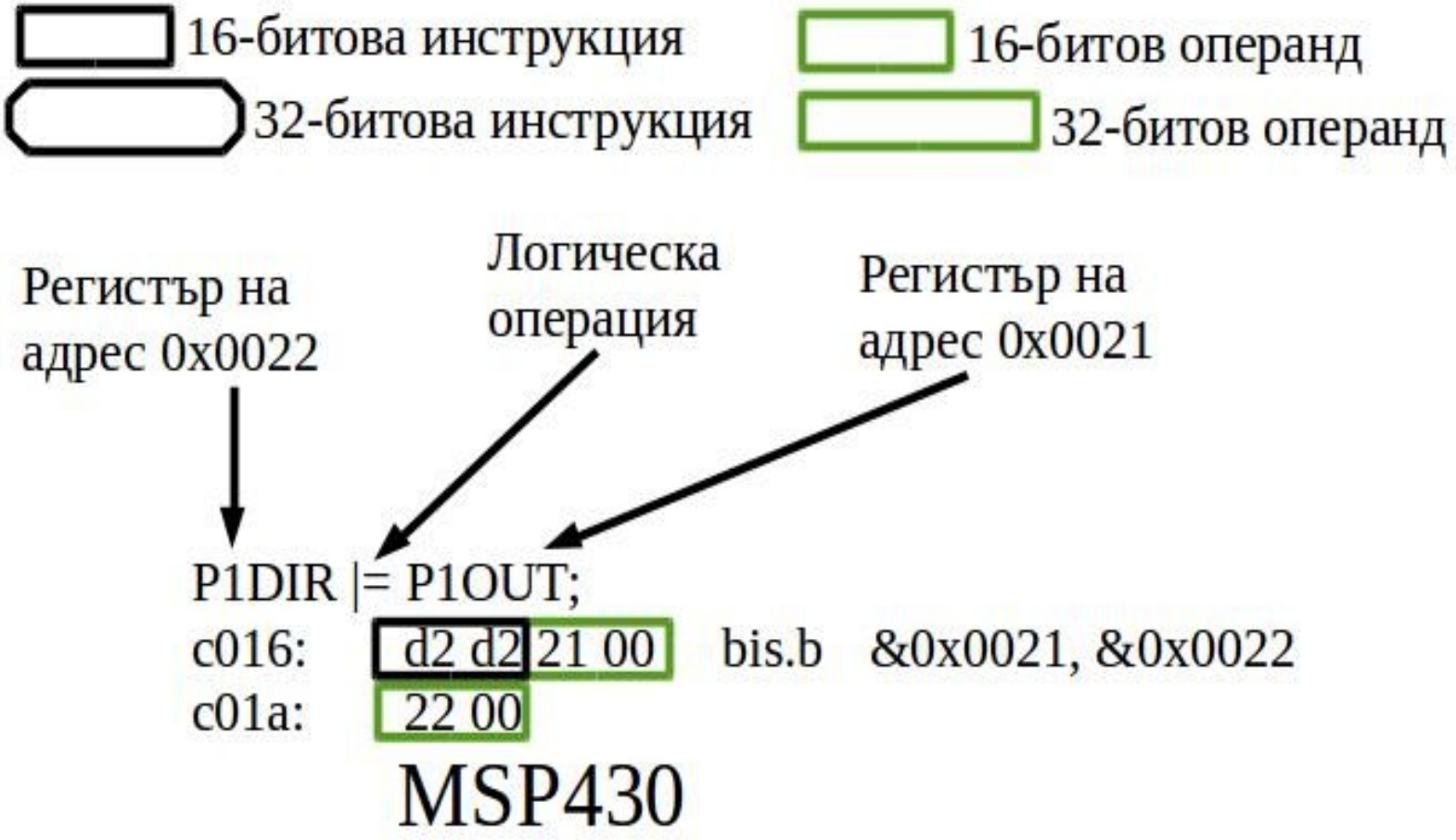
Режими на адресация

Пример – микропроцесорите MSP430 са с регистър-плюс-памет архитектура. Ето как един MSP430 ще приложи логическо ИЛИ на един регистър от паметта с константа:



Режими на адресация

Пример – ето как един MSP430 ще приложи логическо ИЛИ на два регистъра от паметта:



Режими на адресация

Пример – ето как един MSP430 ще приложи логическо И на две числа:

* с регистър от ядрото и с регистър от паметта

```
c000:  82 f5 22 00    and   r5, &0x0022
```

* с два регистъра от ядрото

```
c010:  06 f5          and   r5, r6
```

* с два регистъра от паметта

```
c012:  92 f2 24 00    and   &0x0024, &0x0022
c016:  22 00
```

Режими на адресация

Архитектурата от вида **регистър-памет** (от англ. ез. register-memory architecture) може да извършва операции с

* регистри от ядрото
* комбинация регистри от ядро+памет (но **резултатът** винаги се записва в **регистър от ядрото**).

Това означава, че инструкциите могат да извършват операции върху числа, записани в два регистъра на ядрото, или един регистър от паметта и един регистър от ядрото. Тези архитектури също използват load/store инструкции за достъп до паметта.

Пример – микропроцесорите Intel x86 са от този вид.

Карта на паметта

Адресно поле на един микропроцесор е съвкупността от стойности, които той може да установи на адресната си шина.

Когато микропроцесорът зададе число на адресната си шина се казва, че микропроцесорът **адресира** периферия/памет.

Регистрите на периферните устройства/паметите трябва да приемат адреси, които са от различни числови обхвати. Това гарантира, че комуникацията с тях няма да се обърка - всяка периферия ще има свой уникален адрес (или обхват от адреси) в адресното поле.

Карта на паметта

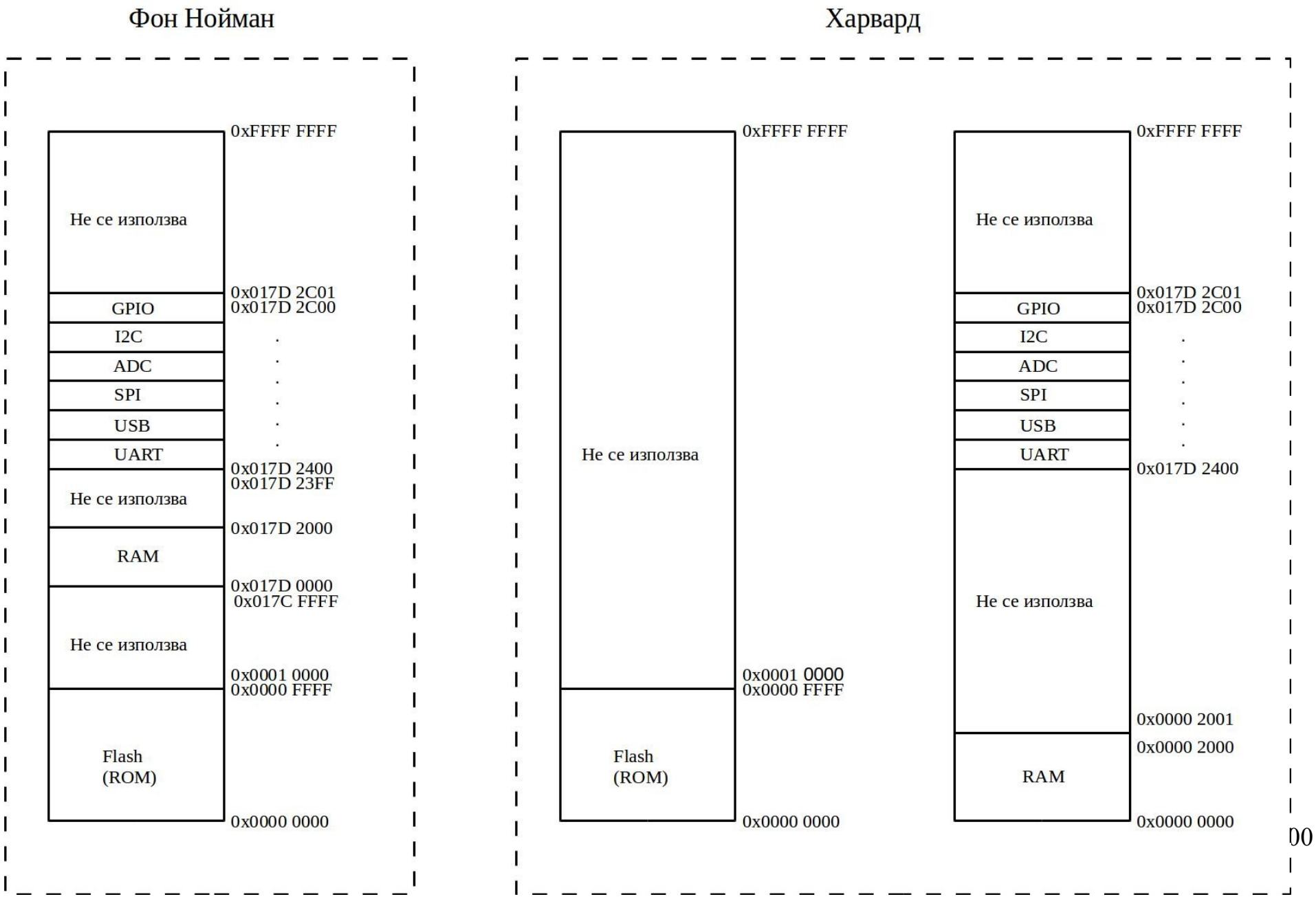
Карта на паметта е графичното представяне на адресното поле на микропроцесора и обхватите от адреси на отделните периферни устройства.

Фон Нойман архитектурите се характеризират с едно адресно поле (една карта на паметта).

Харвард архитектурите притежават две адресни полета (две карти на паметта).

На следващия слайд са показани примери за такива карти на паметта.

Карта на паметта



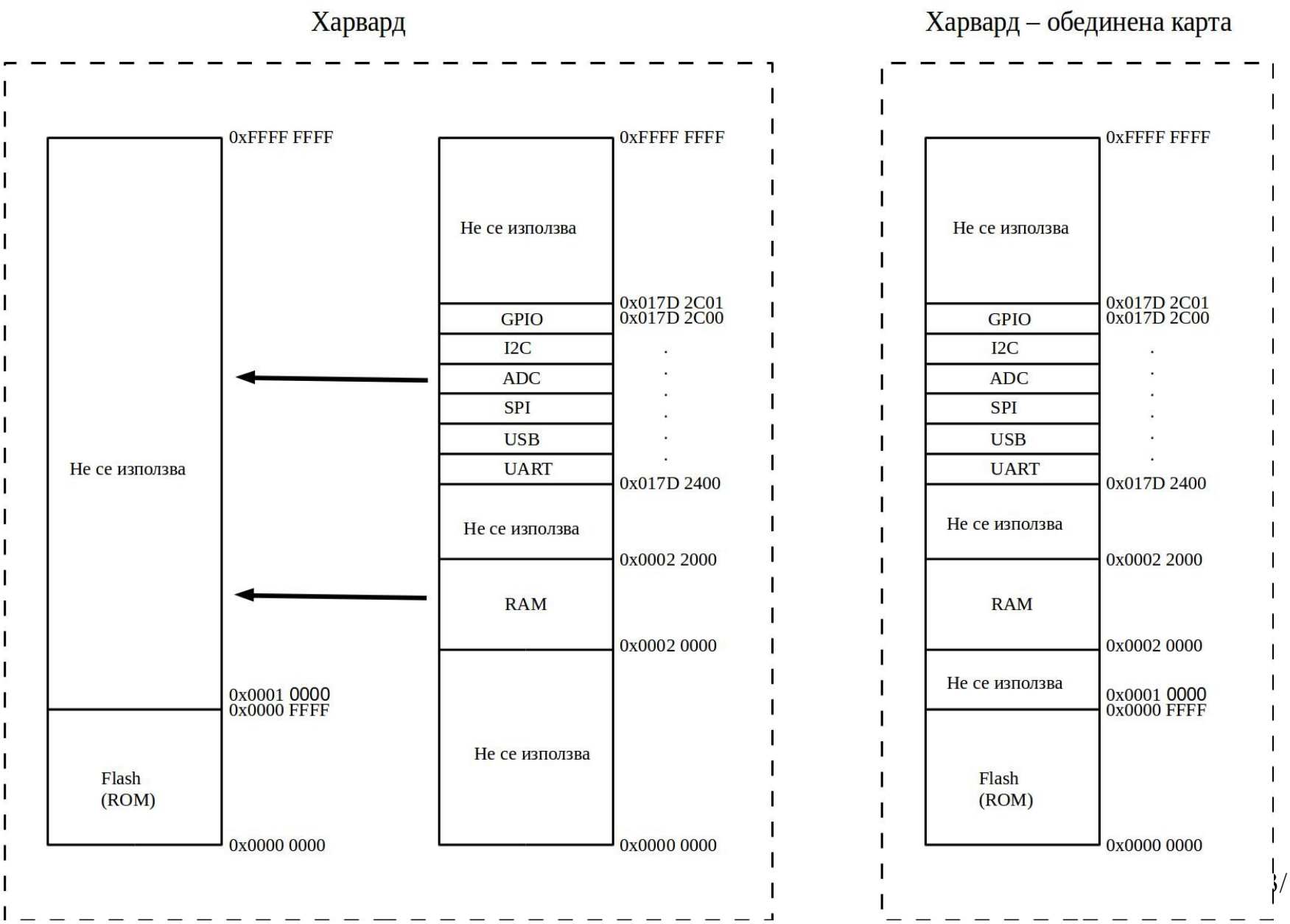
Карта на паметта

Обединена карта на паметта (от англ. ез. unified memory map) е графичното представяне на адресните полета на Харвард микропроцесор като едно единствено адресно поле. Това е възможно само когато периферииите и паметите са разположени на незастъпващи се адреси.

!!! ВНИМАНИЕ!!! Такъв вид представяне може да заблуди проектанта, че даден μPU е фон нойманов, но всъщност той да е харвардски.

На следващият слайд е показана обединена карта на паметта.

Карта на паметта



Карта на паметта

Важна подробност – адресното поле на една магистрала може да бъде разделено на няколко подрегиона чрез превключватели (bus matrix). Това е важно за Харвард микропроцесорите.

Пример –

* харвардски μPU с два набора магистрали се разклонява към три набора магистрали чрез превключвател.
* достъп до Flash паметта – 2 (Харвард)
* достъп до SRAM - 1 (фон Нойман).

* Инструкции + данни → SRAM = бавно
* Инструкции → Flash / данни → SRAM = бързо

Карта на паметта

Програмната и данновата памет са съвкупност от паралелни регистри, чиито адреси нарастват линейно.

Съществуват два вида разполагане на данните в паметта:

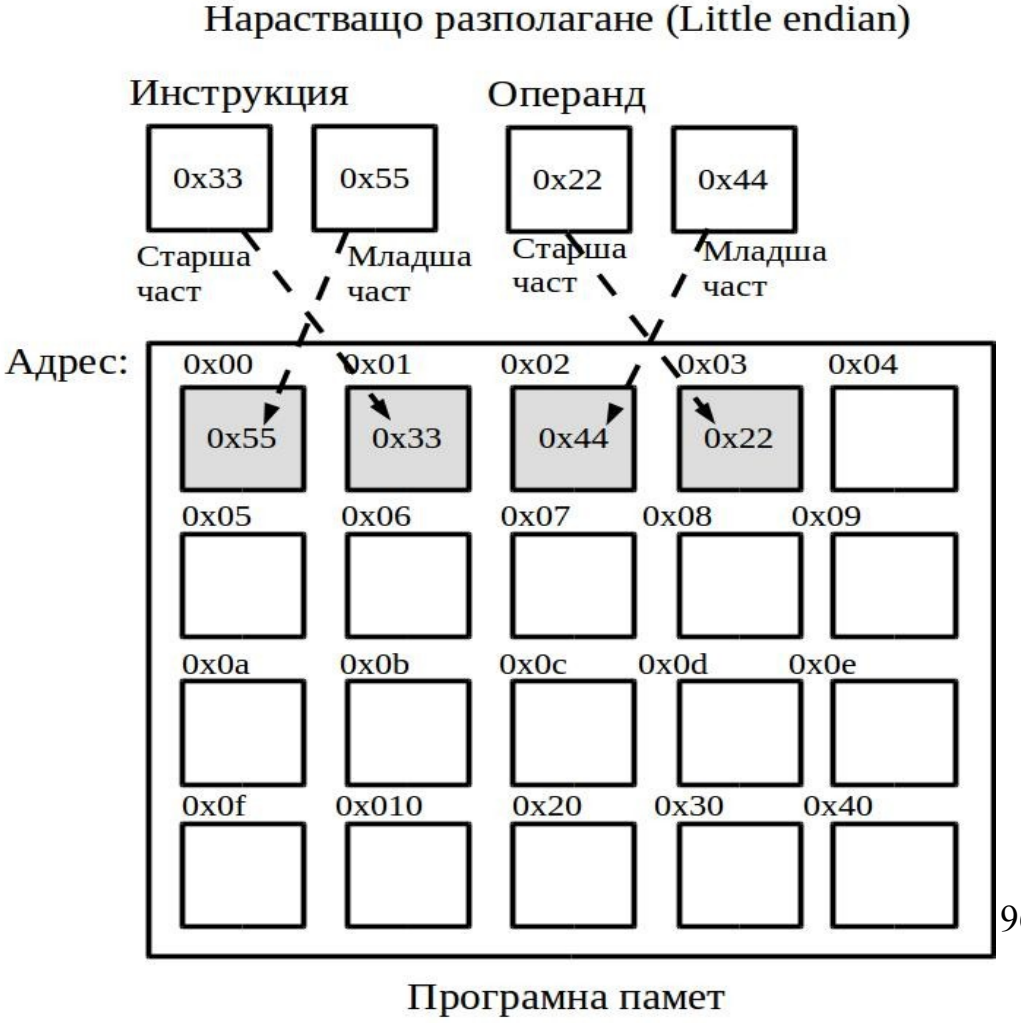
* нарастващо разполагане (**little endian**) – младшият байт се разполага на по-нисък адрес в рамките на една системна дума;
* намаляващо разполагане (**big endian**) - младшият байт се разполага на по-висок адрес в рамките на една системна дума.

Повечето инженери са свикнали да представят числата в big endian формат, където цифрите отляво са старшата част, а цифрите отдясно – младшата част.

Карта на паметта

Пример – микропроцесор с нарастващо разполагане (**little endian**) и 16-битови думи има инструкция 0x3355 с операнд 0x2244. Ако се разгледа дисасемблера на програмата, и адресите на паметта се изобразяват отляво надясно, от ниски към високи, то паметта ще изглежда така:

0x55 0x33 0x44 0x22



Карта на паметта

Пример – микропроцесор с нарастващо разполагане (**little endian**) и 32-битови думи има инструкция 0xf710aabe с операнд 0x12345678. Ако се разгледа дисасемблера, то паметта ще изглежда така:

Адреси

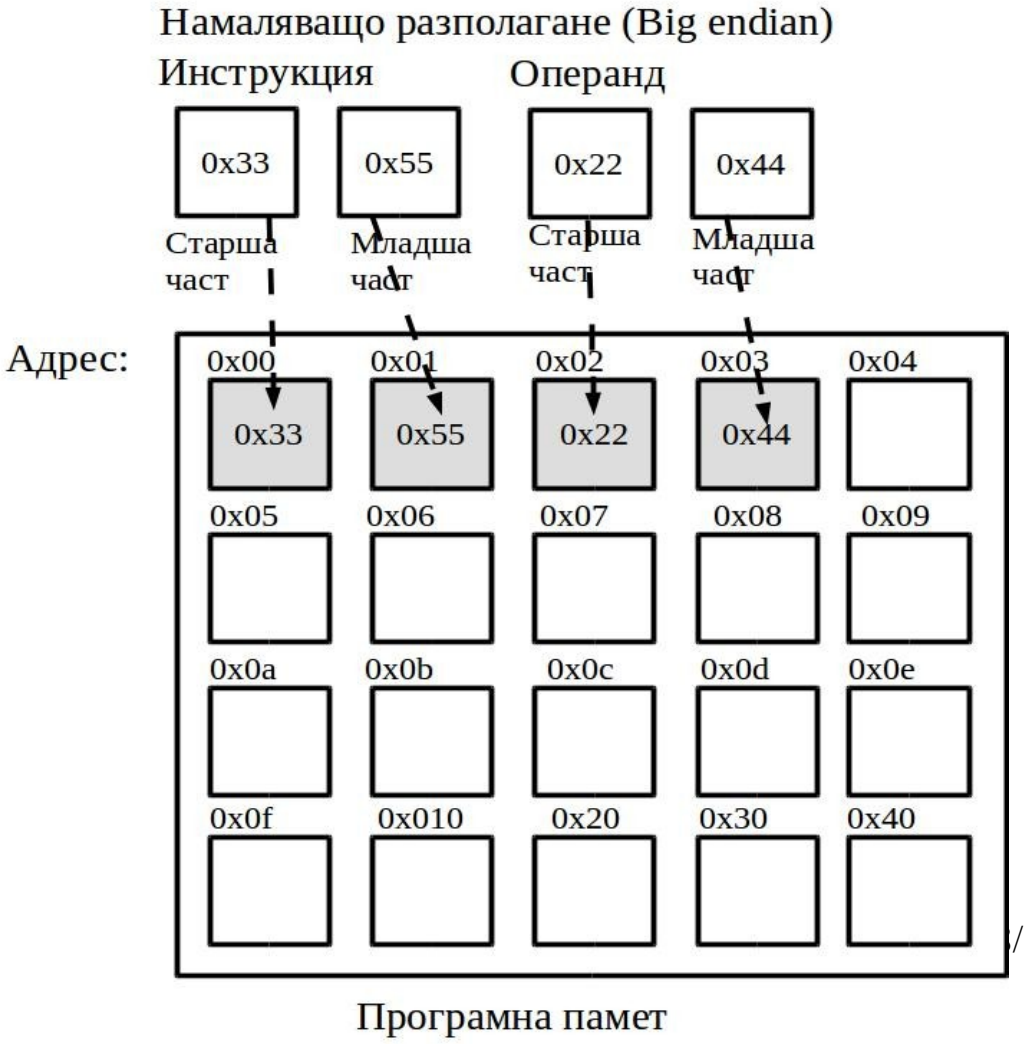
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0xbe	0xaa	0x10	0xf7	0x78	0x56	0x34	0x12

Числа

Карта на паметта

Пример – микропроцесор с намаляващо разполагане (**big endian**) и 16-битови думи има инструкция 0x3355 с операнд 0x2244. Ако се разгледа дисасемблера на програмата, и адресите на паметта се изобразяват отляво надясно, от ниски към високи, то паметта ще изглежда така:

0x33 0x55 0x22 0x44



Карта на паметта

Пример – микропроцесор с намаляващо разполагане (**big endian**) и 32-битови думи има инструкция 0xf710aabe с операнд 0x12345678. Ако се разгледа дисасемблера, то паметта ще изглежда така:

Адреси							
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0xf7	0x10	0xaa	0xbe	0x12	0x34	0x56	0x78
Числа							

Литература

[1] Г. Михов, “Цифрова схемотехника”, ТУ-София, 1999.
[2] P. Wilson, “Design Recipes for FPGAs”, MPG Books Ltd, 2007.
[3] K. Hintz, D. Tabak, “Microcontrollers: Architecture, Implementation, and Programming”, McGraw-Hill Inc, 1992.
[4] Г. Михов, “Настройка и диагностика на микропроцесорни системи”, ТУ-София, 2005.
[5] M. Lipp, *et al*, “Meltdown: Reading Kernel Memory from User Space”, preprint, 2018.
[6] “Cache, Write Buffer and Coprocessors”, ARM7500 Data sheet, 1995.
[7] D. Kumar, R. Behera, K. Pandey, “Concept of a Supervector Processor: A Vector Approach to Superscalar Processor, Design and Performance Analysis”, International Journal of Engineering Research Volume No.2, Issue No.3, pp : 224-227, ISSN : 2319-6890, 2013.
[8] M. Flynn, “Computer architecture: pipelined and parallel processor design”, p. 9-12. ISBN-0867202041, 1995.