Конвейерно изпълнение на инструкцията

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

Изпълнение на инструкцията

Вътрешната структура на един μPU може да се раздели на две части [1]:

*__фронтенд__ (front end) – състои се от контролен модул (включващ програмен брояч PC, регистър на инструкцията IR, регистър на състоянието STAT, и др.) и входно-изходен I/O модул (отговорен за работа с паметта);

*__бекенд__ (back end) – състои се от АЛУ и регистри с общо предназначение GPR.

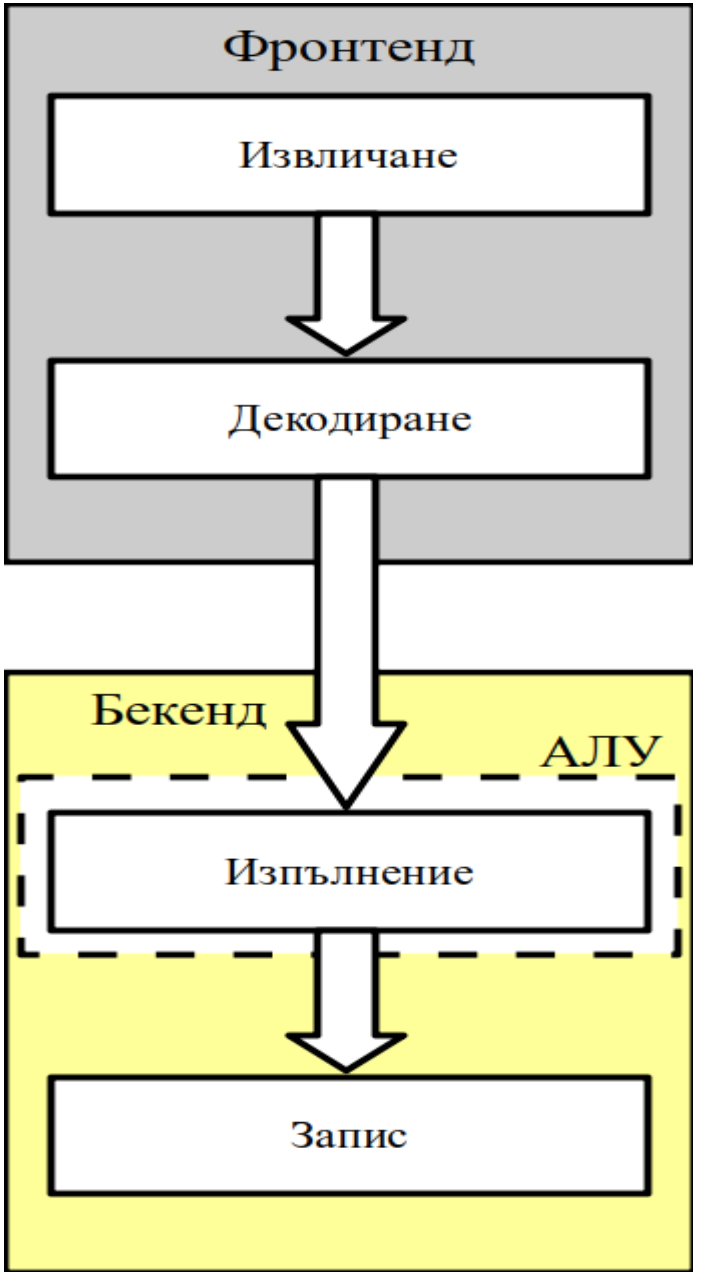
Изпълнение на инструкцията

Изпълнението на една инструкция минава през /поне/ 4 фази:

*извличане (**I**nstruction **F**etch, IF)
*декодиране (**I**nstruction **D**ecode, ID)
*изпълнение (**Ex**ecute, EX)
*запис на резултат (**Wr**ite, WR)

Изпълнение на инструкцията

Извличането и декодирането се извършват от фронтенда.



Изпълнението и записът на резултата се извършват от бекенда.

Изпълнение на инструкцията

Допълнително записът на резултата може да се раздели на две фази:

***вътрешен запис** (**W**rite **B**ack, WB) – след завършване изпълнението на инструкцията, резултатът се записва обратно в регистровия файл на ядрото (или още - работните регистри, GPR);

***външен запис** (**Mem**ory Access, MEM) - след завършване изпълнението на инструкцията, резултатът се записва във данновата (в повечето случаи – RAM) памет.

Изпълнение на инструкцията

Едно типично изпълнение на инструкцията е показано на следващия слайд.

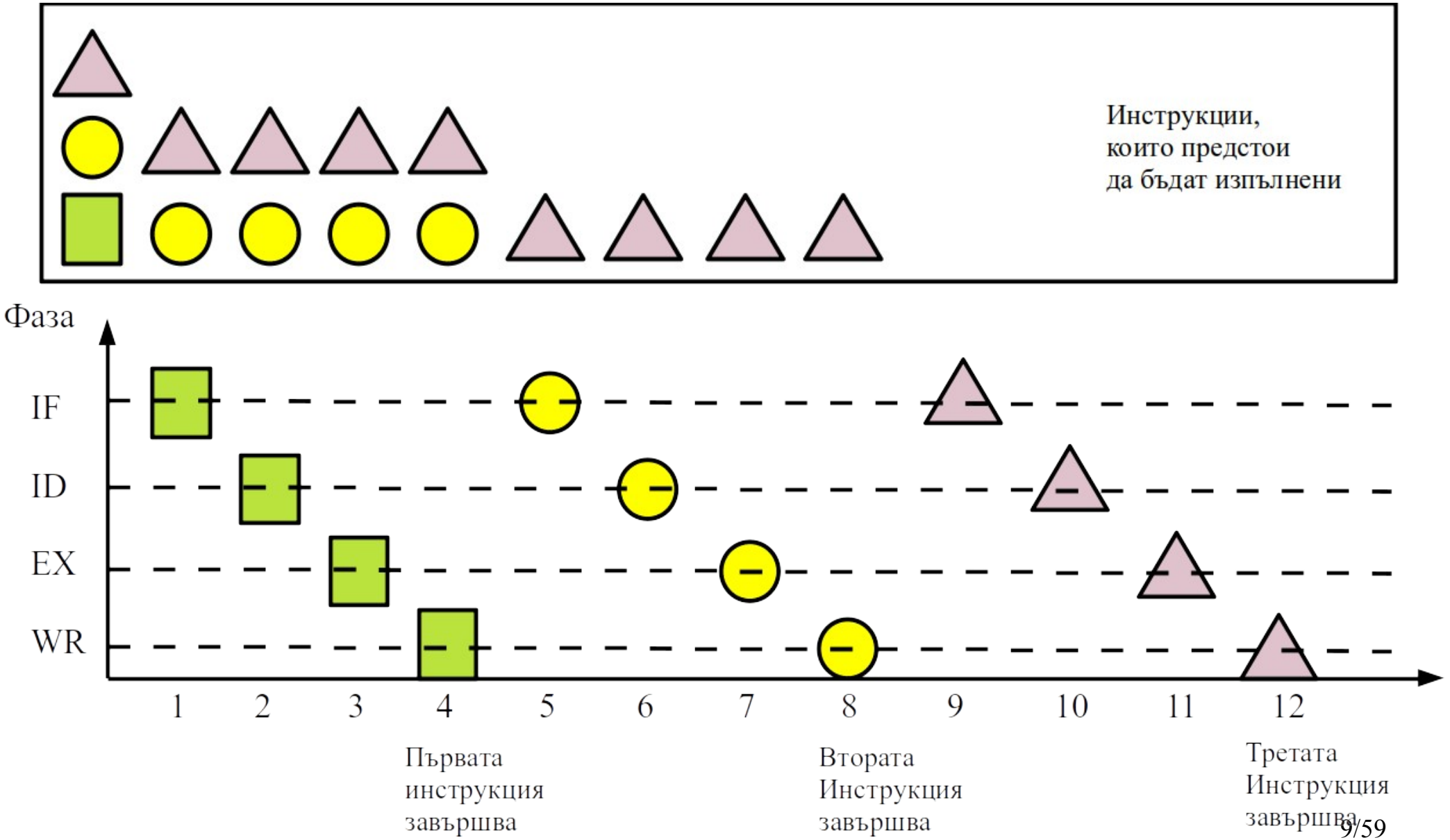
Забележете, че по време на всеки такт само е активна само една от фазите.

Такъв вид изпълнение води до много ниско работно натоварване на отделните модули на μPU, докато една инструкция мине през всичките 4 етапа.

В примерът на следващия слайд инструкцията ще бъде изпълнена за 12 такта.

За увеличение на производителността на един μPU, при запазване на тактовата честота, се реализират т.нар. **конвейерни микропроцесори** (pipelined microprocessors).

Изпълнение на инструкцията



Не-конвейерно изпълнение на инструкция

Конвейери

Конвейерно изпълнение на инструкция (pipelined execution) – отделните микропроцесорни модули, отговорни за IF, ID, EX и WR са автономни и **не** трябва да изчакват всяка инструкция да завърши изпълнението си преди да се захване следващата.

Съвкупността от модули, отговорни за изпълнението на една фаза, формират една **степен на конвейера** (stage).

Чрез увеличаване сложността на хардуера и реализирането на конвейери може да се постигне по-висока производителност без да се увеличава работната честота на ядрото.

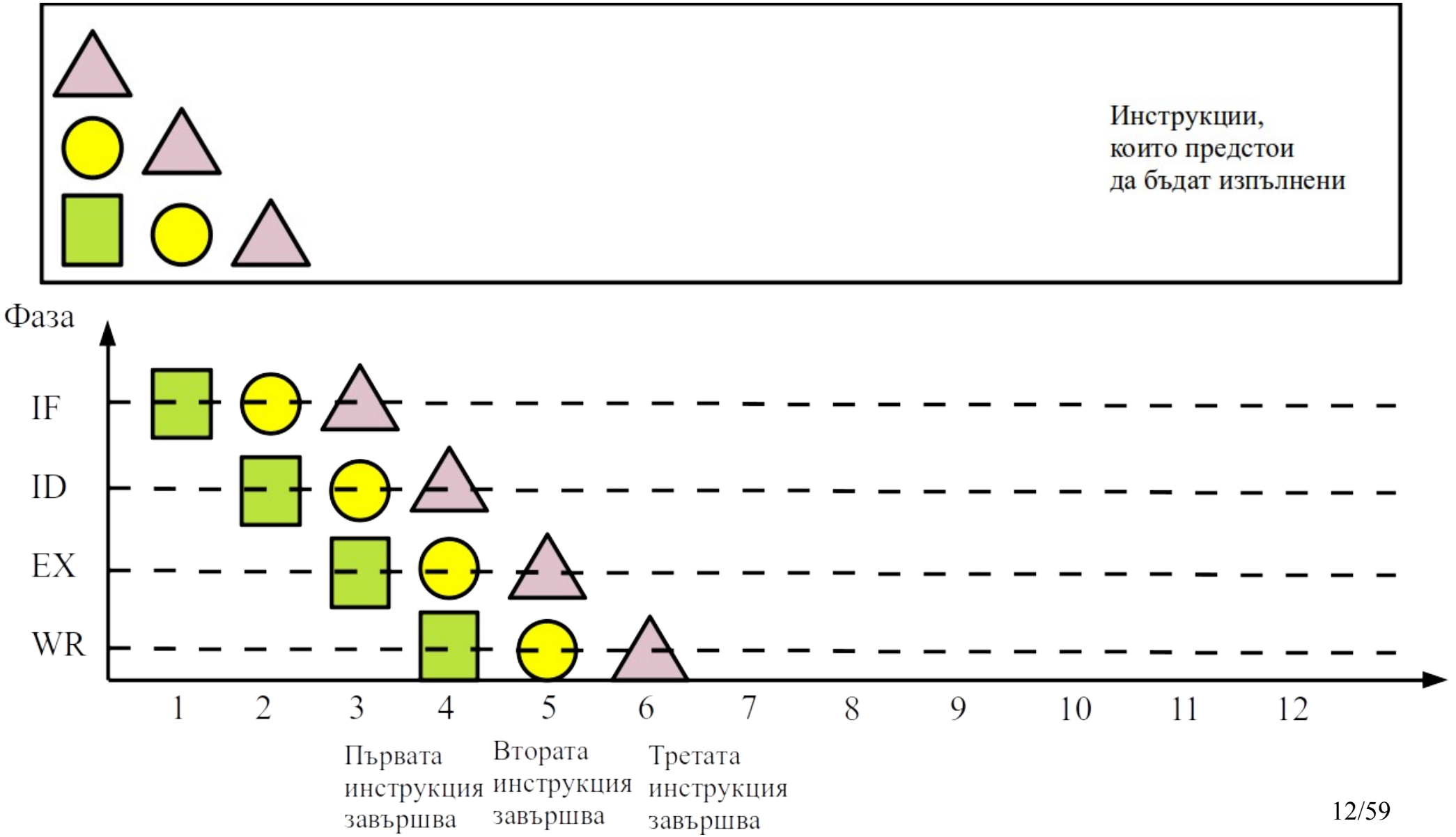
Конвейери

На следващият слайд е показано изпълнението на инструкции на конвейер.

*Процесът започва с извличане (IF1) на първата инструкция.
*След това се преминава във фаза декодиране (ID1). В същия времеви слот втора инструкция започва да се извлича (IF2).
*В третата фаза се изпълнява първата инструкция (EX1). В същия времеви слот втората инструкция се декодира (ID2), а трета бива изличана (IF3).

Трите инструкции се изпълняват за 6 такта или два пъти по-бързо от варианта с не-конвейерния μPU.

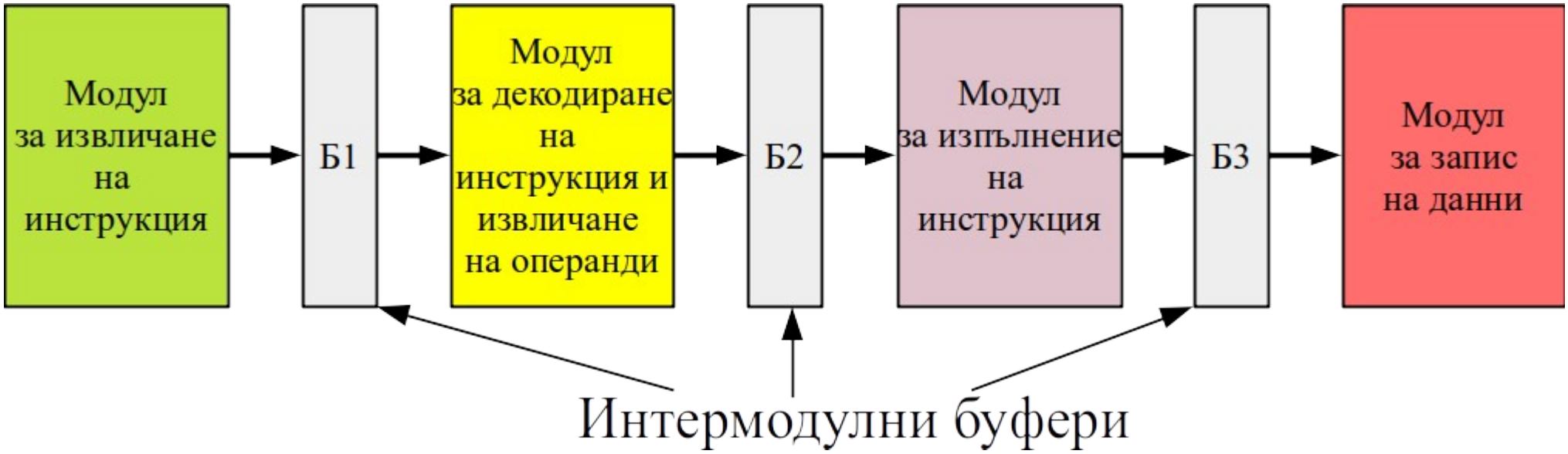
Конвейери



Конвейерно изпълнение на инструкция

Конвейери

За да е възможно изпълнението на конвейер, между отделните модули се добавят буферни регистри, които не са достъпни за програмиста. Това са т.нар. **интермодулни буфери** (interstage buffers).



Конвейери

В литературата може да се срещне твърдението, че **една инструкция се изпълнява за един такт**.

Това всъщност е **еквивалентната производителност** на процесора, когато целият му конвейер е пълен – различни инструкции минават през фазата EX на всеки един такт. Погледнете миналия слайд в интервала такт #3 ÷ такт #5 – на всеки такт се изпълнява инструкция.

Максималната еквивалентна производителност, която μPU може да постигне е 1 инструкция/1 такт, но има събития, които понижават този параметър:
*блокиране на конвейера (pipeline stall)
*зануляване на конвейера (pipeline flush)

Блокиране на конвейерите

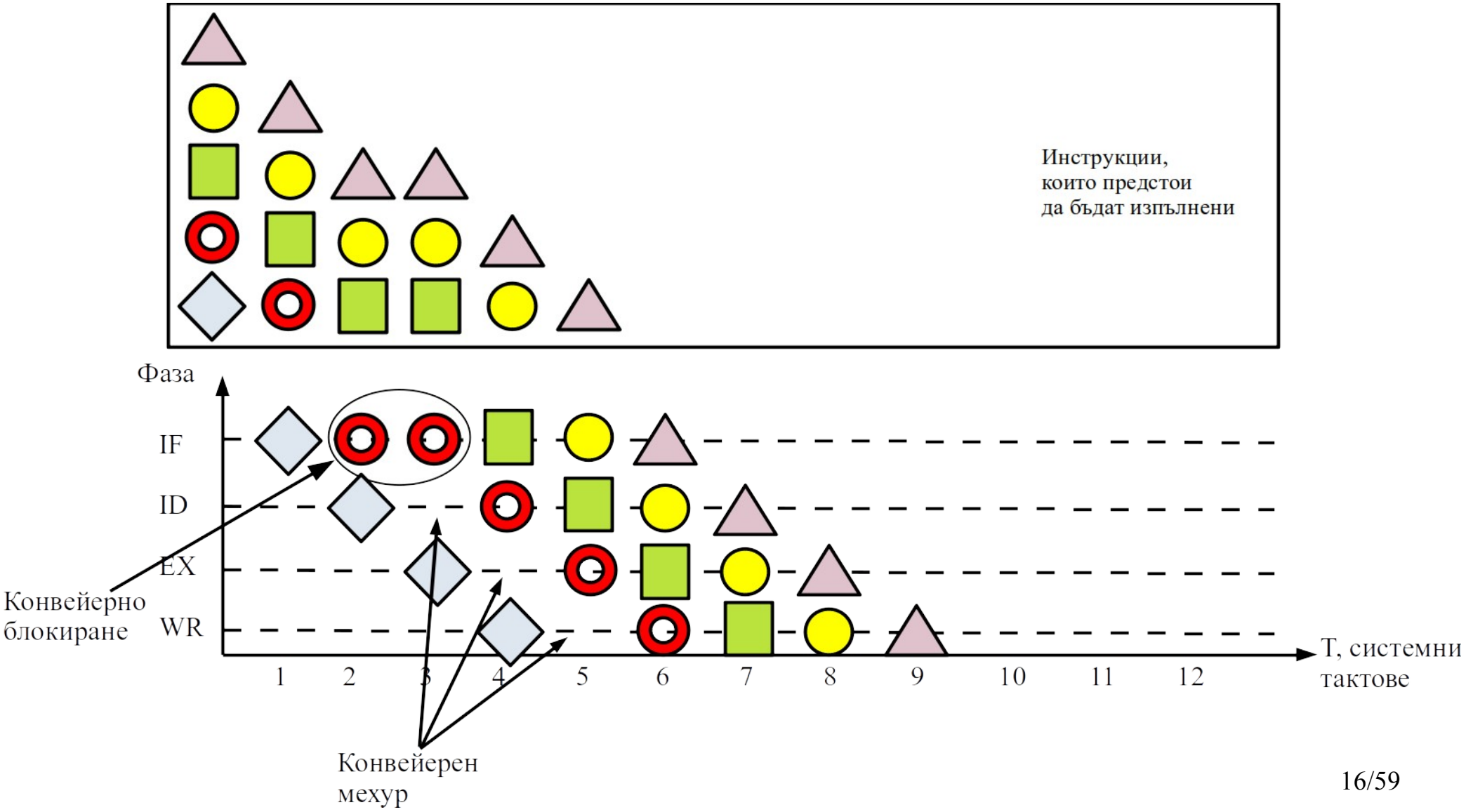
Някои от инструкциите отнемат повече от един такт при преминаването си в дадена фаза. Това води до т.нар. **блокиране на конвейера** (или още "задавяне" на конвейера, pipeline stall).

*Когато една инструкция се забави, забавят се и всички инструкции след нея.
*Когато една инструкция се забави, инструкциите преди нея продължават в следващите фази, докато не завършат изпълнението си.

Това води до т.нар. **мехури в конвейера** (pipeline bubble), които представляват неизползвани модули на μPU.

Следващият слайд илюстрира вариант на блокиране, когато една от инструкциите заеме два такта във фазата на извличане.

Блокиране на конвейерите



Блокиране на конвейерите

На по-следващите два слайда са показани графики от [1]. Те изобразяват производителността на μPU при 2-тактово и 10-тактово блокиране на конвейера.

Производителността на един μPU може да се измери с параметрите:
*тактове-за-инструкция (**C**ycles **P**er **I**nstruction, CPI)
*инструкции-за-такт (**I**nstructions **P**er **C**ycle, IPC)

Тези параметри показват средния брой микропроцесорни тактове, необходими за изпълнението на една инструкция от дадена програма:

Блокиране на конвейерите

$$CPI = \frac{\sum_{i=1}^{m} n_i . T_i}{N}$$

където CPI – параметър “тактове за инструкция”, m – брой видове инструкции, n_i – брой инструкции от вида i в програмата, T_i – брой тактове необходими за изпълнението на инструкция от вида i, N – брой инструкции в цялата програма, без значение от вида им.

Блокиране на конвейерите

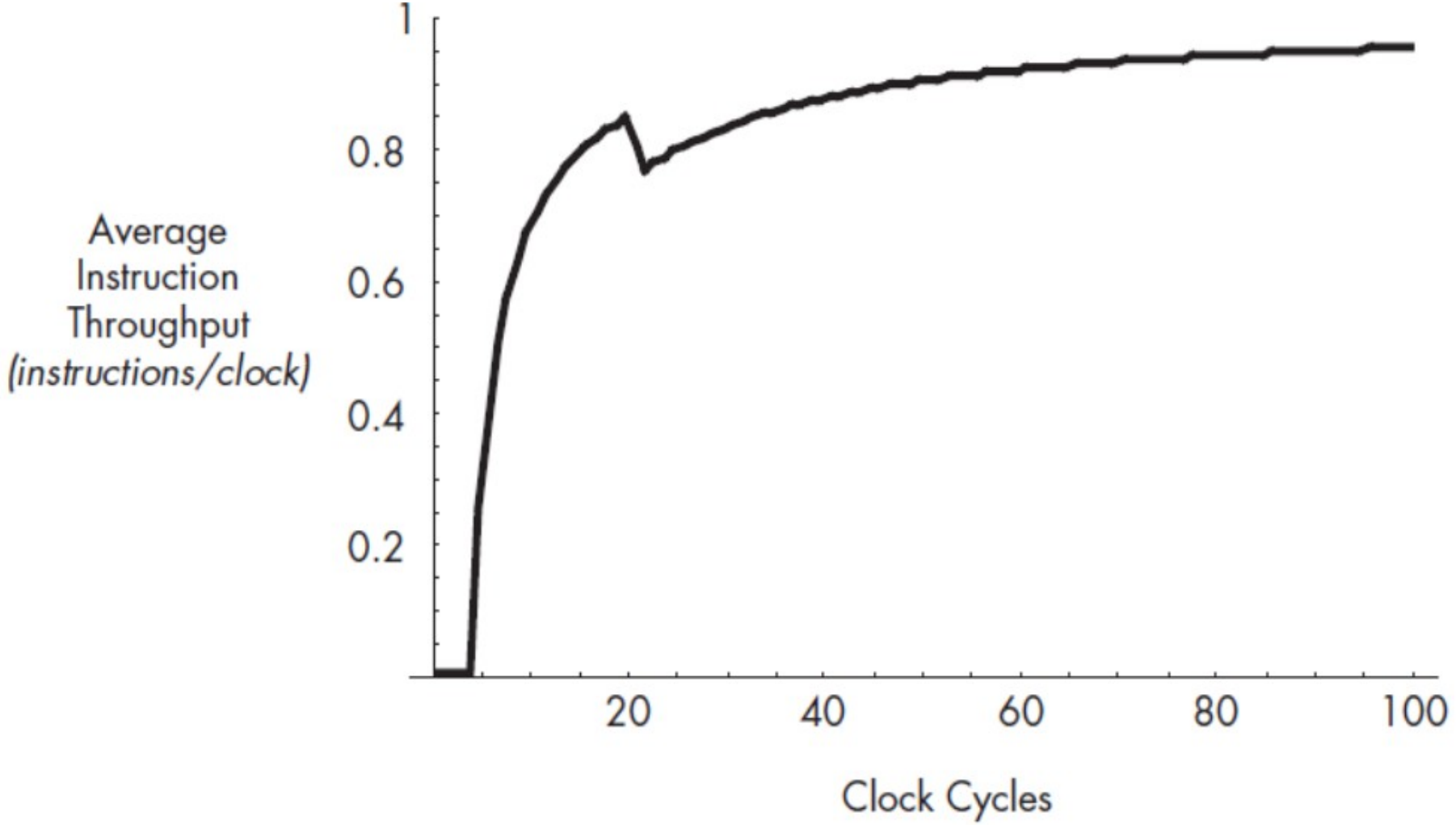


Figure 3-11: Average instruction throughput of a four-stage pipeline with a two-cycle stall

Блокиране на конвейерите

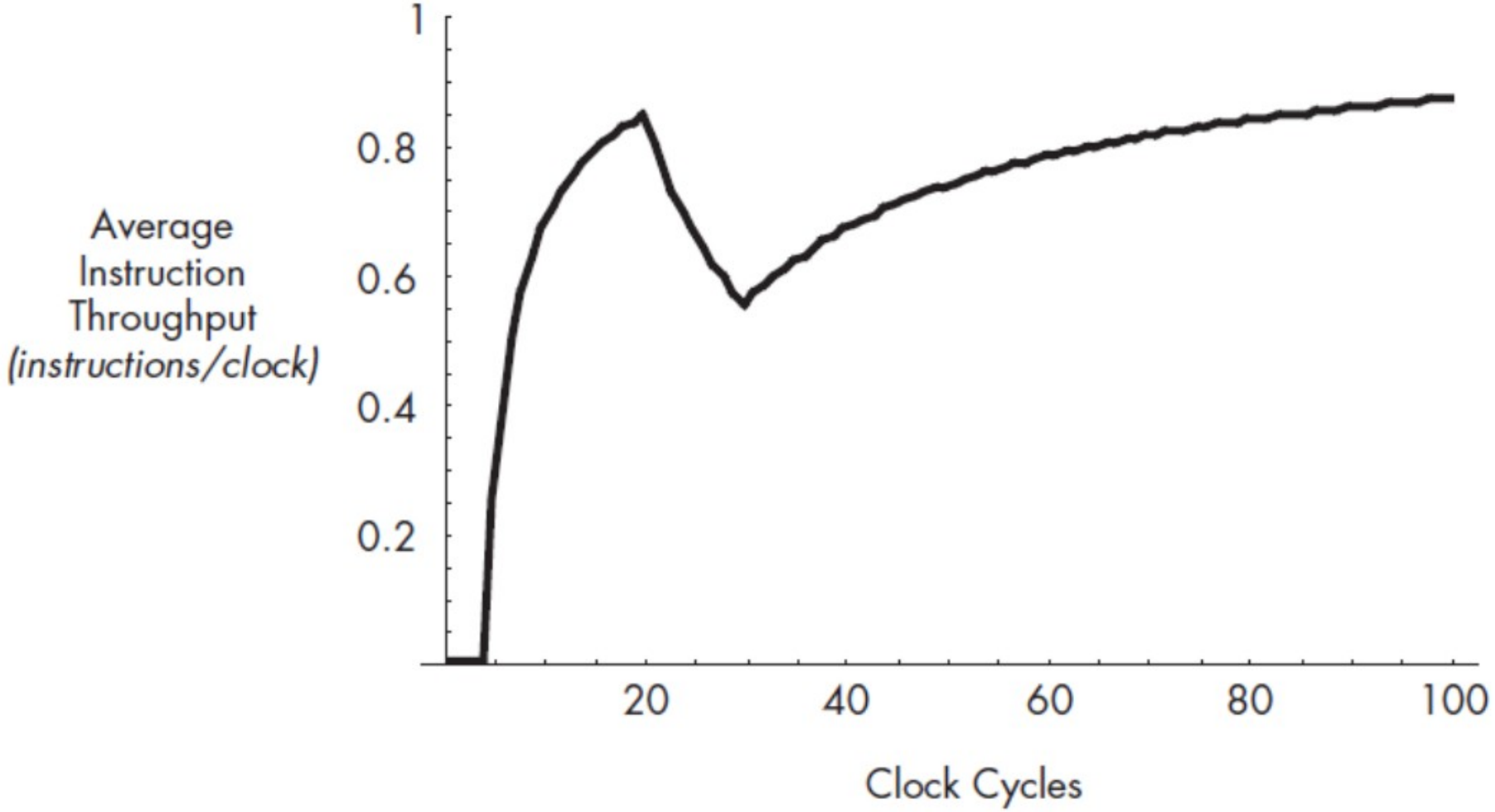


Figure 3-12: Average instruction throughput of a four-stage pipeline with a 10-cycle stall

Блокиране на конвейерите

Събития, които предизвикват блокиране на конвейера се наричат опасности (hazards) [5].

Даннова опасност (data hazard) – събитие, при което операндите-източници и/или операндите-приемници на една инструкция не са достъпни във фазата, в която трябва да се използват.

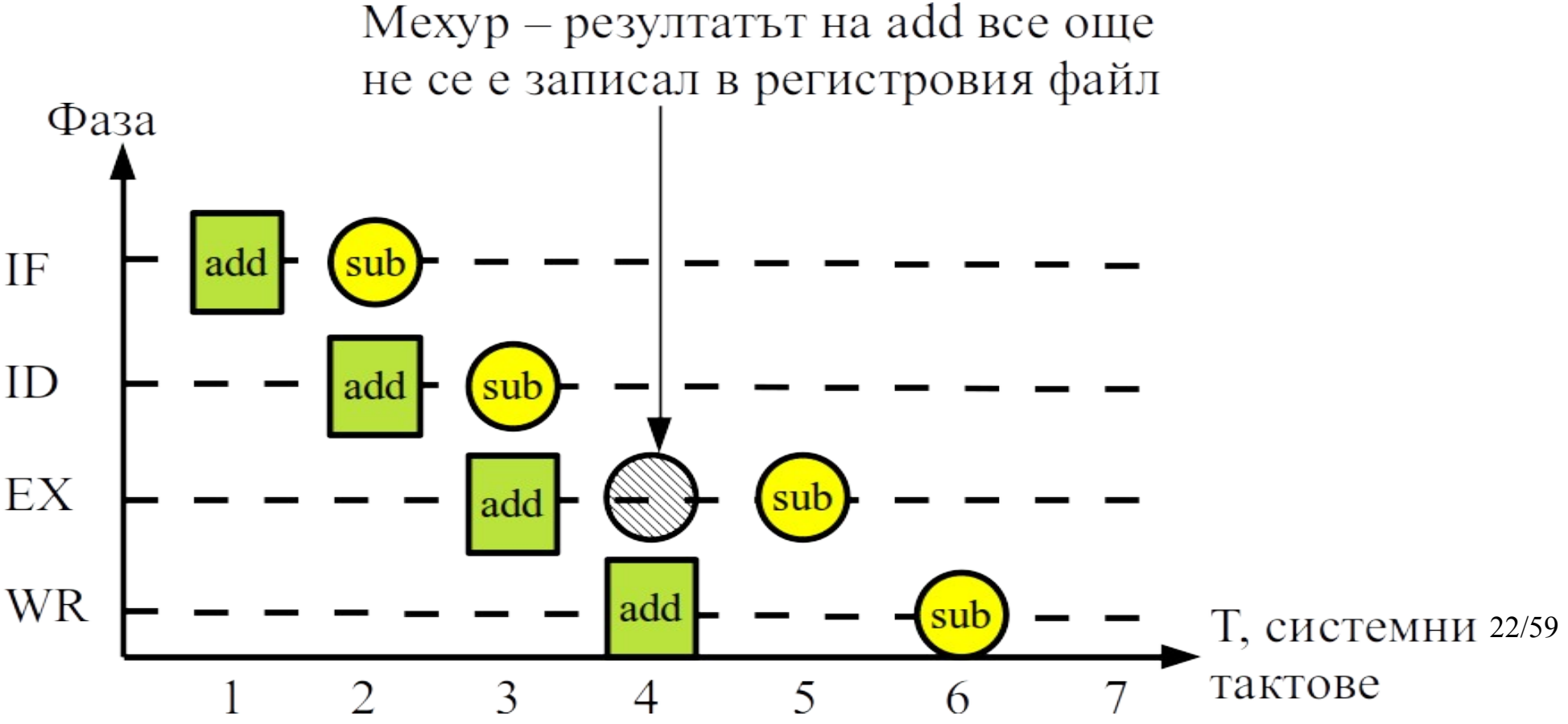
Контролна опасност (control hazard/instruction hazard) – събитие, при което има забавяне в извличането на една инструкция. Това може да се получи, например, от липсата на инструкция в кеш паметта и тогава трябва да бъде извлечена от основната памет.

Структурна опасност (structural hazard) – събитие, при което две или повече инструкции се опитват да достъпят един и същ хардуерен ресурс (най-често памет – затова: отделни кешове и многопортови памети).

Блокиране на конвейерите

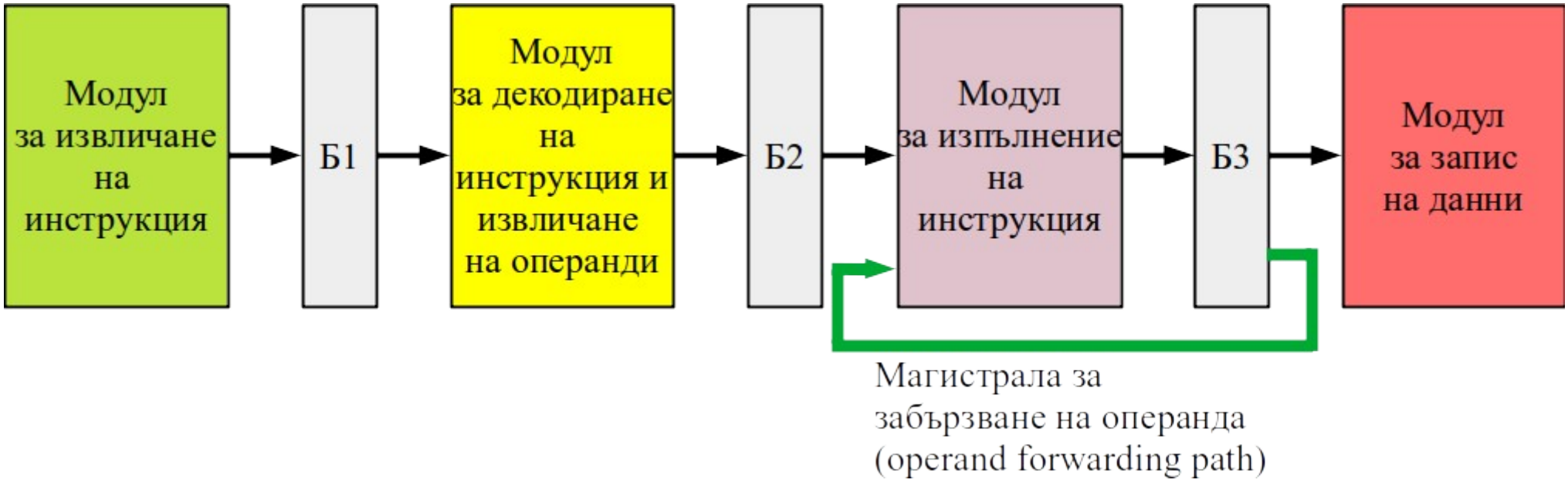
Пример – даннова опасност на две инструкции. Резултатът на add ще бъде използваем чак, когато инстр. мине през фаза WR.

add r1, r2, r3 //събери r2 и r3, запиши резултатът в r1
sub r4, r5, r1 //извади r5 и r1, запиши резултатът в r4



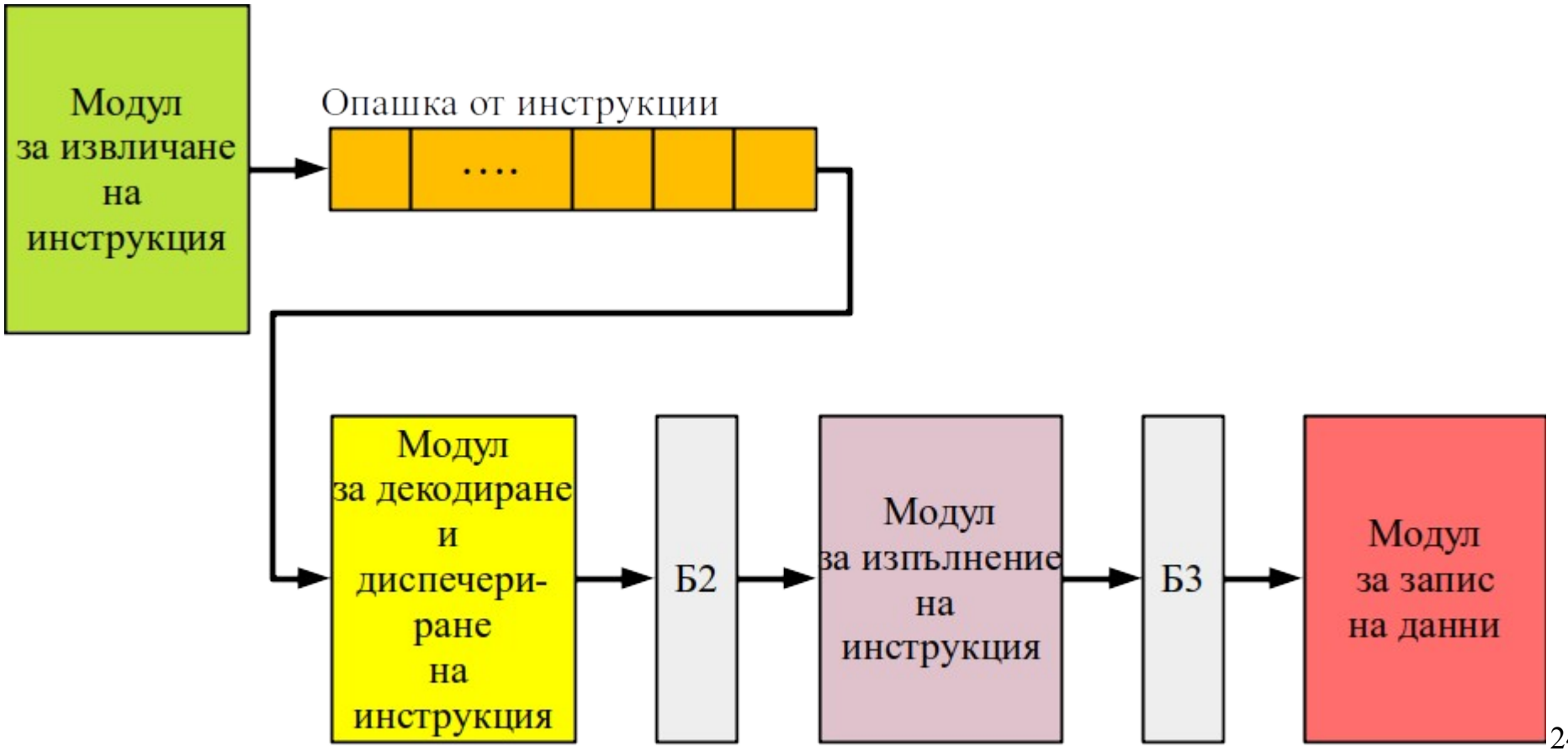
Блокиране на конвейерите

За избягване на данновата опасност се използва една допълнителна магистрала, наречена **магистрала за забързване на операнда**, която е свързана към изходния интермодулен буфер на EX модула и с един мултиплексор се връща на входа на същия този модул. Така инструкциите в EX фазата не трябва да чакат резултатът да се записва обратно в работните регистри на ядрото.



Блокиране на конвейерите

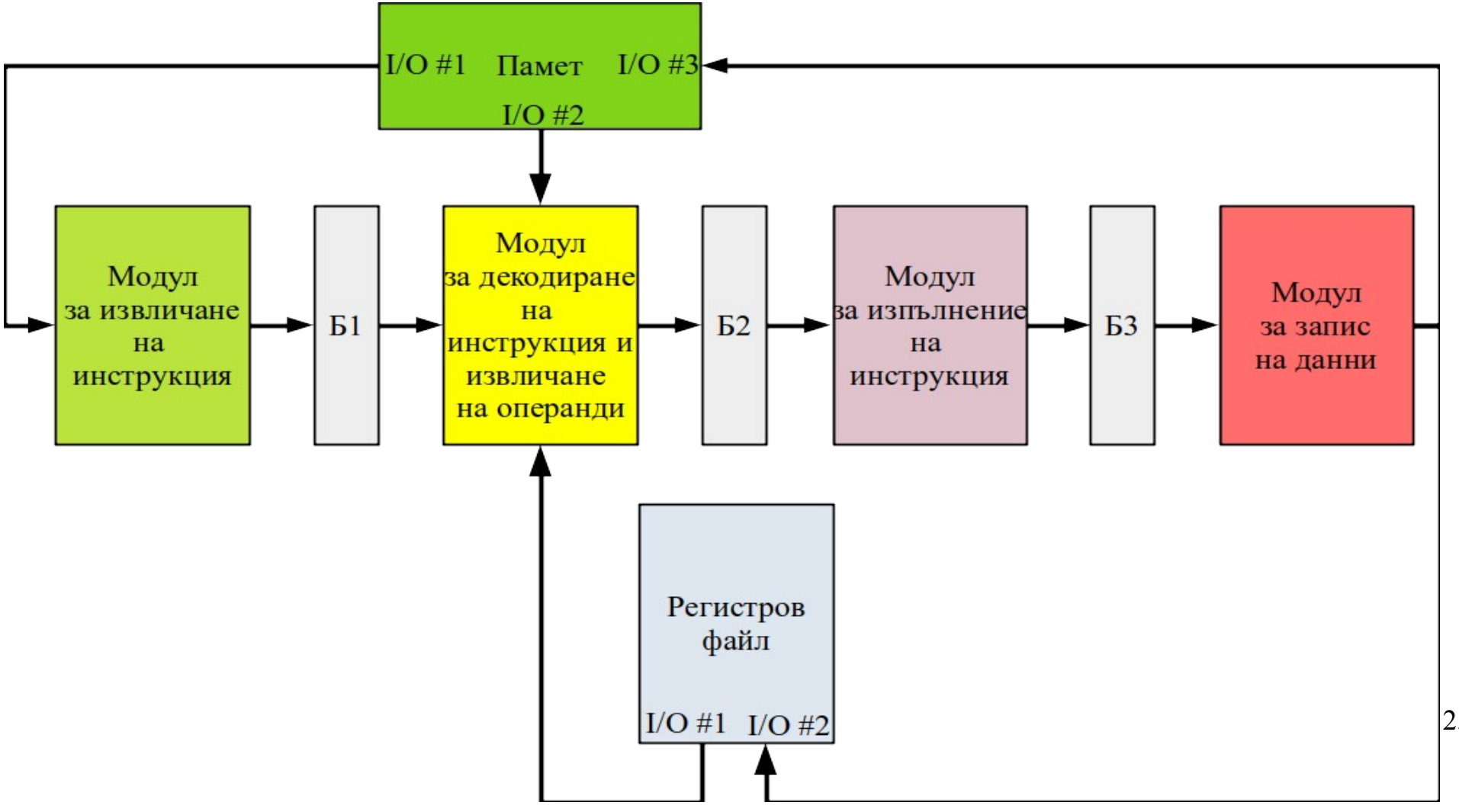
За избягване на контролната опасност се използват опашки от инструкции (буфериране във FIFO буфер). Ако извличането на една инструкция се забави, диспечерът ще продължи да изпълнява инструкции от опашката.



Избягване на контролни опасности чрез опашка

Блокиране на конвейерите

За избягване на структурна опасност се използват отделни модули памет (напр. кеш за данни, кеш за инструкции), или многопортови памети, или многопортови регистрови файлове.



Блокиране на конвейерите

Дори всички инструкции да отнемат по един такт във всяка фаза, понижаване на производителността може да се предизвика и от **преходите в една програма**.

Тогава се казва, че в програмата има **даннова зависимост** (data dependency). В този случай не може да се предскаже в кой клон от програмата ще продължи изпълнението, защото данновата зависимост може да е предизвикана от външни фактори.

На следващия слайд е показан пример с даннова зависимост. На ред 3 (функция function_two( )) се внася даннова зависимост, защото стойността на променливата choice определя дали изпълнението ще мине през “case 1”, или “case 2”, но никога през двете едновременно.

Блокиране на конвейерите

```
1  void function_one(void){
2      int a, b, c, choice;
3   choice = function_two(&a, &b, &c);
4      switch(choice){
5      case 1:
6     a *= a;
7           break;
8      case 2:
9     a = (a * b) + c;
10          break;
11   }
12 }
```

Блокиране на конвейерите

Условното изпълнение води до несигурност в това *коя следваща инструкция* да бъде извлечена.

В примера от миналия слайд, не може да се каже дали да се извлече инструкция от “case 1” или “case 2”.

Ако конвейерът е захванал инструкции от единия клон на програмата, а във фазата EX се окаже, че трябва да се изпълни другия клон, то инструкциите, заредени в степените на конвейера до този момент, ще се окажат невалидни.

Процесът на премахване на инструкции от степените на конвейера, които са се оказали невалидни, се нарича **зануляване на конвейера**.

Блокиране на конвейерите

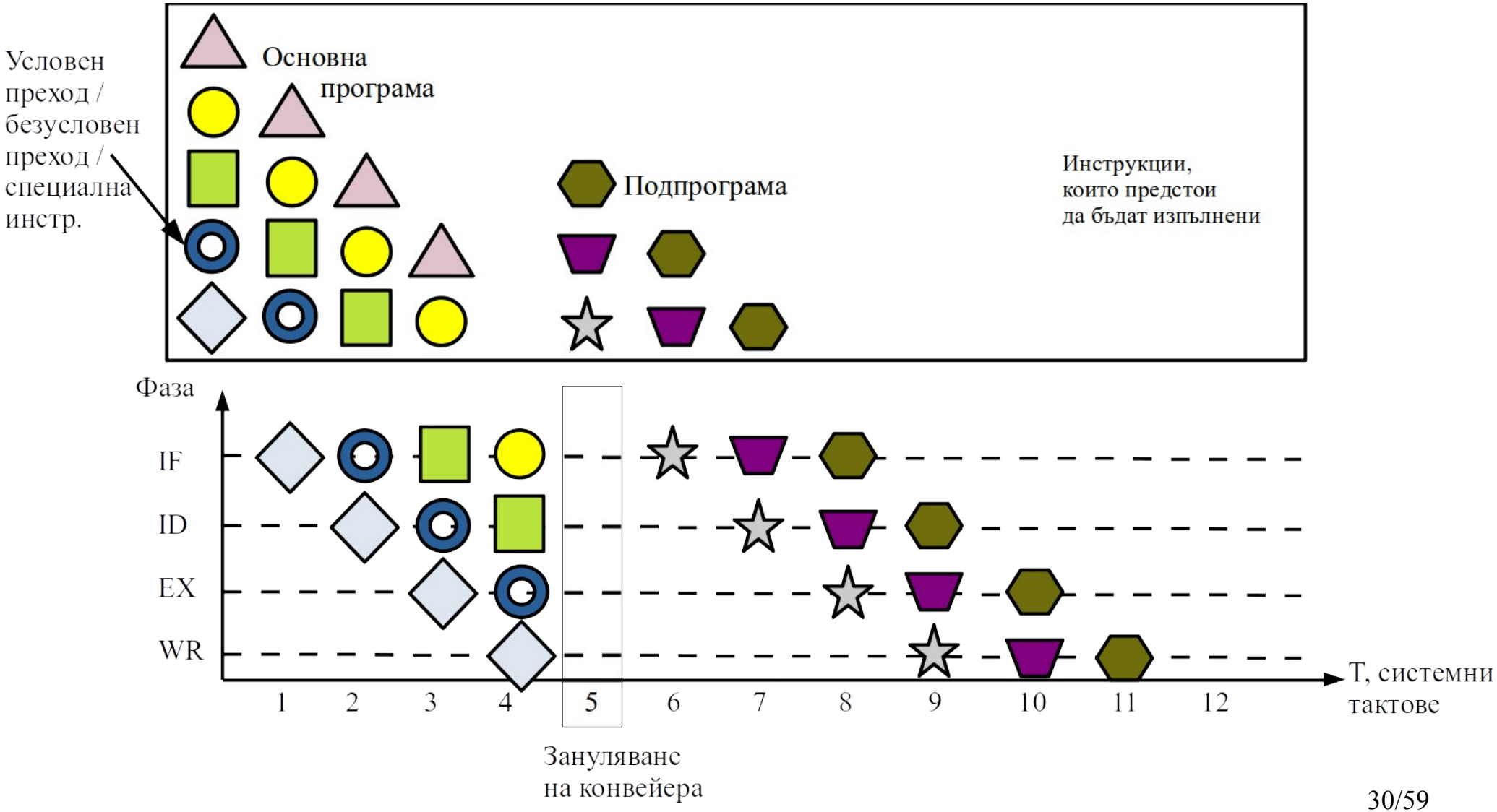
Зануляването на конвейера може да стане по 3 начина:

*чрез използване на **инструкции за разклонения** в програмата (branch instructions) [използва се също термина условен преход]

*чрез използване на **инструкции за преход** (jump instructions) [използва се също термина безусловен преход]

*чрез използване на **специализирани инструкции**

Блокиране на конвейерите



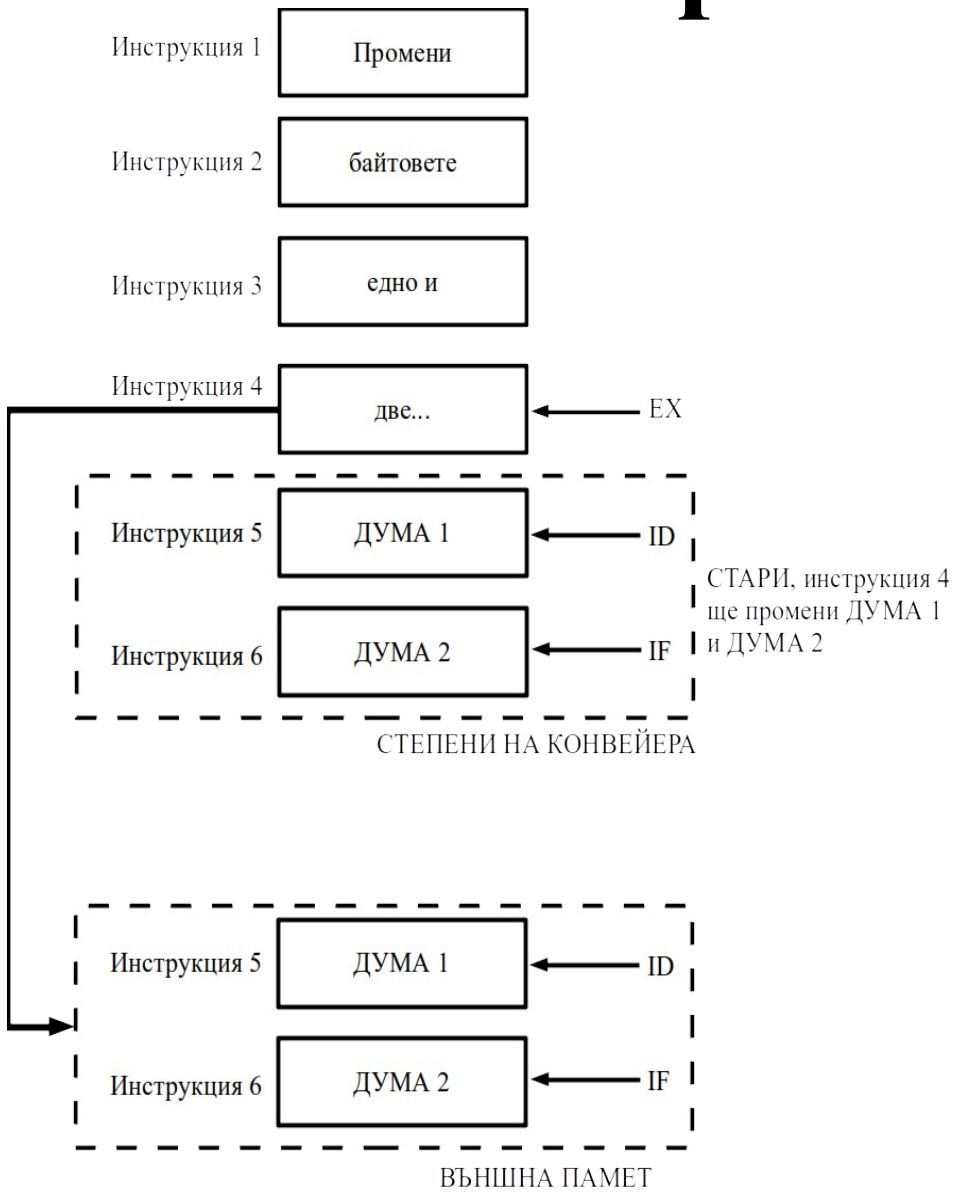
Блокиране на конвейерите

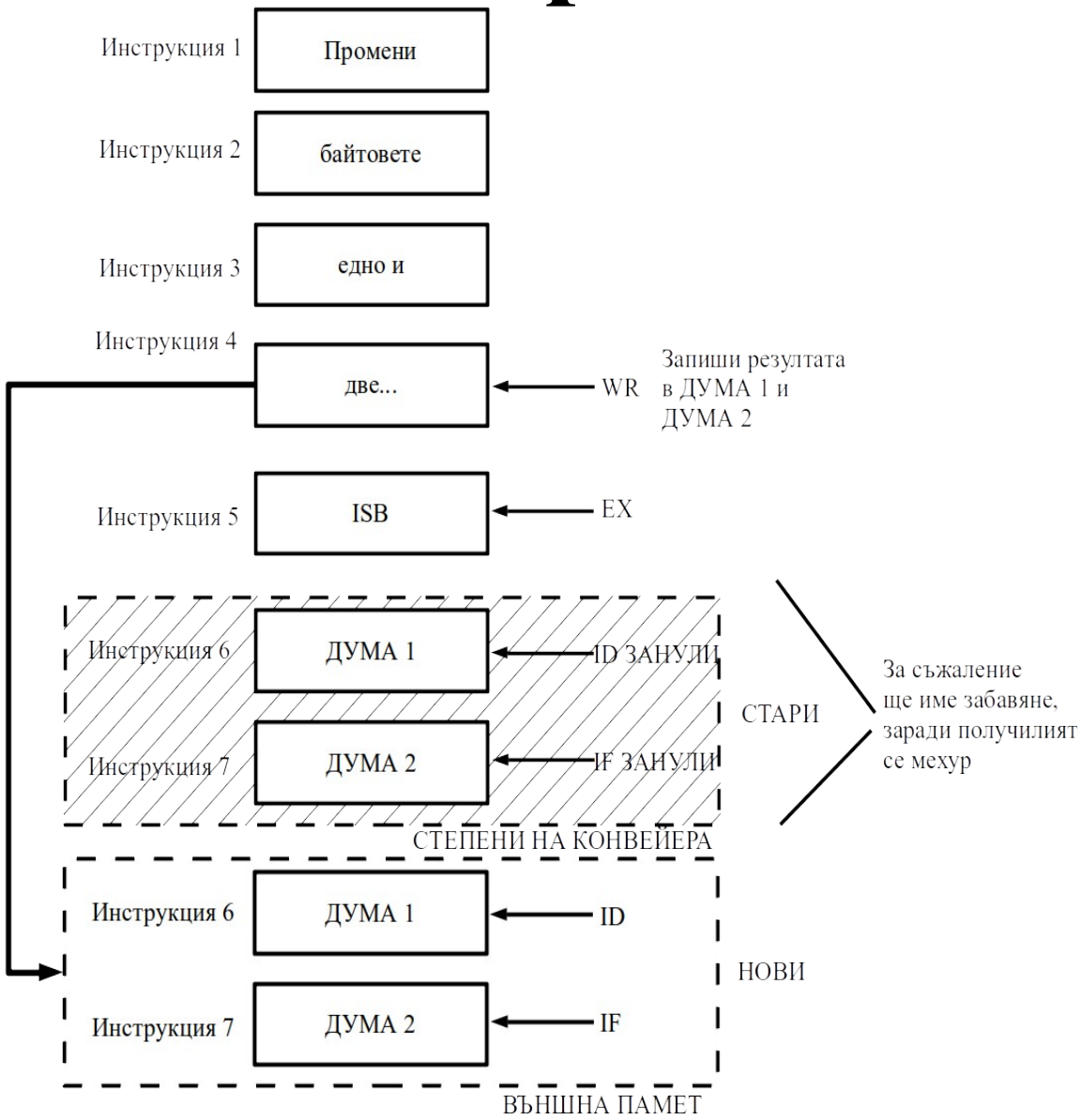
Някои μPU имат специализирани инструкции, които принуждават конвейера да се занули [2].

Пример – инструкцията на ARM Cortex – ISB (**I**nstruction **S**ynchronization **B**arrier). Тя занулява конвейера и подсигурява, че инструкциите след нея (на по-големи адреси) ще бъдат заредени наново. Използва се в самомодифициращ се код, например в приложения с изкуствен интелект.

На следващия слайд е показан пример със самомодифициращ се код.

Блокиране на конвейерите





Конвейери и програмен брояч

Програмният брояч PC е регистър от микропроцесорното ядро, който съдържа адреса на следващата инструкция, която предстои да бъде изпълнена.

След извличане на следващата инструкция, PC се увеличава автоматично (хардуерно), така че да сочи към по-следващата и т.н., докато не се стигне края на програмата.

Увеличаването на PC зависи от размера на инструкцията. При 8-битови инструкции PC се увеличава с +1, при 16-битови с +2, при 32-битови с +4 и т.н.

Конвейери и програмен брояч

При конвейерните µPU програмният брояч PC се увеличава с число, пропорционално на:

***дължината на инструкцията**

***степените на конвейера** от началото до EX модула.

Конвейери и програмен брояч

n - разредност на инструкцията в байтове (8-битови => n=1, 16-битови => n=2, 32-битови => n=4, и т.н.).

Пример – ако степените на конвейера са:
1.IF
2.ID
3.EX
4.WB
програмният брояч, ще се увеличава с 2 x n.

Конвейери и програмен брояч

Пример – ако степените на конвейера са:
1.IF
2.ID
3.REN (Rename & dispatch)
4.Q (Queue)
5.IS (Issue)
6.EX
7.WB (Write Back)
8.MEM (Memory Access)

програмният брояч, ще се увеличава с 5 x n.

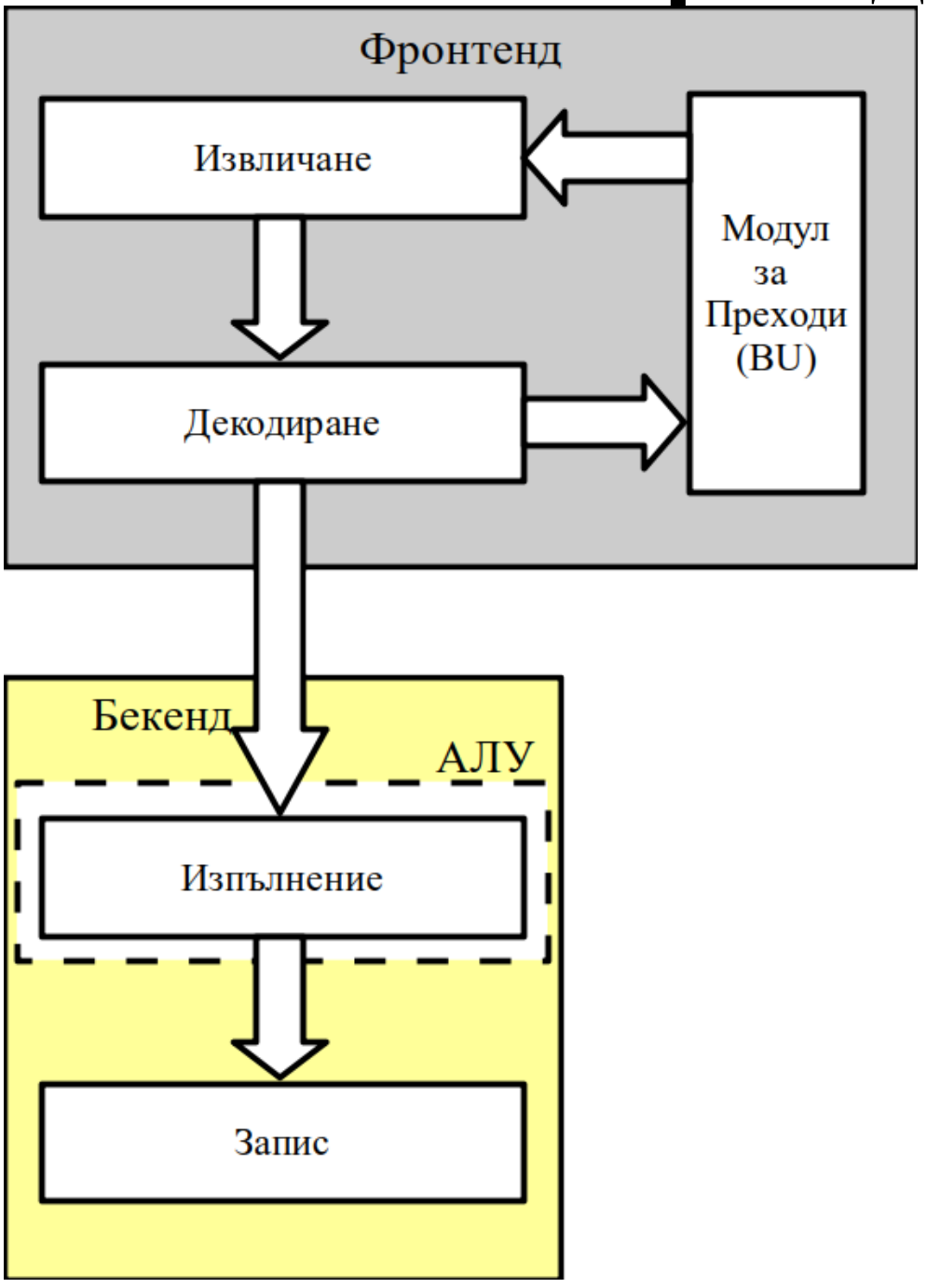
Предсказване на преходите

Мехурите в конвейерното изпълнение трябва да бъдат избягвани, защото това намалява производителността на μPU. Един начин да се постигне това е да се използва предсказване на преходите (branch prediction).

Модул за преходи (**B**ranch **U**nit, BU) – част от фронтенда на μPU, който работи в синхрон с програмния брояч PC, и който управлява модулите за извличане и декодиране на инструкции, като ги насочва да работят с различни части от програмата.

Състои се от два подмодула:
*модул за предсказване на преходи (**B**ranch **P**rediction **U**nit, BPU)
*модул за изпълнение на преходи (**B**ranch **E**xecution **U**nit, BEU)

Предсказване на преходите



Предсказване на преходите

Когато декодерът на инструкцията детектира инструкция за преход, той я изпраща в BU модула за изпълнение.

BU препраща инструкцията в някой от другите функционални модули на ядрото, ако тя е условна. Това е необходимо, за да се тества условието и да се реши дали преходът ще бъде взет или не (taken, not taken branch).

Ако преходът бъде взет, BU трябва да разбере къде е адреса на фрагмента от код, който трябва да се изпълни. Този адрес се нарича **целеви адрес на прехода** (branch target address).

Предсказване на преходите

Модул за предсказване на преходите (**B**ranch **P**rediction **U**nit, BPU) – хардуерен модул на μPU, който се опитва да предскаже резултата на инструкция за условен преход и изчислява адреса на (предвидения) прехода.

Пресказването на преходите може да бъде два вида:
*статично
*динамично

Предсказване на преходите

Статично предсказване на преходите (static branch prediction) – метод, при който BU модула решава, че всички инструкции преди прехода са част от цикъл в програмата и целевия адрес на прехода е в началото на този цикъл.

Грешно предсказване на прехода (branch misprediction) – събитие, при което предсказанието на BPU се оказва грешно (целевия адрес е грешния) и конвейера трябва да се занули, за да се премахнат инструкциите от грешната част на кода.

Предсказване на преходите

Пример – във фрагмента от код по-долу, тялото на for-цикъла ще се изпълни 1000 пъти. Това означава, че инструкцията за тест, имплементираща кода “i < 1000”, в 1000 пъти от случайте ще направи преход към реда “arr_a[i] = myvar * arr_b[i];” и само веднъж към реда “my_func_b(c, d, e);”.

```
my_func_a(a, b);

for(i = 0; i < 1000; i++){ //Тук се решава накъде ще продължи програмата
    arr_a[i] = myvar * arr_b[i];
}

my_func_b(c, d, e);
```

Предсказване на преходите

Ако BPU използва статично предсказване, то на примера от миналия слайд целевия адрес ще бъде верен в 1000 пъти от случаите и само веднъж ще сгреши (когато i = 1000).

Предсказване на преходите

Динамично предсказване на преходите (dynamic branch prediction) – метод, при който BU модула анализира миналото на програмата. Това става чрез статистика, която се води от два модула:

***таблица с история на преходите** (**B**ranch **H**istory **T**able, BHT) – записват се преходите, които са се случили в програмата за определен отрязък от време (в системни тактове). За всеки преход се записва статистическа информация относно вероятността прехода да бъде взет [3].

[1.преход ли е? 2. Каква е вероятността да бъде взет?]

***буфер на целеви адреси** (**B**ranch **T**arget **B**uffer, BTB) – за всеки елемент от BHT има съответстващ елемент в BTB, който представлява целевия адрес на кода от прехода, когато прехода бъде взет. Този адрес е предвиден адрес и не се знае дали е правилния. [1. Ако е взет прехода, къде точно се намира в паметта?]

Спекулативно изпълнение

В микропроцесори, които имат възможност за изпълнение на много инструкции в паралел (суперскаларни архитектури), предсказването на преходите се оказва недостатъчно за повишаване на производителността [4].

Ако предсказването на прехода се окаже грешно, в конвейера се образува мехур и тогава програмата ще се забави, аналогично на микропроцесорна архитектура без предсказване.

За да се елиминира този недостатък, се пристъпва към дублиране на хардуер и осигуряване на повече от един път за изпълнение на инструкции.

Спекулативно изпълнение

Спекулативно изпълнение (speculative execution) – изпълнение на два (или повече) потока от инструкции в паралел за целите на предсказването на прехода. Потоците от инструкции представляват два (или повече) клона от програмата, които се отнасят към един преход.

Когато преходът премине през EX фазата, се разбира кой е правилният и кой е грешният клон. Инструкциите и данните от грешния клон се нулират, а инструкциите и данните от правилния клон се приемат за верни и ядрото продължава изпълнението на програмата без да се получават мехури в конвейера.

Предсказване на прехода трябва да има, защото преходът може да бъде с повече от два клона (т.е. не само if-else, а също и switch).

Спекулативно изпълнение

Валидиране на инструкцията (instruction commit) – процесът на отразяване на резултата от спекулативно изпълнена инструкция в регистрите и модулите на микропроцесорното ядро. Това се случва чак когато преходът е преминал през EX фазата (evaluate).

Преди да бъде валидирана, инструкцията работи с регистри, които не са видими за програмиста.

Спекулативно изпълнение

Пример:

```
if(a == b){
   c = d + e;
}
else{
   c = d – e;
}
```

Нека предположим, че a != b, тогава се получава графиката от следващия слайд.

Спекулативно изпълнение

Линейно и нелинейно изпълнение

Линейно изпълнение на инструкцията (in-order execution) – инструкциите от програмата се изпълняват от μPU в реда, в който компилатора ги е подредил. В случай, че се появи опасност (hazard), μPU изчаква тя да премине и тогава продължава с изпълнението.

Нелинейно изпълнение на инструкцията (**O**ut-**o**f-**O**rder **E**xecution, OoO) – инструкциите от програмата се изпълняват от μPU в ред, различен от избрания от компилатора.

История – OoO е измислен от Robert Tomasulo в IBM през 1967 за FPU на компютъра IBM System/360, модел 91.

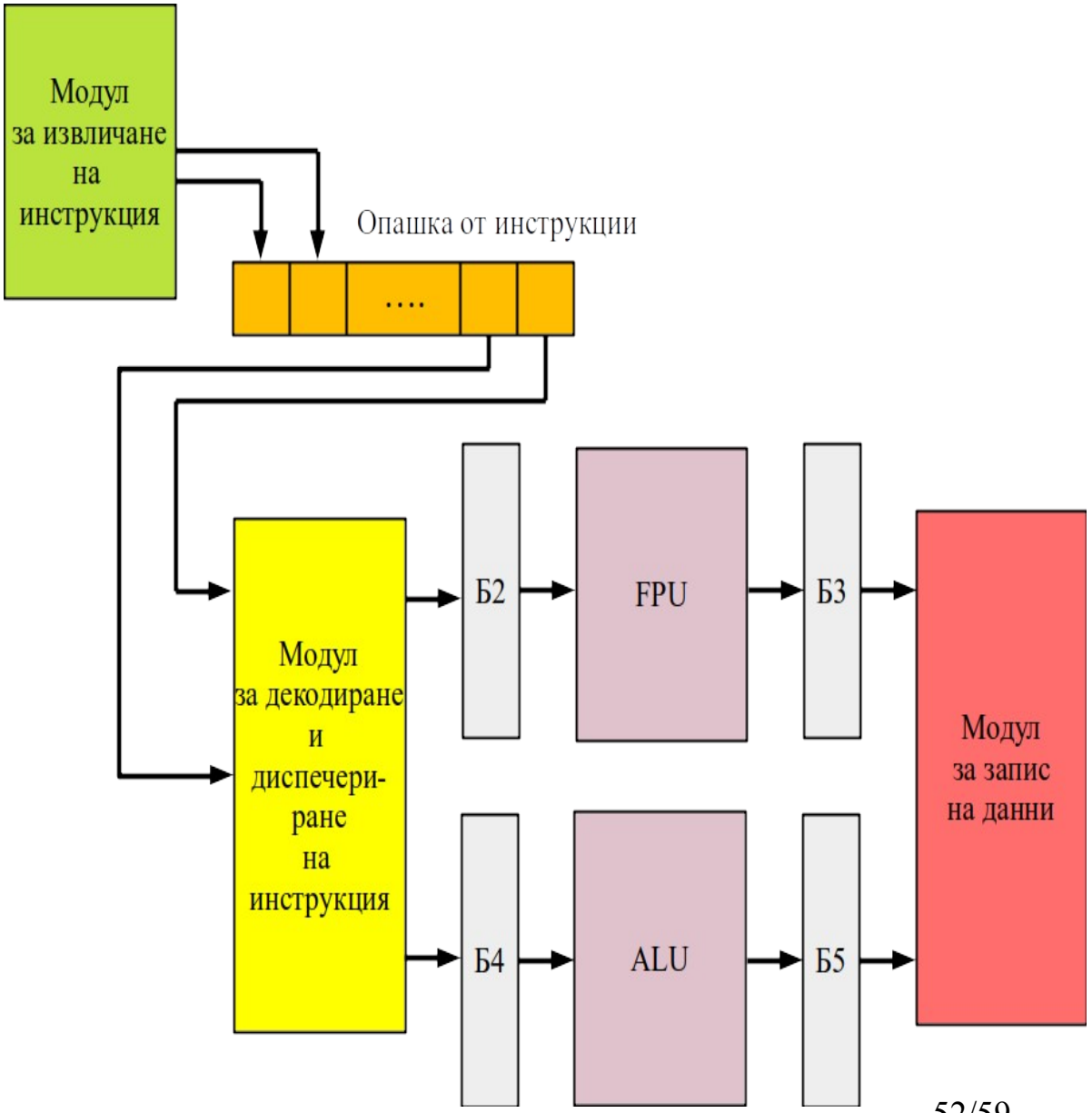
Линейно и нелинейно изпълнение

Пример – в програмата по-долу първите три инструкции се забавят, защото зареждането с инструкцията ldr се забавя. В суперскаларен процесор, диспечерът ще захване четвъртата инструкция (sub), която не зависи от първите две и ще започне да я изпълнява. Налице е OoO.

```
ldr r2, =0x20000040//Зареди адреса на клетка от RAM в r2
ldr r1, [r2, #8]//Зареди данните от адрес 0x20000048 в r1, забавяне заради кеш
add r1, r1, r3 //Събери r1 и r3, запиши резултата обратно в r1
sub r4, r5, r6 //Извади r5 и r6, запиши резултата обратно в r4
```

Линейно и нелинейно изпълнение

Проблемът със суперскаларното изпълнение е, че много често двойките последователни асемблерни инструкции зависят една от друга чрез операндите си => едната част от функционалните модули на суперскаларното ядро остава неизползвана.



Линейно и нелинейно изпълнение

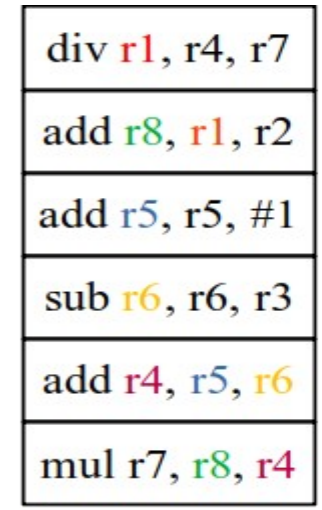
При **µPU с нелинейно изпълнение на инструкцията** (out-of-order) ядрото “разглежда” извадка от няколко последователни инструкции (sliding instruction window) и прави **анализ на зависимостта на операндите** им по време на изпълнение на програмата (**динамично**). Инструкциите, който не зависят една от друга се пускат на функционалните модули за изпълнение.

Резултатите се записват в реда, какъвто е бил в оригиналната програма (in-order).

Всичко става хардуерно и потребителя не вижда тези процеси, освен че програмата му се изпълнява по-бързо.

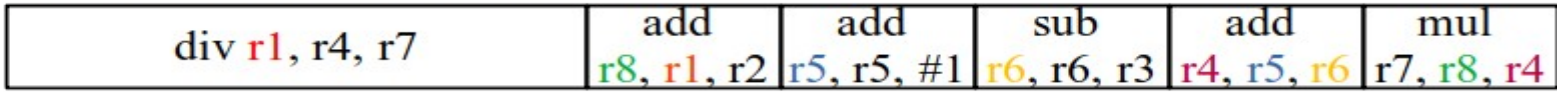
Примерна OoO архитектура е показана на по-следващия слайд.

Линейно и нелинейно изпълнение

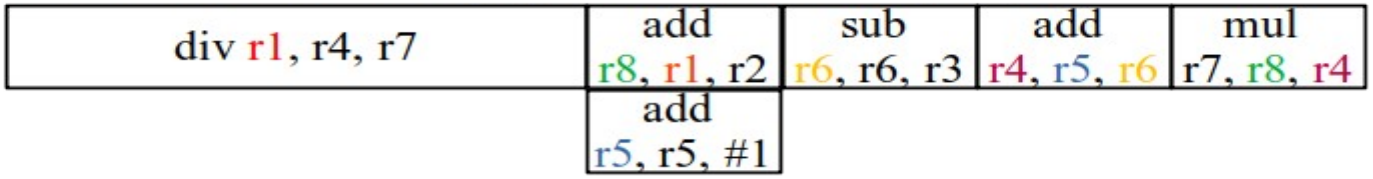


Последователна програма

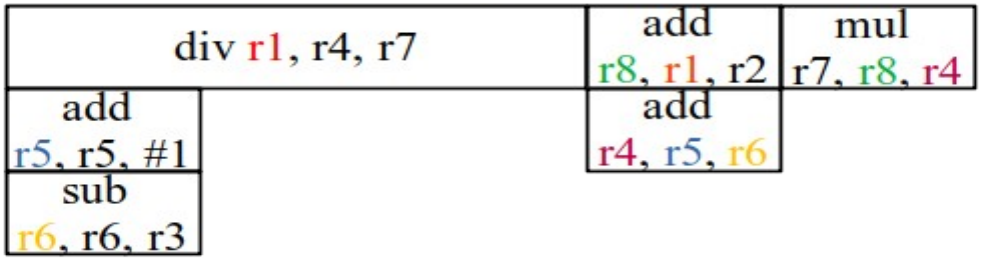
Линейно изпълнение, скаларен μPU



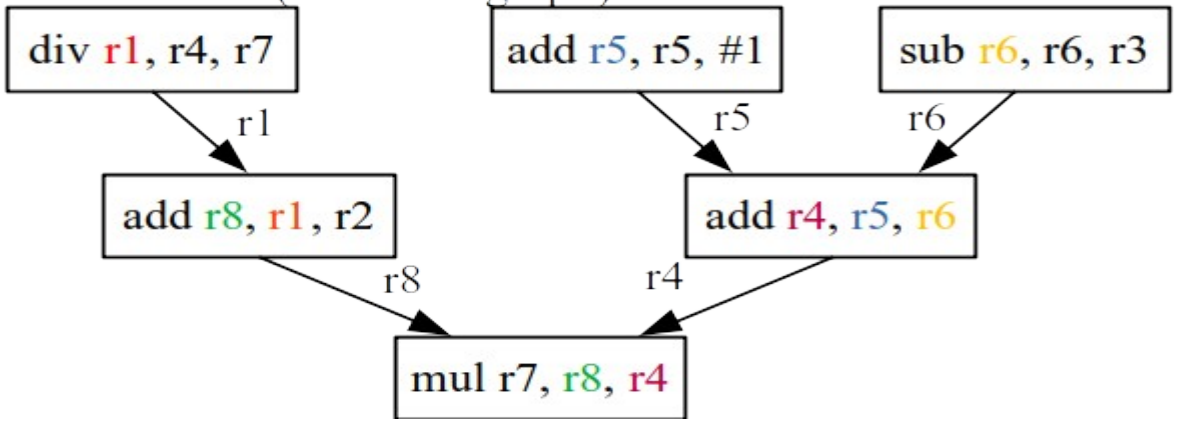
Линейно изпълнение, двупътен суперскаларен μPU



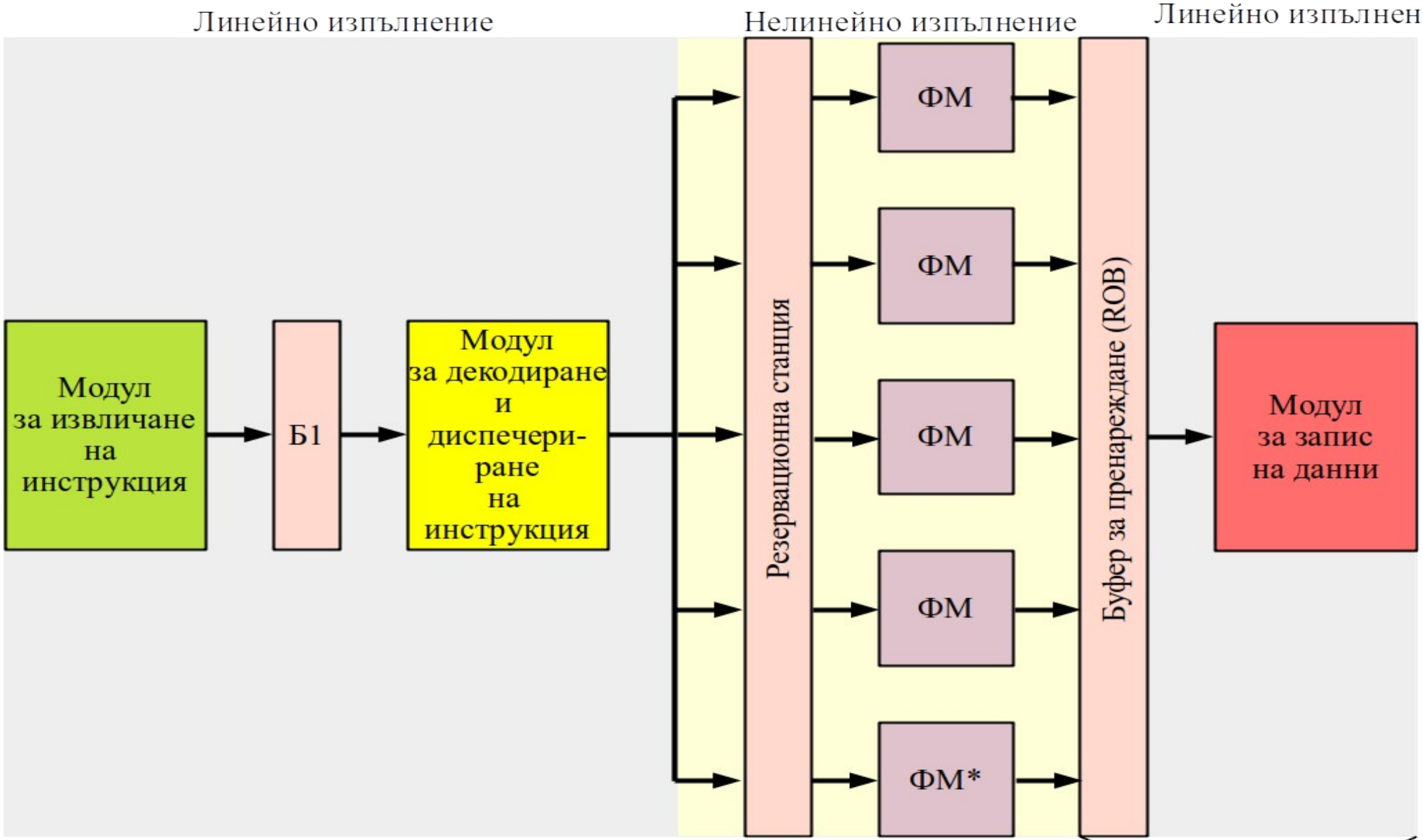
Нелинейно изпълнение



Граф на данновите зависимости (data flow graph)



Линейно и нелинейно изпълнение



Линейно и нелинейно изпълнение

Резервационни станции (reservation station) – буфери за инструкции, в които се чака пристигането на операндите и/или се чака освобождаването на функционални модули, които да ги изпълнят. Веднага щом всички операнди на инструкцията пристигнат и има свободен ФМ модул, инструкцията се изпълнява. След изпълнението се пускат следващите инструкции, чиито **операнди зависят** от предишните.

Буфер за пренареждане (**Re**o**r**der **B**uffer, ROB) – буфер, в който се подреждат инструкциите както са били в оригиналната програма и резултатите им се валидират (commit) като се записват в линеен ред (in-order).

ROB + изходния WR модул се разглеждат като едно и се наричат още модул за валидиране на данни (commit unit). Модулът валидира данните, само когато **всички зависими инструкции** са приключили изпълнението си.

Линейно и нелинейно изпълнение

Зависимостта на операндите в инструкциите може да бъде категоризирана в 3 групи:

*четене-след-запис (**R**ead-**a**fter-**W**rite, RaW)

*запис-след-четене (**W**rite-**a**fter-**R**ead, WaR)

*запис-след-запис (**W**rite-**a**fter-**W**rite, WaW)

Линейно и нелинейно изпълнение

Фазата “декодиране” при OoO архитектурите извършва една допълнителна операция, наречена **преименуване на регистрите** (register renaming). С нейна помощ се разрешават “фалшиви” зависимости на операнди. Това се постига с дублиране на работните регистри на ядрото и записване на операндите временно в дубликатните регистри. След изпълнението на инструкциите в паралел, резултатите се записват в оригиналните работни регистри по време на WR фазата.

Пример – фалшива зависимост на операнди – инструкцията add чете от регистър 1, инструкцията sub записва в регистър 1. Двете инструкции не са зависими, но ако първо се изпълни sub, то add ще сгреши [1].

```
add r3, r1, r2  //r3 = r1 + r2
sub r1, r4, r5  //r1 =  r4 + r5
```

Литература

[1]J. Stokes, “Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture”, Ars Technica Library, 2007.

[2]J. Yiu, “The Definitive Guide to ARM Cortex-M3”, Elsevier, 2007.

[3] D. Kaeli, P. Emma, “Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns”, ACM SIGARCH Computer Architecture News, 1991, DOI: 10.1145/115952.115957

[4] J. Hennessy, D. Petterson, “Computer Architecture: a Quantative Approach”, Elsevier, 2007.

[5] C. Hamacher, Z. Vranesic, S. Zaky, “Computer Organization”, 5th Edition, ISBN 0072320869, pp.453-510, Mc Graw Hill, 2002.

[6] Y. Etsion, “Out-of-order Execution”, Computer architecture, online, https://iis-people.ee.ethz.ch/~gmichi/asocd/addinfo/Out-of-Order_execution.pdf