Състезания между инструкциите при конвейерно изпълнение и тяхното преодоляване

Съдържание:

1. Преглед на видовете състезания;
2. Структурни състезания. Начини за преодоляването им;
3. Зависимости по данни. Видове. Стратегии за тяхното преодоляване.

1. Видове състезанията между инструкциите при конвейерно изпълнение

В предишната лекция посочихме видовете състезания между инструкциите при тяхното конвейерно изпълнение. Сега ще ги повторим накратко.

Състезанията са конфликтни ситуации, които не позволяват да се стартира следващата инструкция в следващия процесорен цикъл. Те възникват вследствие на зависимости, които съществуват между инструкциите по отношение на:

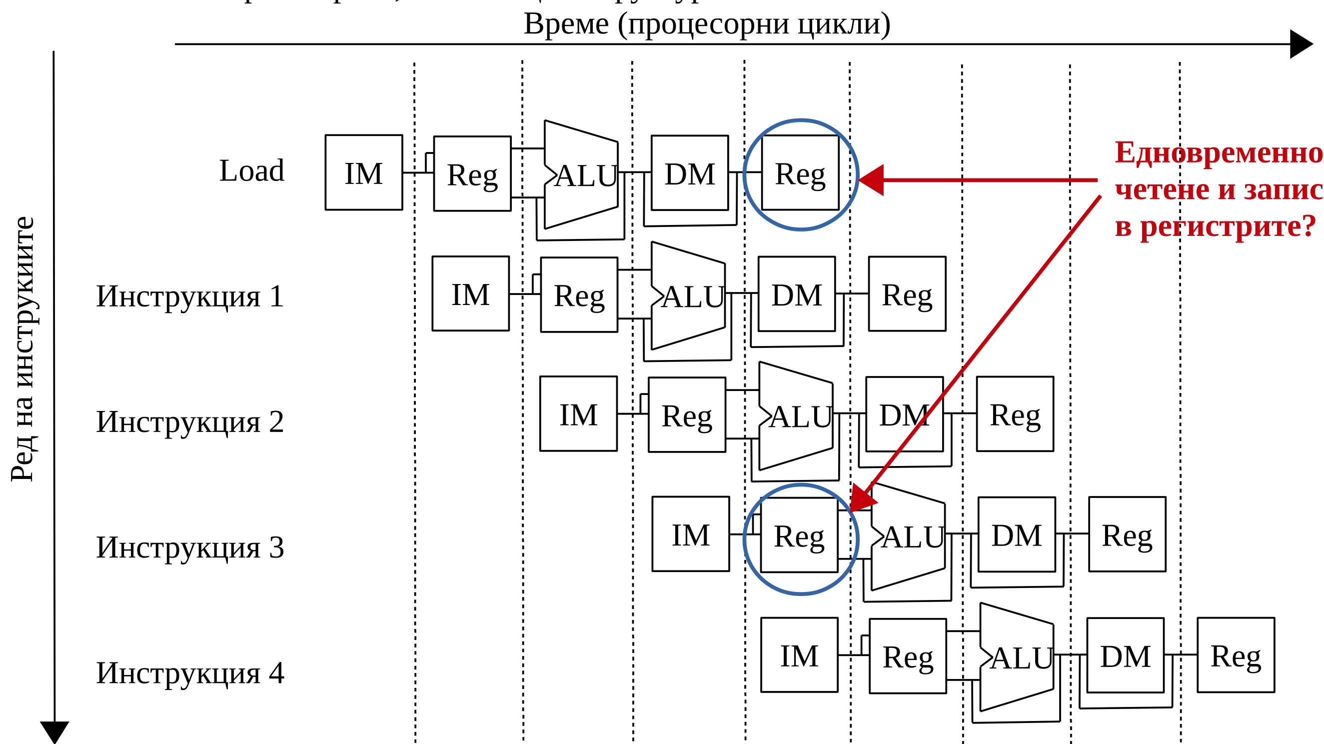
- ***ресурси*** (структурни зависимости) – необходимият ресурс е зает, напр. от него се нуждаят две инструкции, изпълняващи се на различни етапи в конвейера.
- ***данни*** (зависимости по данни) – дължат се на зависимостта между някои инструкции по отношение на използваните операнди. Например, резултатът от една инструкция се използва като входен операнд в следващата. Необходимо е да се изчака докато първата инструкция запише своя резултат.
- ***управление*** (процедурни зависимости) – дължат се на инструкциите за преход, които управляват последователността на инструкциите в изпълнението на програмата.

2. Структурни състезания

Когато изпълнението на дадена комбинация от инструкции в конвейера води до едновременен достъп до даден хардуерен елемент е налице конфликт или състезание за заемане на един физически ресурс. За преодоляване на структурните състезания са възможни две решения:

1. Забавяне, чрез внасяне на празни процесорни цикли преди някои инструкции с цел изчакване докато ресурсът бъде освободен;
2. Добавяне на хардуерни елементи, върху които да се разпредели едновременният достъп от две или повече инструкции.

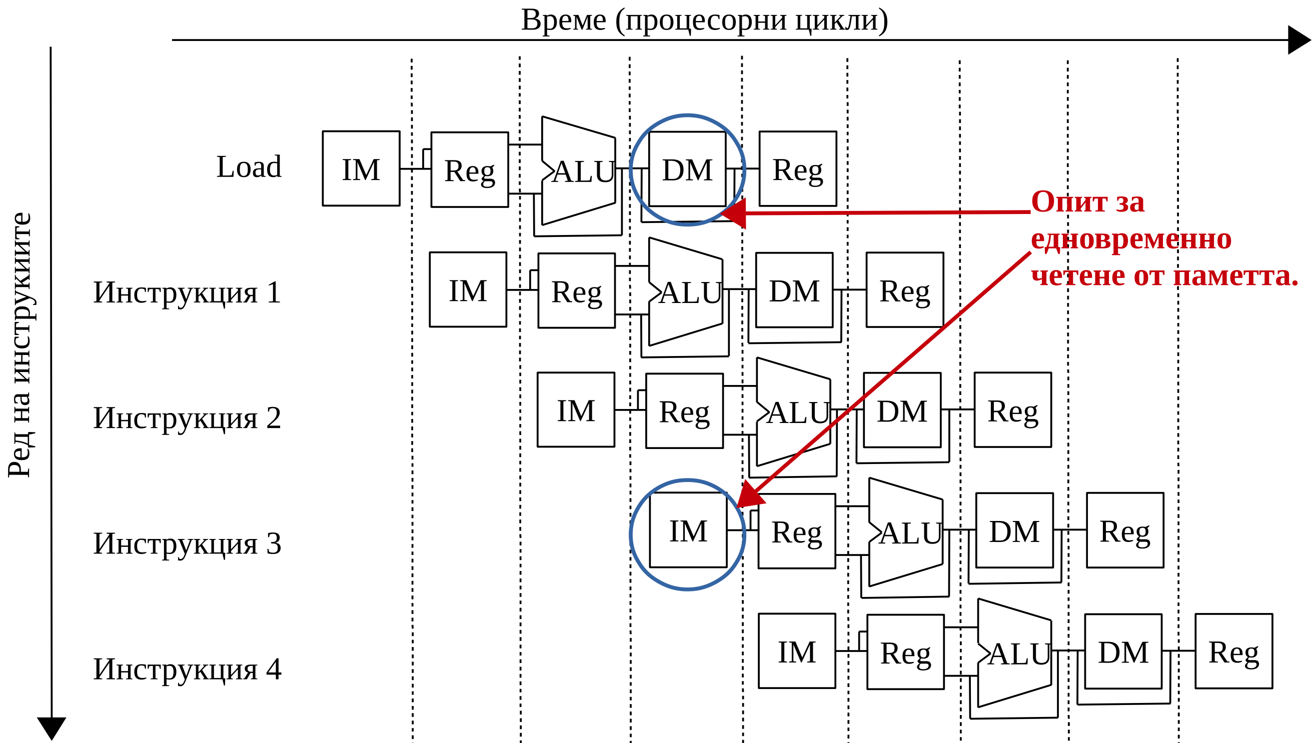
На следващата фигура е представена ситуация, при която има едновременен достъп за четене и запис в регистрите, т.е. налице е структурно състезание.



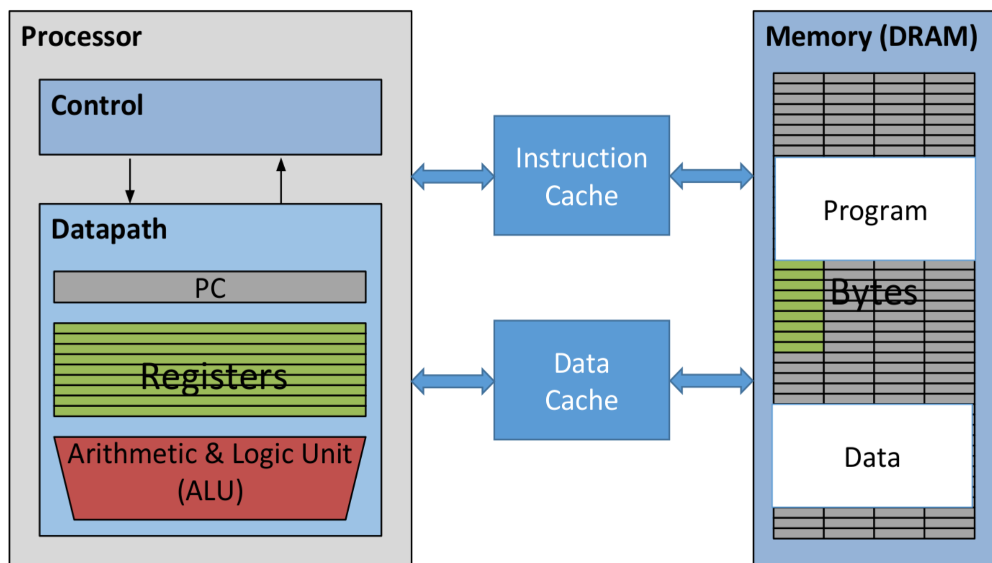
За да се преодолее това състезание за достъп до регистровия файл, може да се подходи по два начина:

- Да се реализират отделни „портове“ за достъп – два независими порта за четене и един независим порт за запис. По този начин могат да се четат едновременно два операнда и да се записва една стойност. Така всяка инструкция може да прочете до два операнда в етапа за декодиране (ID) и да запише една стойност в етапа (WB);
- Да се раздели четенето от записа, така че да се по различно време – записът да се извършва в първата половина на процесорния цикъл, а четенето във втората.

Друг пример за структурно състезание е представен на фигурата по-долу.



В един и същ процесорен цикъл инструкцията *Load* осъществява достъп до паметта, за да прочете данни, докато друга инструкция бива извличана от паметта. За преодоляване на този вид структурно състезание се обособяват два вида кеш памети – за инструкции и за данни, както е представено на фигурата.



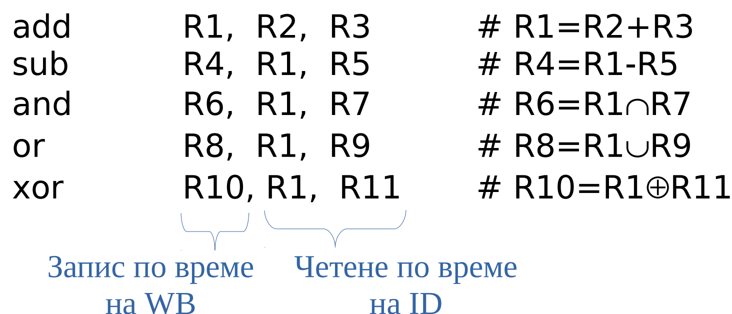
Забележка: Кеш паметта е малка и бърза буферна памет.

3. Зависимости по данни. Видове

Припокриването между инструкциите при тяхното изпълнение може да доведе до промяна в последователността за запис и четене на данните, които са общи за две или повече инструкции, намиращи се на различни етапи от своето изпълнение в конвейера. Това създава опасност от използване на остарели или неправилно модифицирани данни. (Тъй като припокриването на инструкциите в конвейера може да промени редът за четене/запис на операндите спрямо реда при последователното им неконвейерно изпълнение). Да приемем, че в една програма инструкцията *i* се намира преди инструкция *j* и че двете инструкции използват регистъра *x*. Има три различни типа зависимости по данни, които могат да възникнат между *i* и *j*:

- **Зависимост от типа RAW** (read after write – четене след запис): най-често срещаната. Случва се когато инструкция *j* чете от регистър *x* преди инструкция *i* да запише в него. Ако това състезание не бъде предотвратено, инструкция *j* ще използва старата стойност в *x*.
- **Зависимост от типа WAR** (write after read – запис след четене). Възниква, когато инструкция *i* чете от регистър *x* след като инструкция *j* е записала в него. В този случай инструкцията *i* ще използва грешна стойност от *x*. Състезанията от типа WAR са невъзможни в класическия петстепенен конвейер за работа с цели числа. Те възникват, когато се използва конвейерна обработка с динамично планиране (пренареждане на инструкциите към момента на тяхното изпълнение).
- **Зависимост от типа WAW** (write after write – запис след запис). Възниква, когато инструкция *i* записва в регистър *x* след като инструкция *j* е записала в него. Когато това се случи, регистър *x* ще съдържа грешна стойност. Състезанията от типа WAW също са невъзможни в класическия петстепенен конвейер за работа с цели числа, но те възникват, когато инструкциите се пренареждат или когато времената за изпълнението на различните видове инструкции не са еднакви.

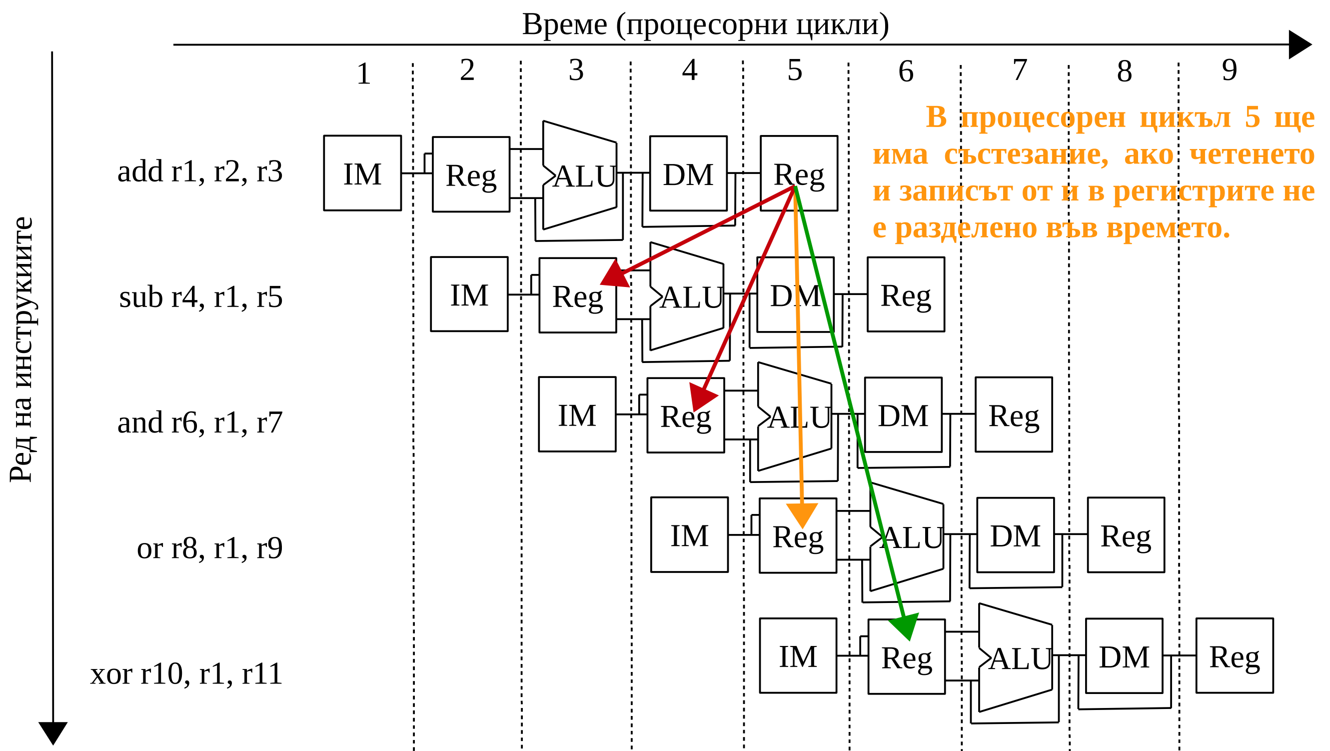
Нека разгледаме конвейерното изпълнение на следващите инструкции:



Всички инструкции след *add* използват резултата от нея. Както е показано на фигурата по-долу, инструкцията *add* записва резултата в *R1* на последния етап от конвейера – *WB*, но инструкцията *sub* чете от *R1* на етапа *ID*, което създава зависимост от типа *RAW*. Ако не се вземат предпазни мерки за предотвратяването му, инструкцията *sub* ще прочете грешна стойност и ще се опита да я използва.

Инструкцията *and* също създава зависимост от типа RAW. Както можем да видим от фигурата, записът в *R1* не завършва до края на петия процесорен цикъл. По този начин инструкцията *and*, която чете от регистрите по време на четвъртия процесорен цикъл, ще получи грешни данни.

Инструкцията *xor* работи правилно, тъй като тя чете от регистъра през процесорен цикъл 6, след като записа в *R1* вече е направен. Инструкцията *or* ще работи без да създава състезание, ако четенето от регистрите е през втората част на процесорния цикъл, а записът през първата.



3.1. Предотвратяване на състезанията за данни посредством застои

Първото решение, което може да се изпълни, за да се предотвратят състезанията, в представения по-горе случай, е да се внесат празни процесорни цикли след инструкцията add, както е показано на следващата фигура.

Време (процесорни цикли)

Ред на инструкиите

add r1, r2, r3

nop

nop

sub r4, r1, r5

and r6, r1, r7

or r8, r1, r9

xor r10, r1, r11

IM Reg ALU DM Reg

bubble

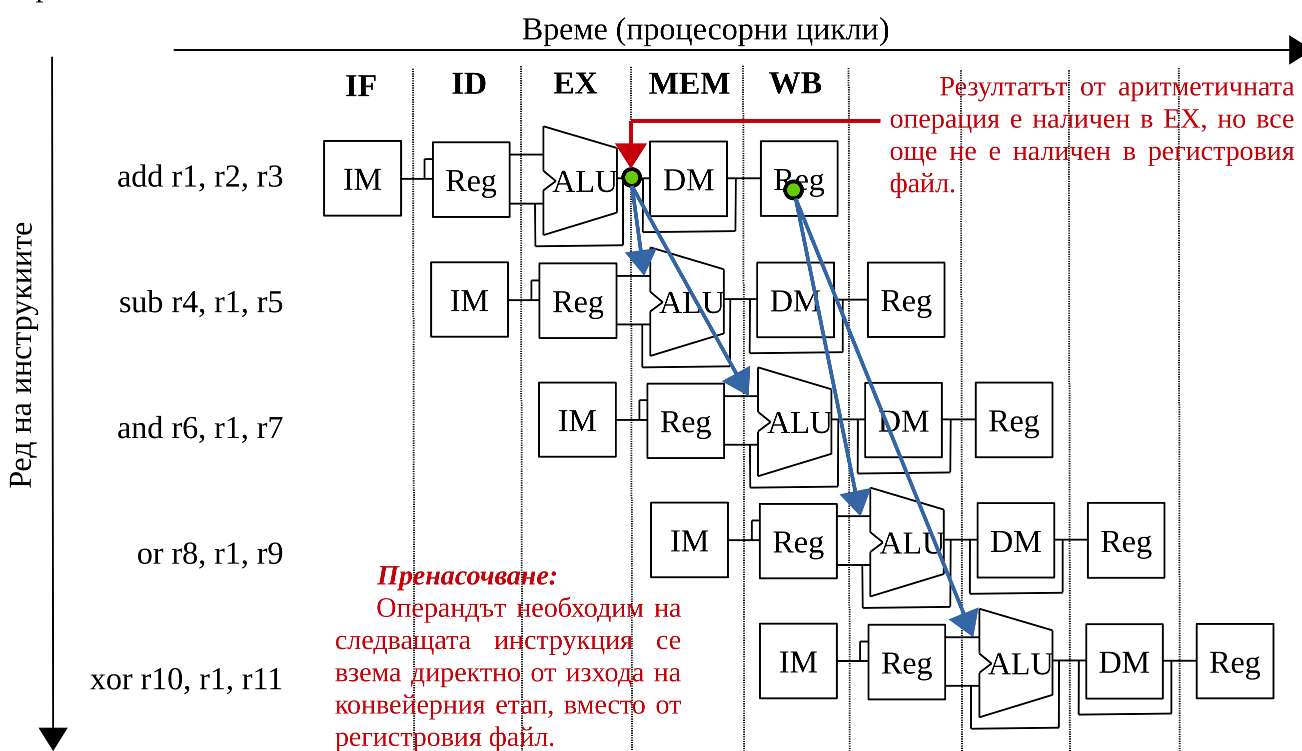
Състезанието се предотвратява, при условие, че четенето от регистрите е през втората част на процесорния цикъл, а записът през първата.

Внасянето на празни процесорни цикли се осъществява като се добави празната инструкция nop в кода на програмата.

Застоите намаляват производителността на конвейера, но са необходими за да се получи коректен резултат. За избягване на застоите могат да се използват софтуерни и хардуерни средства. Софтуерните средства са от страна на компилатора, който може да пренареди инструкциите в кода, за да се избегнат състезанията и застоите. Това изисква познания за структурата и функционирането на конвейера. При хардуерните решения, в конвейера се добавят схеми за пренасочване на данните. Чрез схемите за пренасочване резултатът от една инструкция се предоставят там където е необходим на следващите.

3.2. Избягване на застоите, предизвикани от зависимостите по данни чрез пренасочване

Представените на фигурата в точка 3 зависимости по данни могат да се предотвратят, чрез добавяне на пренасочване, така че резултатът от сумирането (инструкция *add*) да се запише и там където е необходим на следващата инструкция *sub*, избягвайки по този начин вмъкването на застои в работата на конвейера. Следващата фигура показва как работи пренасочването.



Техниката на пренасочване позволява резултатът да се използва веднага след като бъде изчислен, без да се чака да бъде записан в регистър.

Описаното пренасочване предава резултатът от изхода на функционалното устройство ALU директно към входа на същото, но на пренасочването може да се погледне по-общо, като на техника за предаване на резултат от изхода на функционално устройство в един конвейерен етап директно към входа на друго функционално устройство в друг конвейерен етап, както е в следния пример:

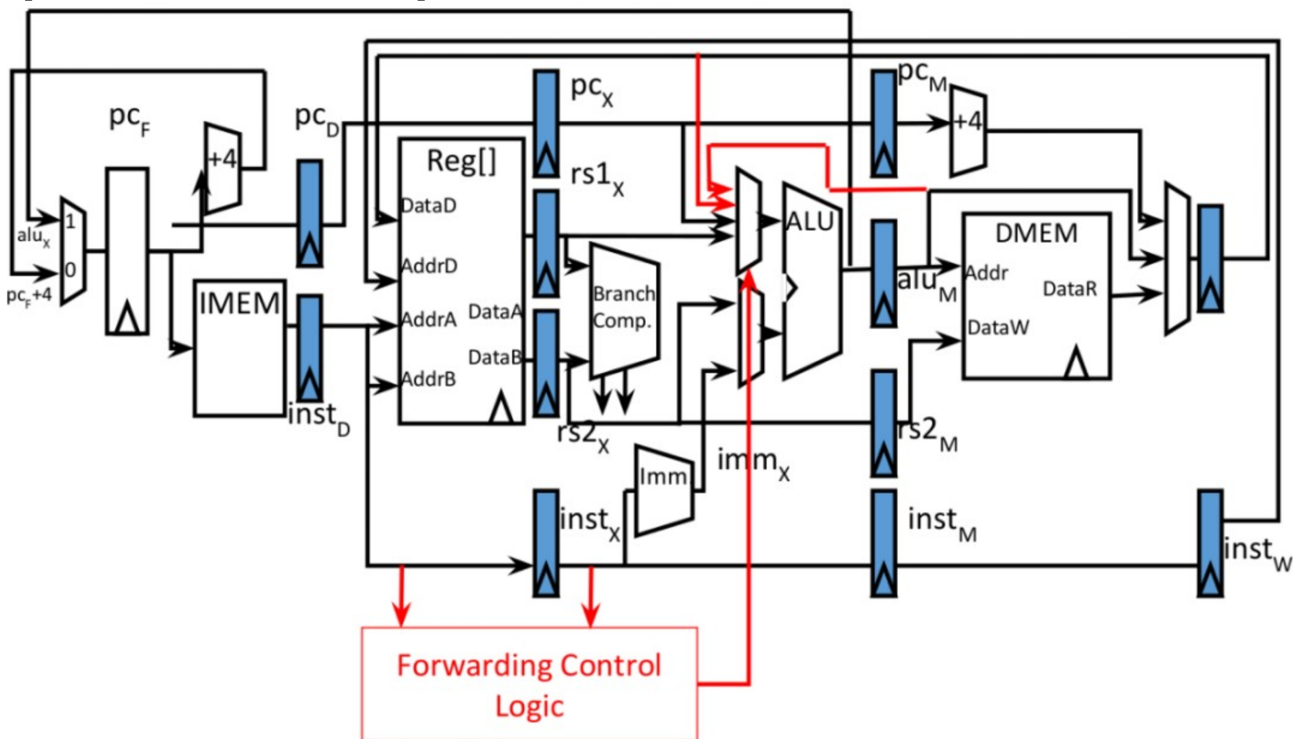
```
add     R1,R2,R3      # R1 = R2 + R3
lw      R4,0(R1)      # R4 = Mem(0 + R1)
sw      R4,12(R1)     # Mem(12 + R1) = R4
```

За да се избегнат застои в тази последователност от инструкции е необходимо да се направи пренасочване на стойностите от изходите на функционалните устройства ALU (аритметично-логическо устройство) и DM (памет за данни) директно към входовете на ALU и DM.

За реализация на пренасочването са необходим допълнителни връзки и схеми по пътя

на данните в конвейера. Например, необходима е схема, която да може да провери дали регистърът, в който ще се запише резултатът от изпълнението на една инструкция е същият като някой от регистрите, в които се намират входните операнди на следващата инструкция. Ако регистрите съвпадат, то същата схема трябва да подаде управляващи сигнали към превключващите схеми (мултиплексори), които да пренасочат данните.

На следващата фигура е представена в блоков вид схема на конвейер с пренасочване. С червено са начертани схемата за управление на пренасочването (Forwarding Control Logic) и двете обратни връзки от изходите елементите ALU и DMEM към входа на ALU или, ако вземем предвид етапите, от изходите на етапите EXE и MEM към входа на етапа EXE. Със синьо са оцветени регистрите между отделните етапи. Пред ALU има начертани мултиплексори. Според подадения към тях управляващ сигнал, те прехвърлят един от входовете си към единствения си изход и така се избира данните от кои регистри да се подадат на входовете на ALU. Единият от мултиплексорите се управлява от схемата за управление на пренасочването. Точно този мултиплексор физически превключва и пренасочва данните в конвейера от изходите на етапите EXE и MEM към входа на етапа EXE.



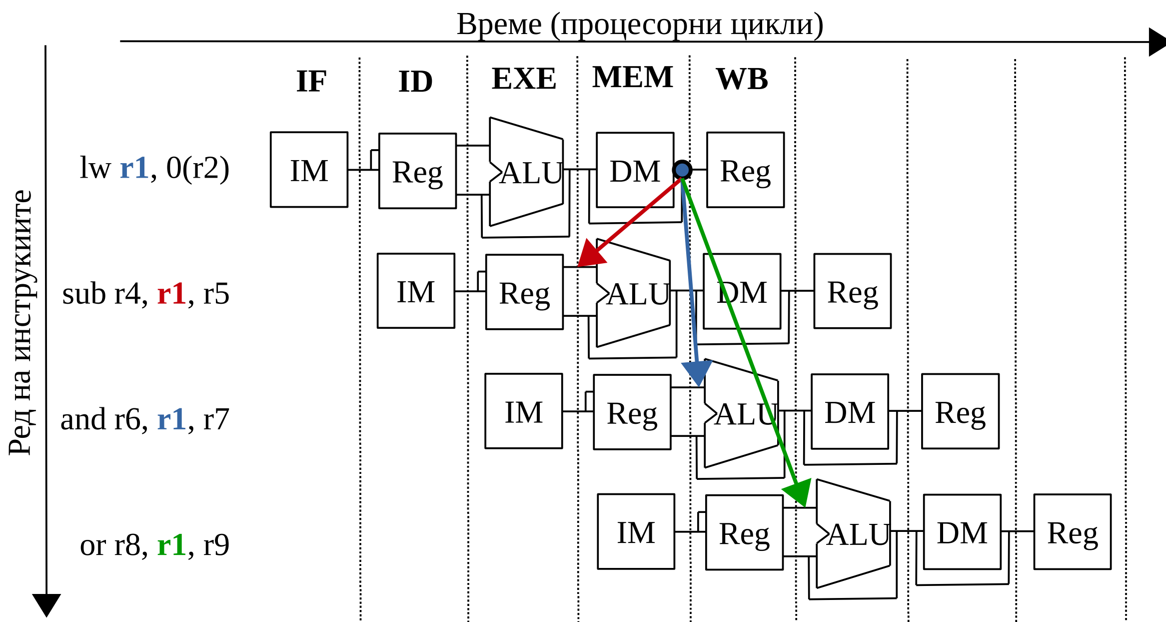
3.3. Зависимости по данни за които са необходими застои

Не всички зависимости по данни могат да се елиминират, посредством пренасочване, както е показано в следващия пример.

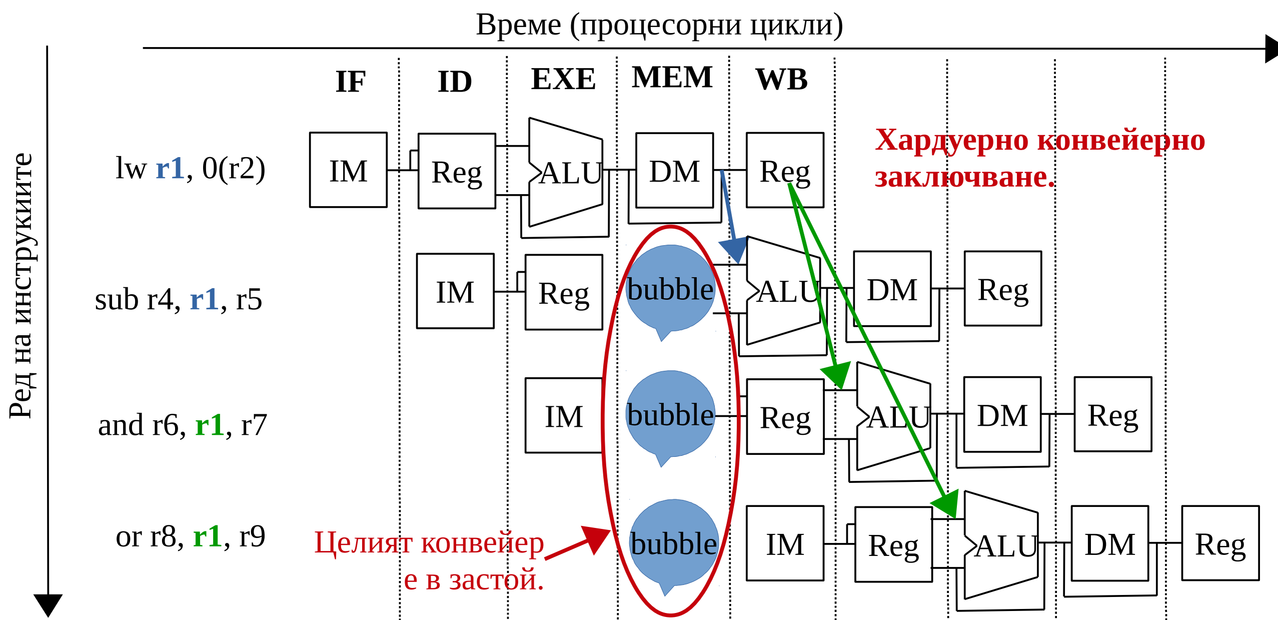
```
lw      R1,0(R2)      # R1 <- Mem(0 + R2)
sub     R4,R1,R5      # R4 <- R1 + R5
and     R6,R1,R7      # R6 <- R1 + R7
or      R8,R1,R9      # R8 <- R1 + R9
```

Конвейерната обработка за този пример е показана на следващата фигура. Този случай е различен от ситуацията с пренасочване на данни между две или повече последователни аритметични инструкции. Инструкцията *ld* прочита своите данни в края на етапът MEM (при който *ld* има достъп и работи с паметта). Инструкцията *sub* трябва да получи своите данни в началото на същия процесорен цикъл. Налице е разминаване във времето. Данните за операцията *sub* са били необходими преди да могат да бъдат прочетени от паметта и няма как да се върнат назад във времето. Както е показано на фигурата по-долу прочетените данни от паметта могат да се пренасочат към входа на ALU, за да бъдат налични в следващия процесорен цикъл, когато ALU ще извършва операцията *and*. За инструкцията *or* не е необходимо да се използва пренасочване, тъй като, тя ще получи стойността от регистровия

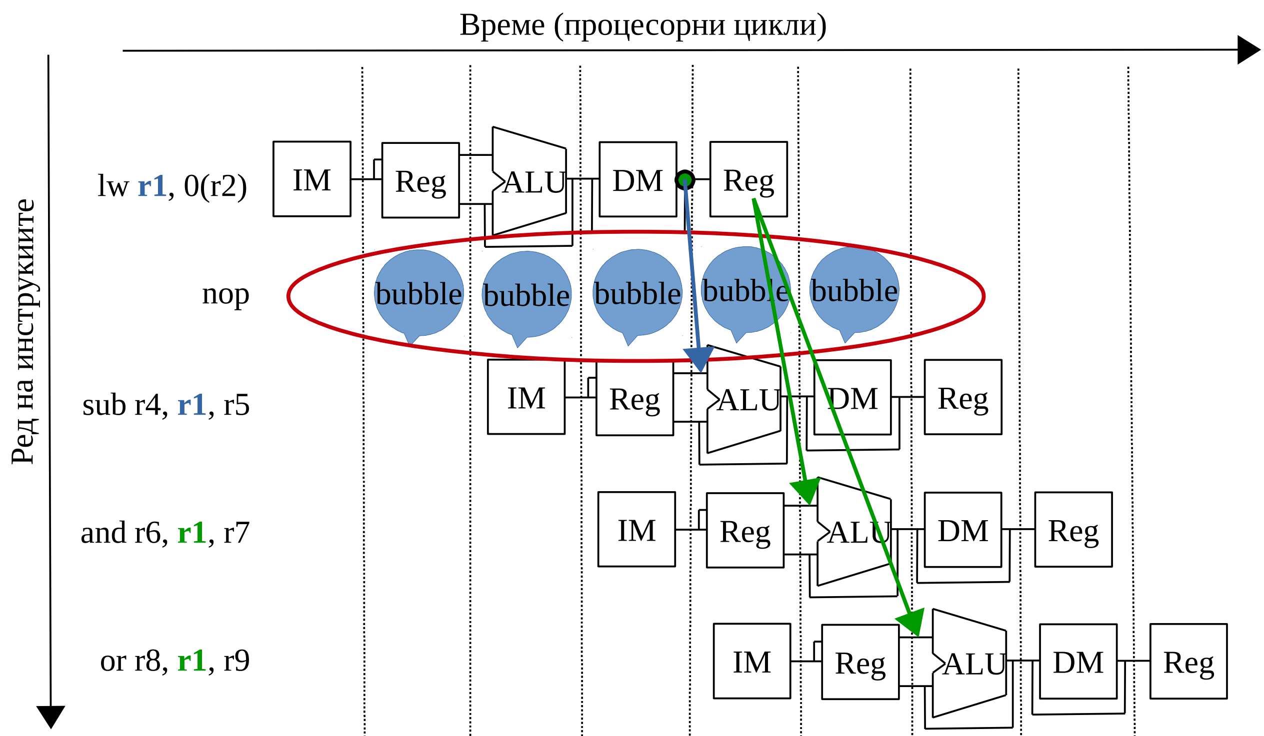
файл. За инструкцията *sub*, пренасочения резултат пристига твърде късно – в края на процесорния цикъл, а е бил необходим в началото.



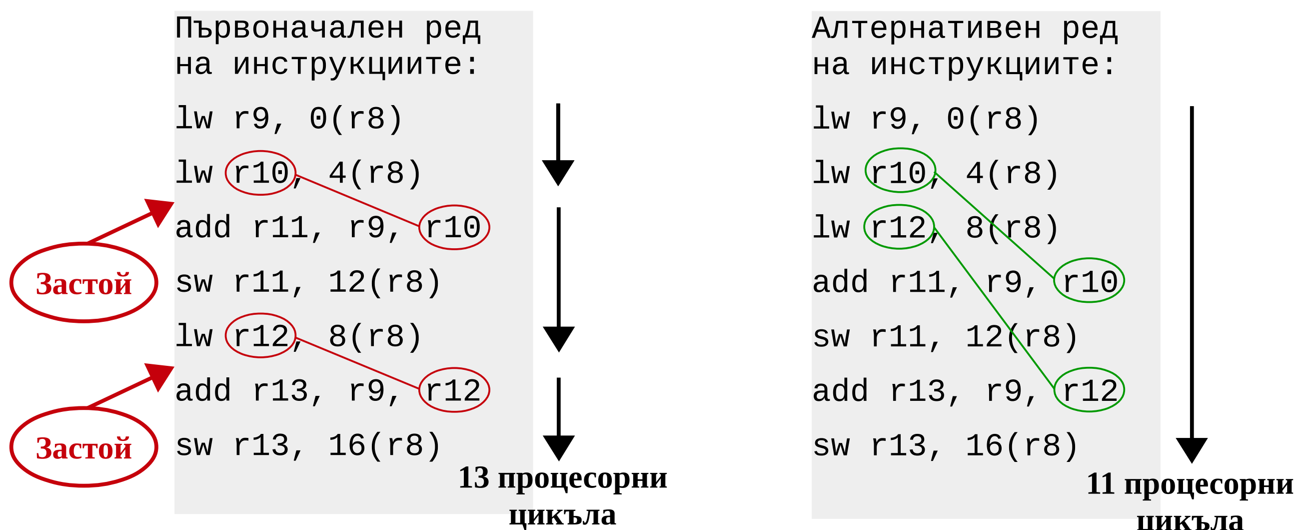
Инструкциите за зареждане на данни от паметта имат времезакъснение или латентност, което не може да бъде елиминирана само с пренасочване. Необходимо е добавяне на хардуер (схемна логика), наречена „pipeline interlock” (конвейерно заключване/блокиране), за да се изпълни коректно заложената последователност в програмата. Конвейерното заключване открива зависимости по данни и внася застои в изпълнението на поредната инструкция, докато състезанието се прекрати. Принципът е същият както при застоите в структурните състезания. На следващата фигура е показано конвейерното изпълнение на инструкциите с внасяне на застой при хардуерно заключване. Тъй като застоят забавя изпълнението на *sub* и следващите след нея инструкции с един процесорен цикъл, то отпада необходимостта от пренасочване за инструкцията *and*, а пренасочване за *or* не е необходимо дори и без внасяне на застой.



Застоят при хардуерно заключване е еквивалентен на застоя при включване на празната инструкция *nop* в кода на програмата както е показано на следващата фигура.



Хардуерното отлагане на изпълнението на инструкциите след инструкция за зареждане *lw* с един процесорен цикъл, ако следващата инструкция използва резултата от lw, или еквивалентното вмъкване на инструкция *nop* в кода на асемблер, води до загуба на производителност. Вместо да се изчаква по този начин и така да се губи един процесорен цикъл, компилаторът или програмата за асемблиране могат да вземат несвързана инструкция от друго място от кода и да я поставят веднага след инструкцията за зареждане от паметта. Това пренареждане на кода позволява да се избегне използването на резултата от *lw* в следващата я непосредствено инструкция. Пример за такова пренареждане на кода е представен със следващата програма на асемблер, която реализира следните операции: D=A+B; E=A+C.



В следващата лекция ще научите за последния вид състезания между инструкциите в конвейера – процедурните зависимости и какви са начините за тяхното преодоляване.