

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!



Съдържание

1. Изпълнение на инструкцията
2. Конвейери
3. Блокиране на конвейерите
4. Конвейери и програмен брояч
5. Предсказване на преходите (branch prediction)
6. Спекулативно изпълнение (speculative execution)

Изпълнение на инструкцията

Вътрешната структура на един μ PU 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

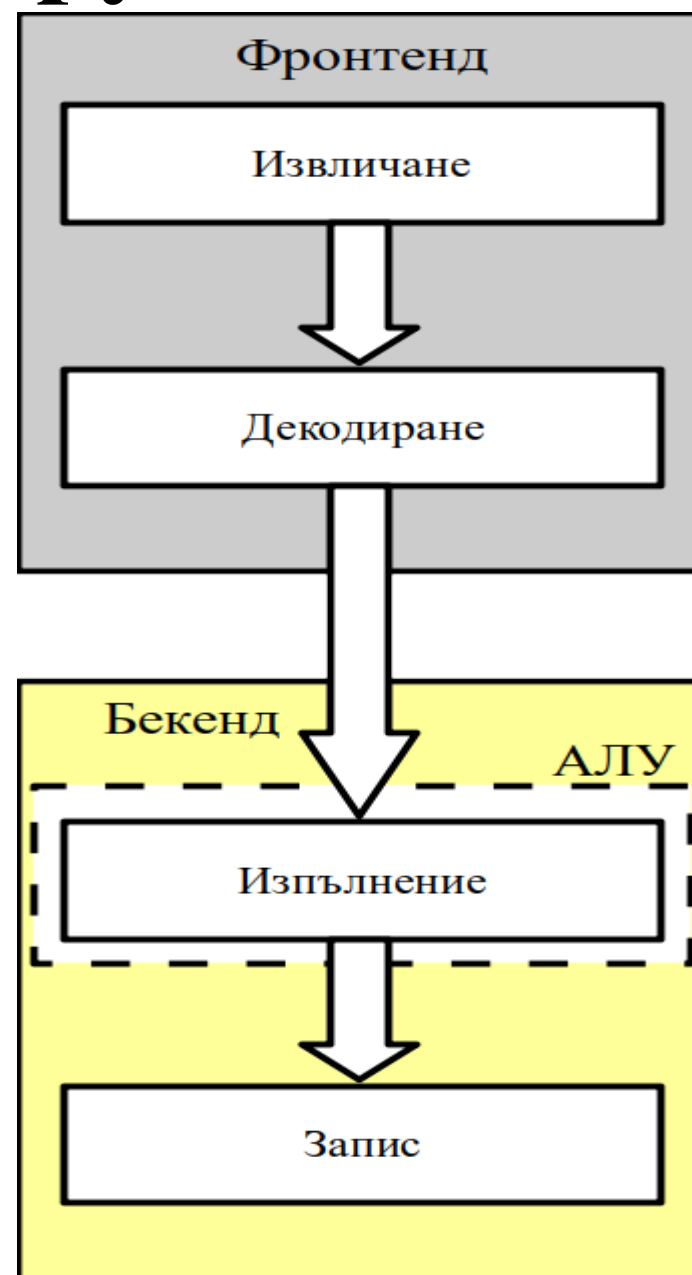
Изпълнението на една инструкция минава през /поне/ 4 фази:

- *извличане (**I**nstruction **F**etch, IF)
- *декодиране (**I**nstruction **D**ecode, ID)
- *изпълнение (**E**xecute, EX)
- *запис на резултат (**W**rite, 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 (Write Back, WB)** – след завършване изпълнението на инструкцията, резултатът се записва обратно в регистровия файл на ядрото (или още - работните регистри, GPR);

***външен запис (Memory 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.

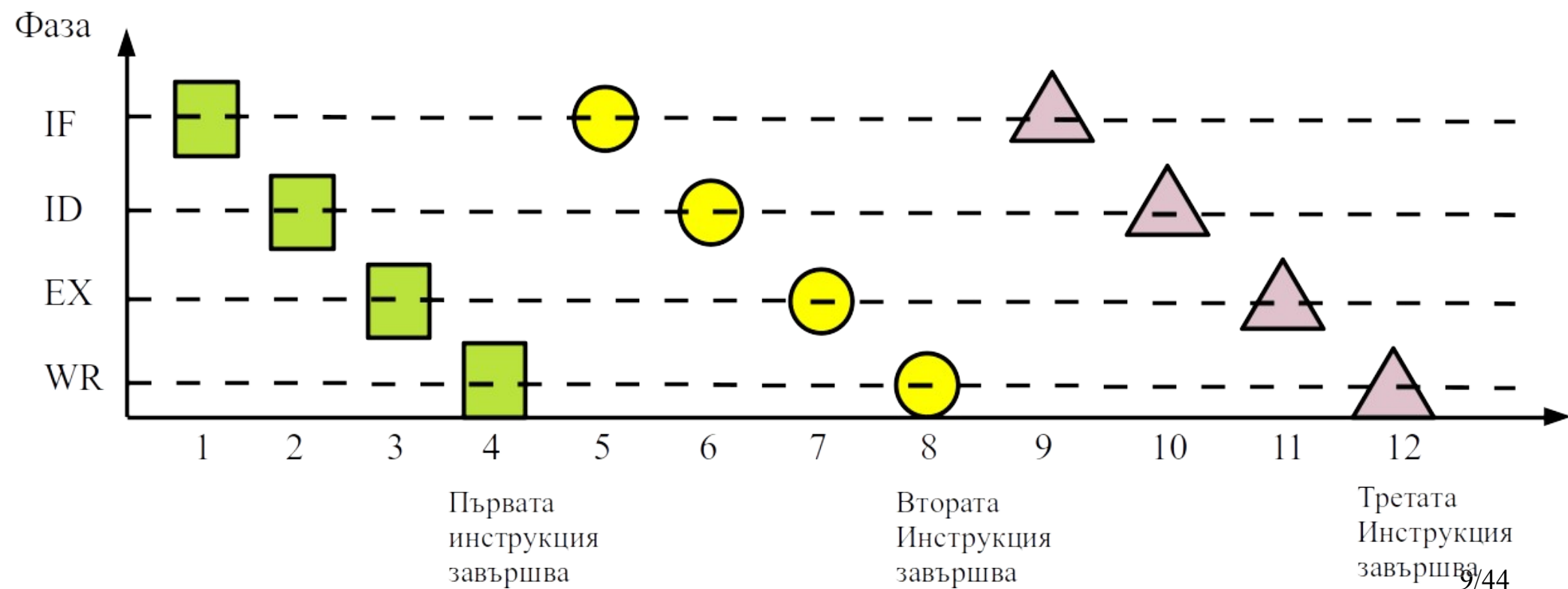
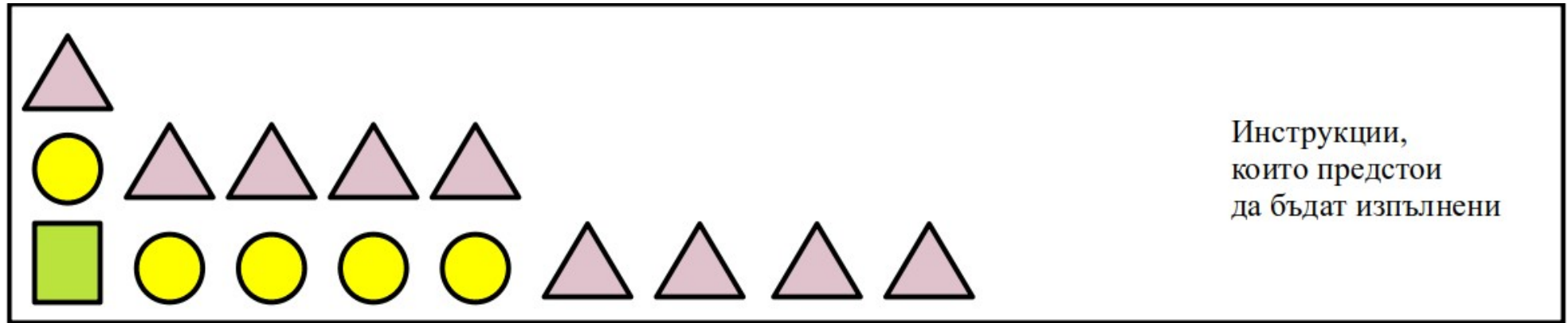
Забележете, че по време на всеки такт само е активна само една от фазите.

Такъв вид изпълнение води до много ниско работно натоварване на отделните модули на μ PU, 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 μ PU, 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 **повисока производителност** без да се увеличава работната честота на ядрото.

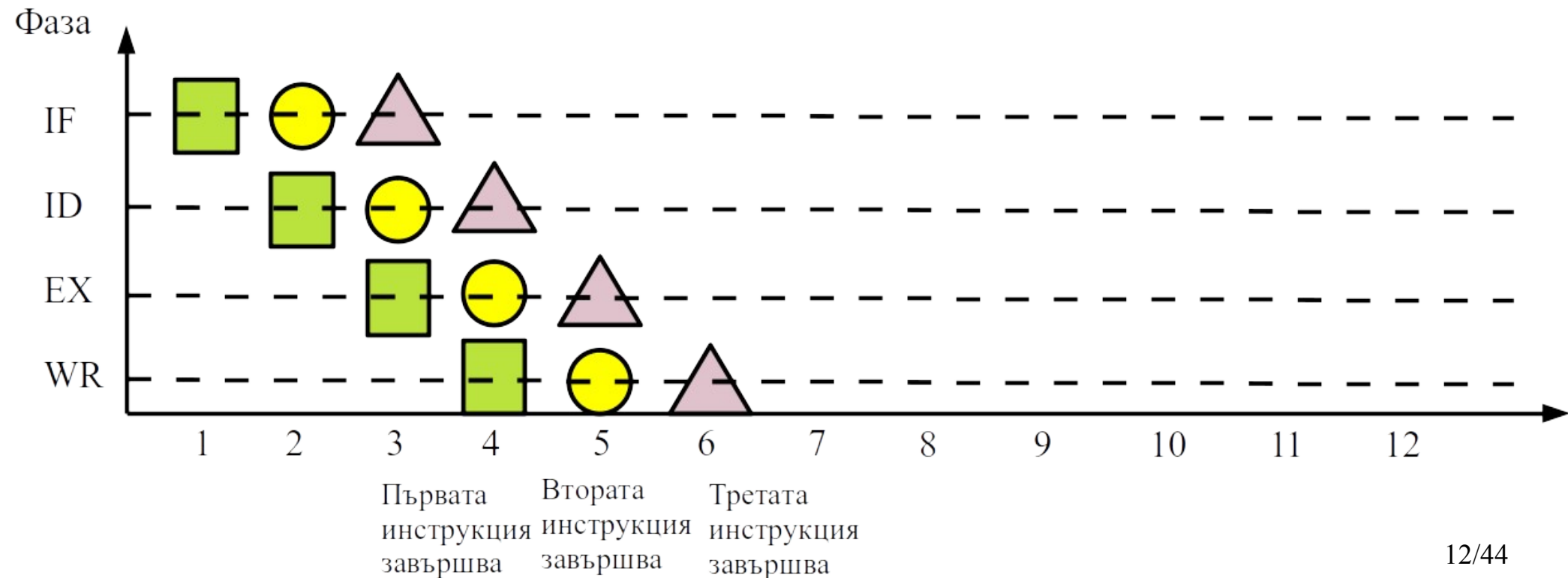
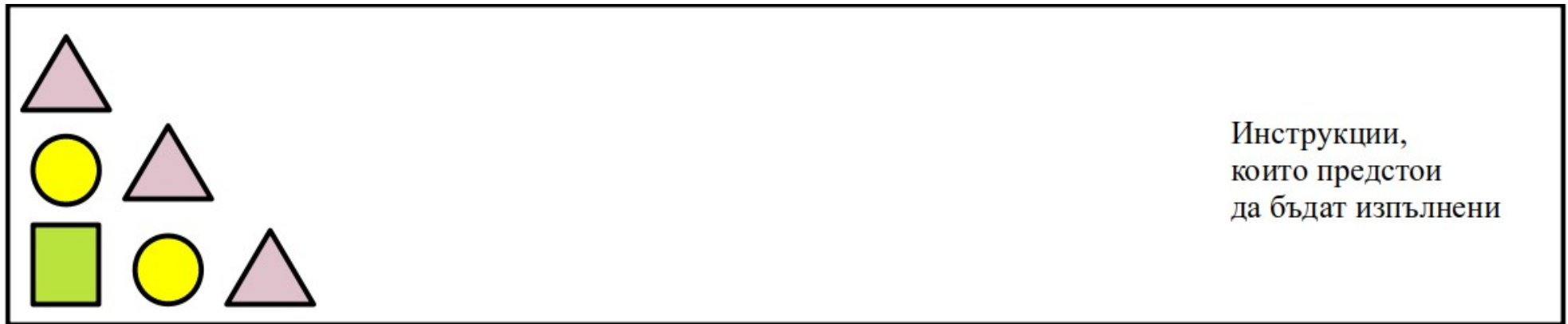
Конвейери

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 извличана (IF3).

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

Конвейери



Конвейерно изпълнение на инструкция

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).

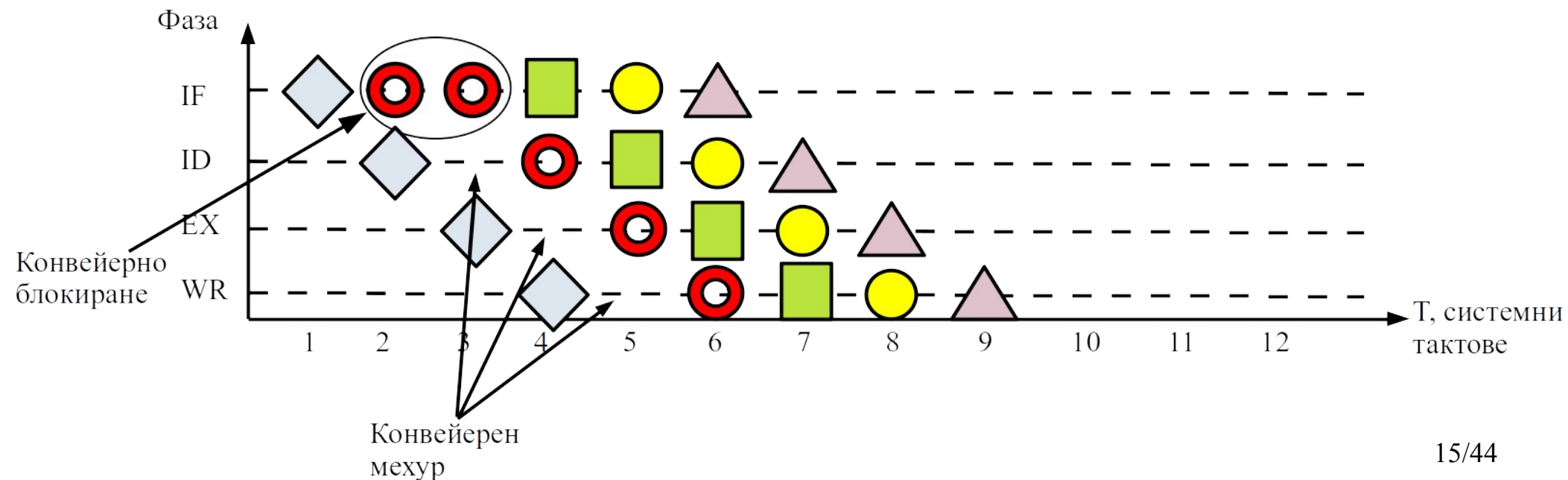
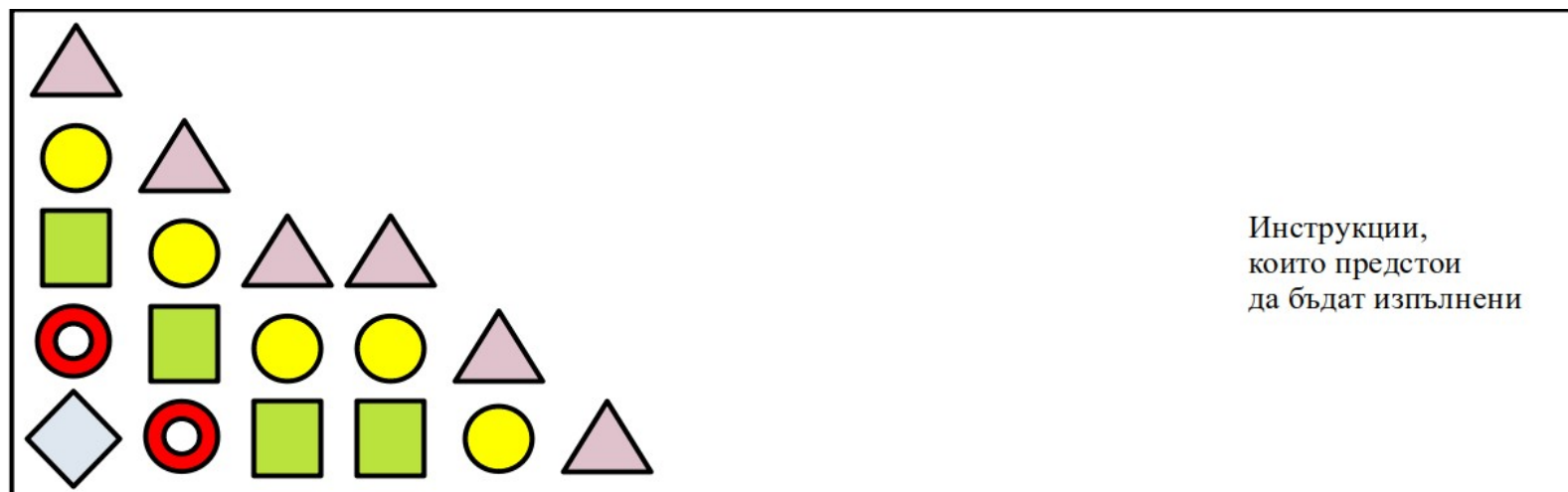
*Когато една инструкция се забави, забавят се и всички инструкции след нея.

*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 μ PU.

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 μ PU при 2-тактово и 10-тактово блокиране на конвейера.

Производителността на един μ PU може да се измери с параметрите:

- *тактове-за-инструкция (**Cycles Per Instruction, CPI**)

- *инструкции-за-такт (**Instructions Per Cycle, IPC**)

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

Блокиране на конвейерите

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

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

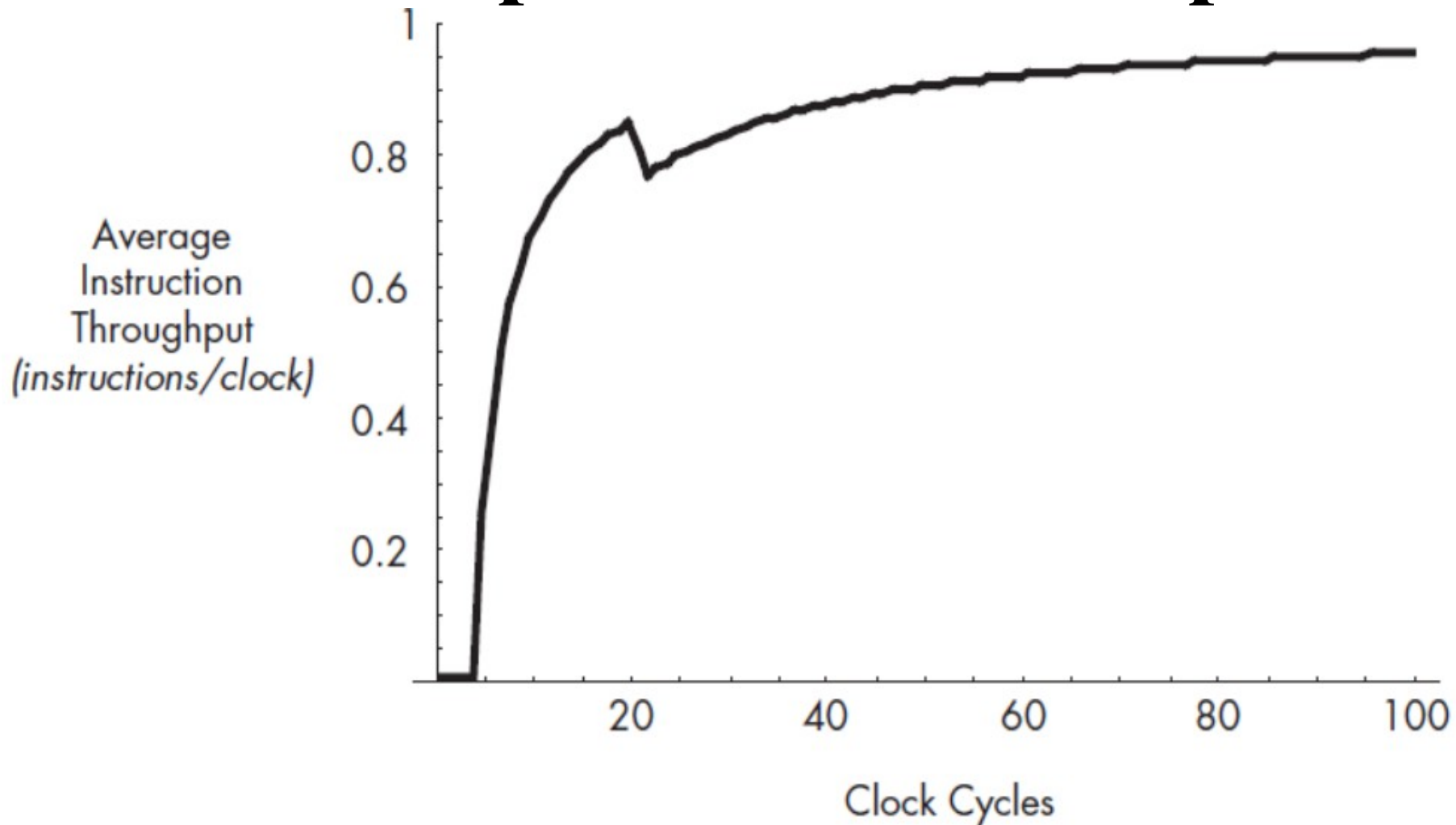


Figure 3-11: Average instruction throughput of a four-stage pipeline with a two-cycle stall

Блокиране на конвейерите

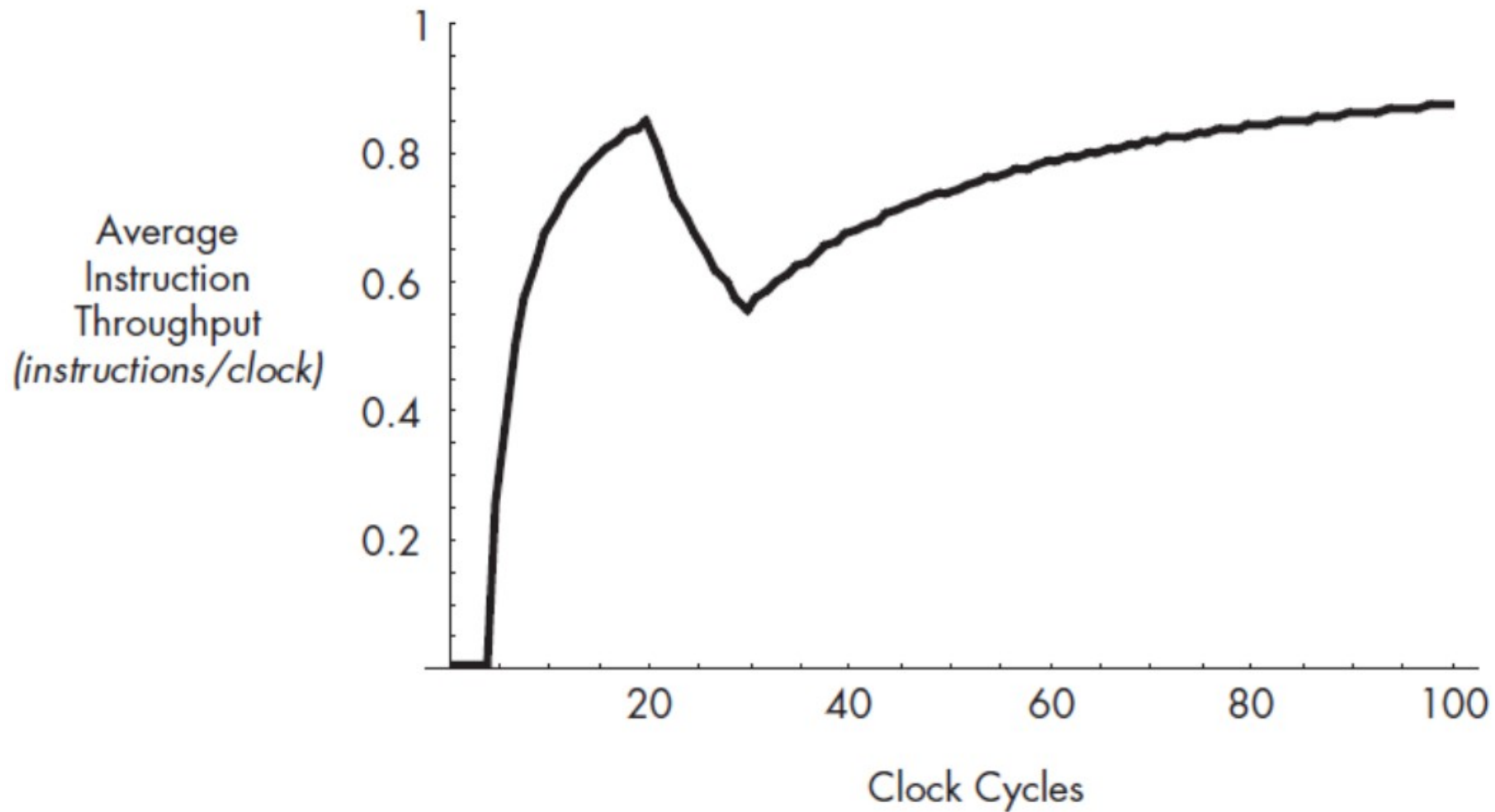


Figure 3-12: Average instruction throughput of a four-stage pipeline with a 10-cycle stall

Блокиране на конвейерите

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

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:

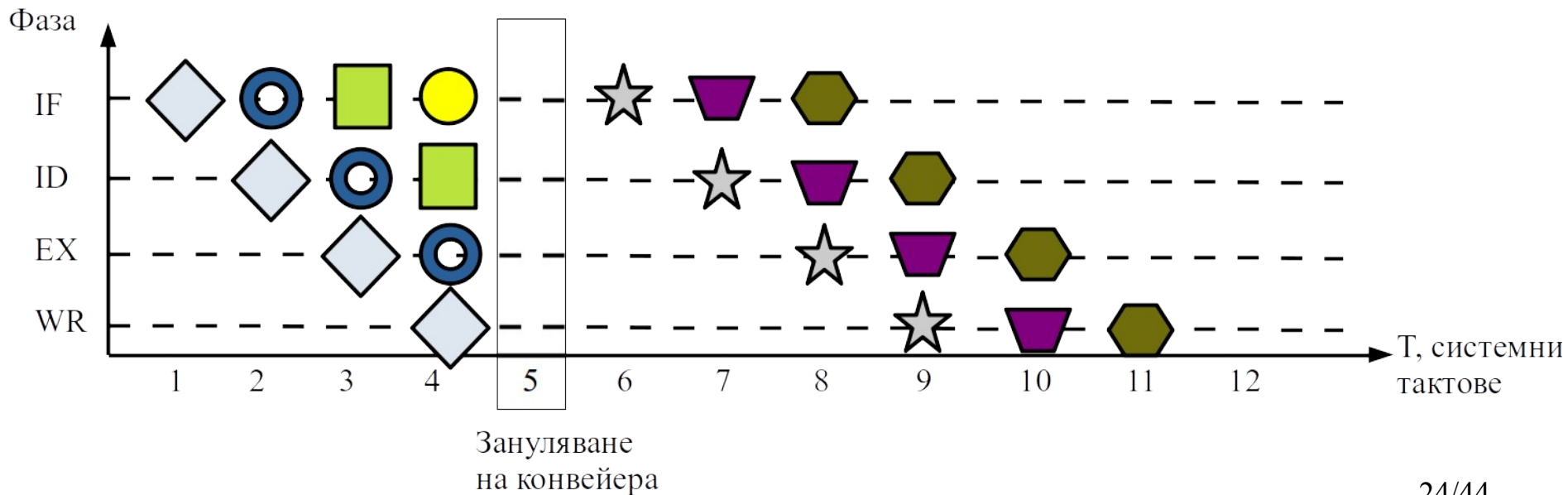
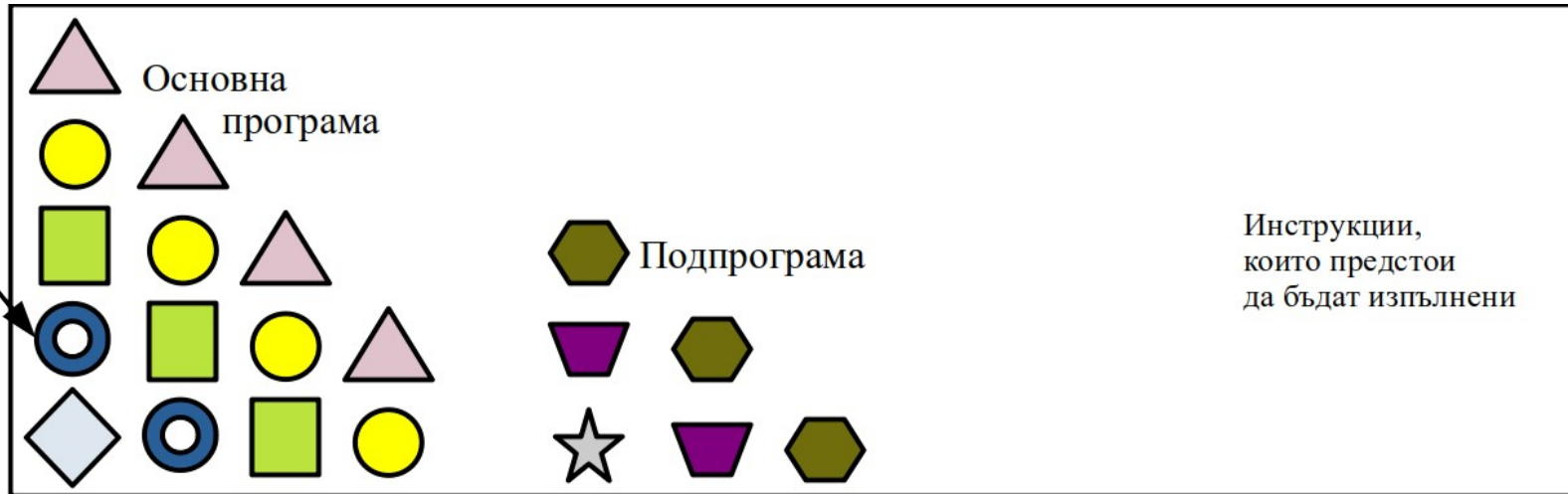
***чрез използване на инструкции за разклонения в програмата (branch instructions)** [използва се също термина условен преход]

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

***чрез използване на специализирани инструкции**

Блокиране на конвейерите

Условен
преход /
безусловен
преход /
специална
инстр.



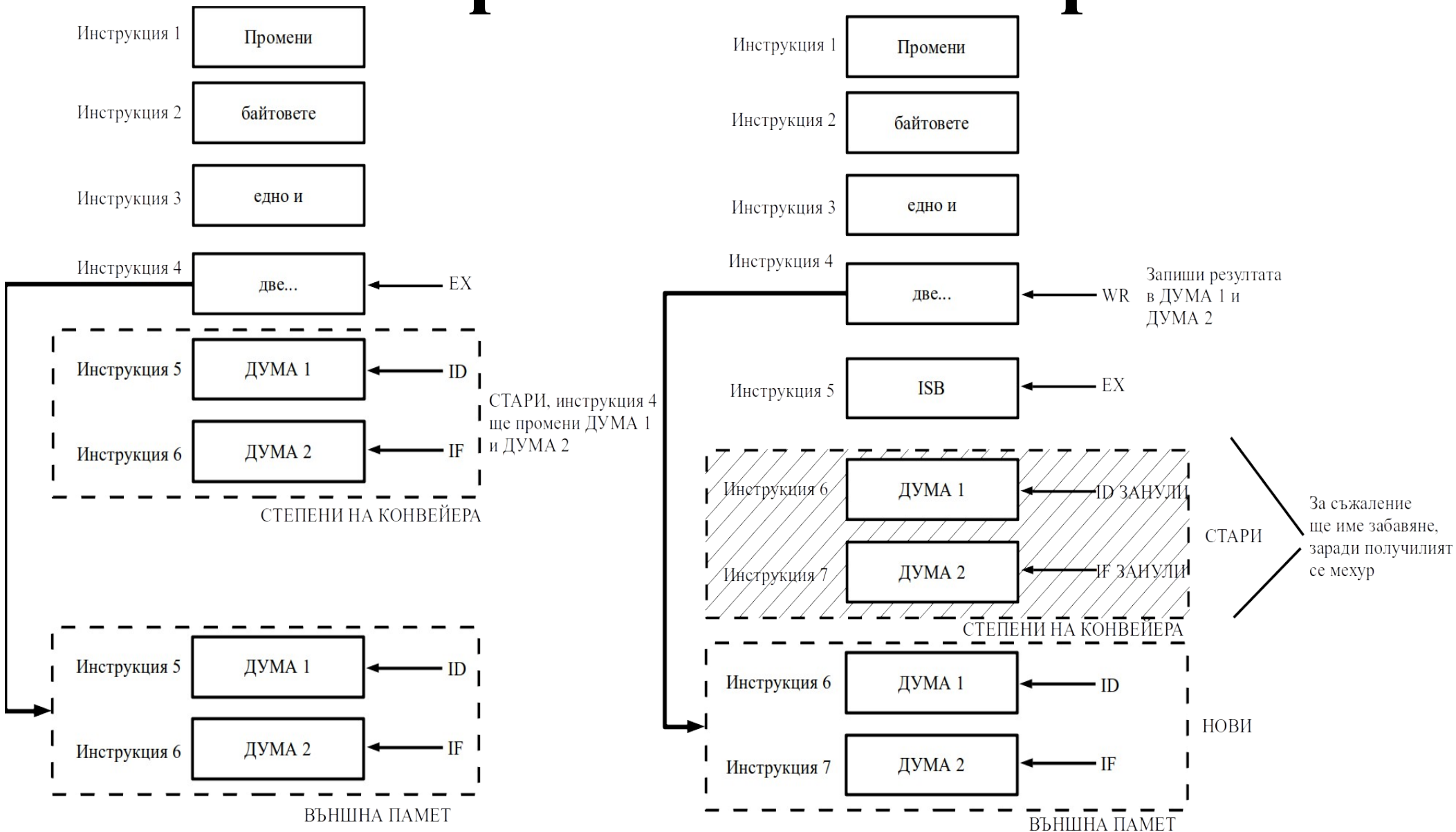
Блокиране на конвейерите

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

Пример – инструкцията на ARM Cortex – ISB (Instruction Synchronization Barrier). Тя занулява конвейера и подсигурява, че инструкциите след нея (на по-големи адреси) ще бъдат заредени наново. Използва се в саmomодифициращ се код, например в приложения с изкуствен интелект.

На следващия слайд е показан пример със саmomодифициращ се код.

Блокиране на конвейерите



Конвейери и програмен брояч

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

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

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

Конвейери и програмен брояч

При конвейерните μ PU 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 $\Rightarrow n=1$, 16-битови $\Rightarrow n=2$, 32-битови $\Rightarrow n=4$, и т.н.).

Пример – ако степените на конвейера са:

1.IF

2.ID

3.EX

4.WB

програмният брояч, ще се увеличава с $2 \times n$.

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 $5 \times n$.

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).

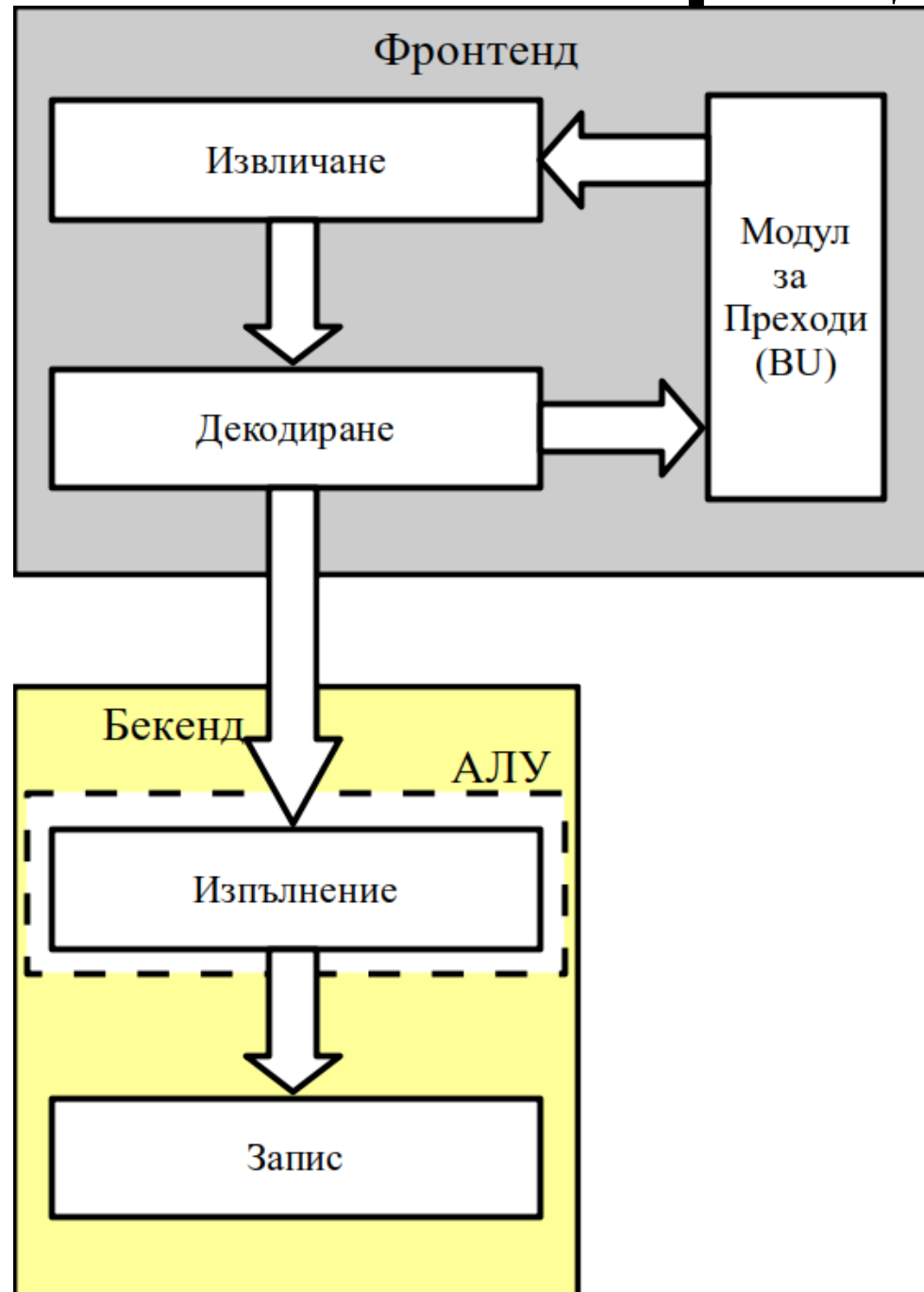
Модул за преходи (Branch Unit, BU) – част от фронтенда на μPU , който работи в синхрон с програмния брояч PC, и който управлява модулите за извличане и декодиране на инструкции, като ги насочва да работят с различни части от програмата.

Състои се от два подмодула:

- *модул за предсказване на преходи (**Branch Prediction Unit, BPU**)

- *модул за изпълнение на преходи (**Branch Execution Unit, 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 (Branch Prediction Unit, BPU) – хардуерен модул на μ PU, който се опитва да предскаже резултата на инструкция за условен преход и изчислява адреса на (предвидения) прехода.

Предсказването 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 случаите 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 ВРУ използва статично предсказване, то на примера от миналия слайд целевия адрес ще бъде верен в 1000 пъти от случаите и само веднъж ще сгреша (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 (Branch History Table, 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 (Branch Target Buffer, 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 ЕХ 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 ЕХ фазата (evaluate).

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

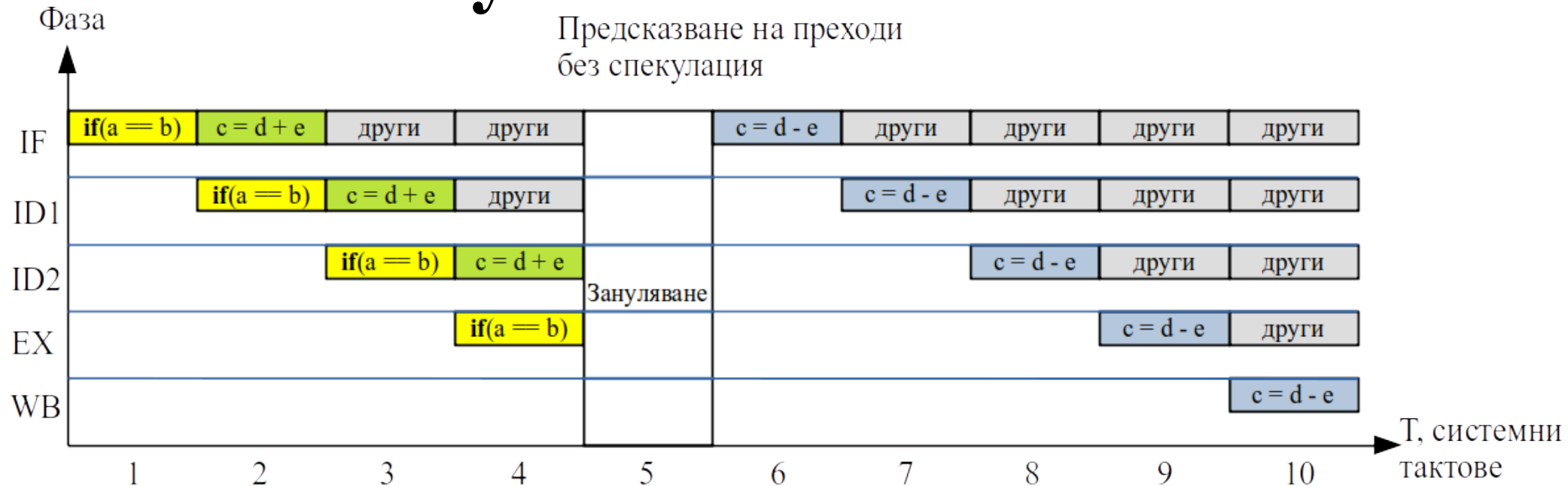
Спекулативно изпълнение

Пример:

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

Нека предположим, че $a \neq b$, тогава се получава графиката от следващия слайд.

Спекулативно изпълнение



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. Pettersen, “Computer Architecture: a Quantative Approach”, Elsevier, 2007.