Паралелизъм на ниво инструкции

Съдържание:

1. Зависимости между инструкциите;
2. Производителност на процесора при конвейерно изпълнение;
3. Планиране на изпълнението на инструкциите;
4. Изпълнение с пренареждане на инструкциите. Реализация.

За да се подобри производителността на процесорите, след 1985 г., се използва конвейерно изпълнение на инструкциите. Застъпеното конвейерно изпълнение на инструкциите се нарича паралелизъм на ниво инструкции (на англ. instruction-level parallelism или ILP). Техниките, чрез които то се осъществява се разделят на две основни групи:

1. *динамично* откриване и използване на паралелизма между инструкциите в програмата, чрез *хардуера*, по време на нейното изпълнение;
2. *статично* откриване и използване на паралелизма, чрез *софтуера*, по време на *компилацията*.

1. Зависимости между инструкциите

Операндите (данните) на една инструкция в кода на програмата, често са операнди и на други инструкции в същата програма. Обикновено инструкциите с общи операнди са съседни. Еднаквите операнди или общите данни на две или повече съседни инструкции се наричат *зависимости* между инструкциите. При застъпено (конвейерно) изпълнение на инструкциите, зависимостите по данни водят до възникване на състезания. Състезанията са ситуации при които, заложения в програмата ред за изпълнение на операциите върху данните се променя. Това води до използване, от страна на инструкциите, на данни, които на се коректни и до сгрешен краен резултат.

Зависимостите са свойства на програмата, а състезанията – на конвейера. Наличието на зависимости в програмата показва потенциал за възникване на състезания в конвейера. Хардуерната схема на конвейера трябва да е проектирана така, че да открива възникващите състезания и да задържа изпълнението, толкова процесорни цикъла, колкото са необходими, докато състезанието отпадне. Застъпеното (паралелно) изпълнение на две инструкции е възможно само, ако между тях няма зависимости.

Една инструкция ***j*** е зависима от инструкцията ***i***, ако полученият от инструкцията ***i*** резултат се използва от инструкцията ***j*** или, ако инструкцията ***j*** е зависима от ***k*** и инструкцията ***k*** е зависима от инструкцията ***i***.

Зависимости по данни:

```
LD   $8,0($9)
ADD  $12,$8,$10
SD   $12,0($9)
```

В показания пример има зависимости между инструкциите ***LD*** и ***ADD***, както и между ***ADD*** и ***SD***. И в двата случая първата инструкция променя регистъра, който е входен операнд за следващата. Такива зависимости се наричат ***зависимости по данни*** и водят до възникване на състезания от вида ***RAW*** в конвейера.

Анти-зависимост се нарича зависимостта, при която една инструкция записва в регистър (или клетка от паметта), от който (която) предходната инструкция чете. Както е показано в следващия пример.

Анти-зависимост:

```
LW   $9, 100($10)
ADD  $10, $11, $12
```

Наличието на анти-зависимости в кода на програмата може да доведе до възникне на състезания от вида ***WAR*** при паралелното изпълнение на инструкциите в конвейера.

Зависимости по изход са тези, при които две инструкции записват в един и същ регистър или една и съща клетка от паметта. Потенциално те могат да доведат до състезания в конвейера от вида ***WAW***. По своята същност това са структурни състезания – едновременно

използване на един и същ ресурс. Както е в примера по-долу – две инструкции записват в един и същи регистър ***$9***.

Зависимост по изход:

```
LW   $9, 100($10)
ADD  $10, $9, $10
ADD  $9, $11, $12
```

2. Производителност на процесора при конвейерно изпълнение

Вече знаем, че за да се сравни производителността на два процесора (или на две архитектури), времето, необходимо за изпълнение на една програма от единия процесор (архитектура) се разделя на времето, необходимо за изпълнение на същата програма от другия процесор (архитектура).

Времето за изпълнение от процесора се дава със следното уравнение:

$$CPU_{time} = T_C \cdot CPI \cdot IC \quad ,$$

където *CPI* е *средният брой цикли необходими за изпълнението на една инструкция*, *IC* е *общият брой на инструкциите в програмата*, а T_C е *продължителността на процесорния цикъл* в секунди.

Тъй като програмата е една и съща, то общият брой инструкции *IC* също ще бъде еднакъв за двата сравнявани процесора. Ако и периодът на процесорния цикъл T_C е еднакъв (което е така, ако се сравняват две архитектури – без и с направени подобрения – на един и същи процесор), то за да се сравни кой процесор е по производителен е достатъчно да се определи *CPI* – средният брой цикли необходими за изпълнението на една инструкция.

Стойността на *CPI* за процесор с конвейер е:

$$CPI_{Pipelining} = 1 + S_{StucturalHazard} + S_{DataHazard} + S_{ControlHazard} \quad ,$$

където $S_{Structual}$ е броят празни процесорни цикли (застоите), необходими за преодоляване на структурните състезания (*WAW*), $S_{DataHazard}$ е броят празни процесорни цикли (застоите), необходими за преодоляване на състезанията от вида *RAW* и и $S_{ControlHazard}$ е броят празни процесорни цикли (застоите), необходими за преодоляване на състезанията свързани с инструкциите за преход.

Ако няма застои в конвейера (идеален конвейер), то след всеки процесорен цикъл от конвейера ще излиза по една инструкция. В реален конвейер, обаче са необходими застои за преодоляване на състезанията между инструкциите, което обяснява как е получено горното уравнение. За да се намали броят на процесорните цикли за изпълнение на една инструкция в конвейера е необходимо да се намалят стойностите на членовете в дясната част на равенството. За тази цел се използват различни техники, които попадат в една от двете основни групи представени по-горе. Част от тях, като пренасочване (намалява или избягва застоите при състезания от вида RAW) и предсказване на преходи (намалява или избягва застоите, причинени от инструкции за преход), вече разгледахме. Други, като планиране на изпълнението на инструкциите (статично от компилатора и динамично от хардуера) предстои да обсъдим.

3. Планиране на изпълнението на инструкции

Планирането на изпълнението на инструкции е подход за повишаване на производителността на конвейера, който се възползва от паралелизъма между инструкциите в една програма.

Броят на инструкциите, които могат да се изпълняват паралелно (да се застъпват) в конвейера е ограничен от инструкциите за преход. Правени за изследвания, които показват, че делът на инструкциите за преход в програмите за RISC процесор, е между 15 и 25 процента. Това означава, че средно в кода на програмата, преди и след всяка последователност от три до шест инструкции, има инструкция за преход. Като добавим и това, че между тези три до шест инструкции може да има зависимости по данни или структурни зависимости, то броя на последователно разположените инструкциите, които могат да се изпълняват застъпено в конвейера без да се внасят застои рязко намалява. Това налага да се разработят и реализират техники за пренареждане (планиране) на инструкциите в кода на програмата с цел

уплътняването им в конвейера, като се избягват застоите.

3.1. Статично планиране на инструкциите (пренареждане от компилатора преди изпълнението на програмата)

Нека разгледаме следния програмен цикъл:

```
for (i=999; i>=0; i=i+1)
      x[i] = x[i] + 10;
```

Във всяка итерация на цикъла има много малко паралелизъм, но между отделните итерации има – всяка итерация е независима от другата.

Нека да разгледаме кода на асемблер, чрез който програмният цикъл е представен с MIPS инструкции.

```
Loop:  ld     $8,0($9)      //регистърът $8 е текущият елемент от масива x[i]
       addi   $12,$8,10     //добавя 10 към текущия елемент
       sd     $12,0($9)     //съхранява резултата
       addi   $9,$9,4       //увеличава указателя с 4 байта (32 бита)
       bne    $9,$10,Loop   //преход за следващата итерация, ако $9≠$10
```

Без планиране, ако вземем само забавянето заради зависимостите по данни (в различен цвят), една итерация ще се изпълни за 9 процесорни цикъла:

```
Loop:  ld     $8,0($9)      //процесорен цикъл 1
       nop                  //процесорен цикъл 2  застой
       addi   $12,$8,10     //процесорен цикъл 3
       nop                  //процесорен цикъл 4  застой
       nop                  //процесорен цикъл 5  застой
       sd     $12,0($9)     //процесорен цикъл 6
       addi   $9,$9,4       //процесорен цикъл 7
       nop                  //процесорен цикъл 8  застой
       bne    $9,$10,Loop   //процесорен цикъл 9
```

Оказва се, че горният програмен цикъл ще се изпълни в процесор без конвейер само за 5 процесорни цикъла, колкото е броят на инструкциите в първоначалния код, а при изпълнението му в процесор с конвейер ще има забавяне с 4 процесорни цикъла.

Ако има планиране, то компилаторът може да пренареди инструкциите, като премести ***addi $9,$9,4***, както е показано по долу:

```
Loop:  ld     $8,0($9)      //процесорен цикъл 1
       addi   $9,$9,4       //процесорен цикъл 2
       addi   $12,$8,10     //процесорен цикъл 3
       nop                  //процесорен цикъл 4  застой
       nop                  //процесорен цикъл 5  застой
       sd     $12,-4($9)    //процесорен цикъл 6
       bne    $9,$10,Loop   //процесорен цикъл 7
```

Преместването на ***addi*** налага промяна на отместването в ***sd***. С пренареждането броят на процесорните цикли за една итерация се намалява, но отново са повече от 5, т.е. вместо конвейерът да ускори изпълнението на програмния цикъл – го забавя.

Един прост начин, който позволява да се избегне това забавяне е да се увеличи броят на инструкциите, които участват в итерацията, т.е. броя на инструкциите между две инструкции за преход. Тази техника на англ. се нарича „loop unrolling“ или „разгъване на цикъла“. При „разгъването“, просто, тялото на цикъла се копира няколко пъти преди инструкцията за преход, както е показано по долу:

```
Loop:  ld     $8,0($9)      }
       addi   $12,$8,10     } Тяло на цикъла
       sd     $12,0($9)     }
       ld     $14,4($9)     }
       addi   $16,$14,10    } Тялото на цикъла се повтаря
       sd     $16,4($9)     }
       ld     $18,8($9)     }
       addi   $20,$18,10    } Тялото на цикъла се повтаря
       sd     $20,8($9)     }
       ld     $22,12($9)    }
       addi   $24,$22,10    } Тялото на цикъла се повтаря
       sd     $24,12($9)    }
       addi   $9,$9,16      } Индексът нараства и се
       bne    $9,$10,Loop   } проверява условието
```

Този „разгънат цикъл“ ще се изпълни за 27 процесорни цикъла – инструкциите в кода са 14; всяка инструкция **`ld`** води до застой от 1 процесорен цикъл; между инструкциите **`addi`** и **`sd`** трябва да има забавяне от два процесорни цикъла; преди инструкцията ***`bne`*** също трябва да има застой от 1 процесорен цикъл; или ***14 + 4*(2 + 1) + 1 = 27*** процесорни цикъла. Разделяйки на 4 (тялото на програмния цикъл се изпълнява 4 пъти) се получава ***6,75*** процесорни цикъла за една итерация.

Сега нека да пренаредим инструкциите:

```
Loop: ld     $8,0($9)
      ld     $14,4($9)
      ld     $18,8($9)
      ld     $22,12($9)
      addi   $12,$8,10
      addi   $16,$14,10
      addi   $20,$18,10
      addi   $24,$22,10
      sd     $12,0($9)
      sd     $16,4($9)
      addi   $9,$9,16
      sd     $20,8($9)
      sd     $24,12($9)
      bne    $9,$10,Loop
```

След четирикратното „разгъване“ на програмният цикъл се породи достатъчен паралелизъм между инструкциите, така че кодът да може да се планира без вмъкване на застои. Времето за изпълнение намалява до ***14*** процесорни цикъла, колкото са инструкциите в кода, което прави ***3,5*** процесорни цикъла за една итерация (***14/4 = 3,5***). Ако направим сравнение с оригиналния код без планиране на инструкциите и без „разгъване на цикъла“, който, без конвейер се изпълнява за ***5*** процесорни цикъла, то тук времето за изпълнение намалява ***1,43*** пъти (***5/3,5 = 1,43***).

Увеличаването на броя на повторенията на инструкциите в тялото на цикъла, например ***8*** повторения вместо ***4*** ще доведе до още по-голямо намаляване на времето за изпълнение. Обаче, „разгъването на цикъла“ не може да бъде много голямо поради следните ограничения:

- увеличаваният размер на кода;
- потенциалният недостиг на регистри.

3.2. *Динамично планиране на инструкциите (пренареждане от хардуера по време на изпълнението на програмата)*

Конвейерите без динамично планиране извличат и изпълняват инструкциите в реда, в който те се появяват в програмния код. Ако изпълнението на дадена инструкция трябва да бъде задържано, поради структурна зависимост или зависимост от предходни инструкции, то извличането и изпълнението на следващите инструкции, също не може да продължи. Нови инструкции не се извличат и не се пускат за изпълнение, докато възникналото състезание не бъде преодоляно. За да се преодолее тази загуба на производителност, в компилатора се прави опит да се планират инструкциите, като се пренаредят, така че да се избегнат зависимостите; този подход, както видяхме, се нарича статично планиране.

Динамичното планиране е друг подход, чрез който се намалява загубата на производителност в конвейера. При него хардуерът пренарежда изпълнението на инструкциите, за да се намалят застоите.

Основната идея при динамичното планиране е да се даде възможност на инструкциите, които се намират след задържаната инструкция в конвейера и са независими от възникналото състезание да продължат да се изпълняват.

Структурните зависимости и зависимостите по данни се проверяват по време на декодиране на инструкцията (на етапа *ID*): когато инструкцията не е зависима от данните на предходните инструкции в конвейера или необходимите функционални елементи за нейното изпълнение са свободни, то изпълнението ѝ може да продължи и след етапа *ID*. За да може

изпълнението на инструкцията да продължи веднага щом операндите ѝ са налице, дори ако изпълнението на предходната инструкция е задържано, етапът *ID* се разделя на два етапа:

1. *Допускане* – инструкцията се декодира, прави се проверка за структурни зависимости;
2. *Четене на операндите* – ако има зависимости по данни се изчаква докато отпаднат, след което се прочитат операндите.

Етапът *IF* предшества етапа „*допускане*“, а етапът *EXE* следва етапа „*четене на операндите*“.

Трябва да се има предвид, че инструкциите, които имат операнди с плаваща запетая отнемат няколко процесорни цикъла при извършване на операцията на етапа *EXE*. Различните операции за числа с плаваща запетая отнемат различен брой процесорни цикъла. Ето защо, за да се изпълняват няколко инструкции едновременно, се налага да се отчита кога започва и кога завършва изпълнението на всяка една инструкция.

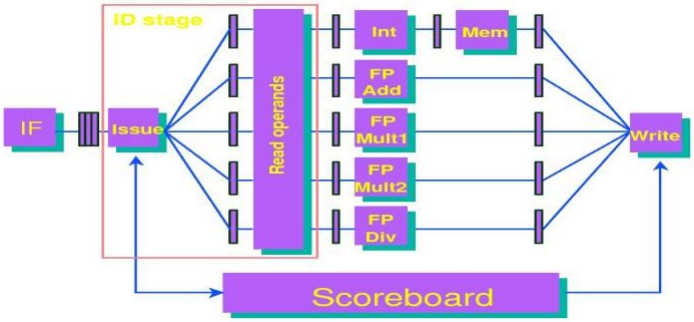
4. Изпълнение с пренареждане на инструкциите. Реализация

4.1. Централизирано управление на инструкциите – Scoreboard

Една от първите хардуерни реализации на динамично планиране на изпълнението на инструкциите е осъществена в суперкомпютъра CDC 6600. Тя е наречена „*Scoreboard*“. Целта е средният брой процесорни цикли, необходими за изпълнението на една инструкция в конвейера да се поддържа близо до единица, *CPI = 1*. Когато поредната инструкция от кода трябва да се задържи, тъй като зависи от данните на предходните инструкции, които все още са в конвейера, се търси друга инструкция, която да не зависи от нито една задържана или активна инструкция, за да бъде допусната (при липса на структурни зависимости, т.е. има достатъчно ресурси) и изпълнена.

По същество s*coreboard* е централизирана хардуерна схема, която открива зависимостите и управлява изпълнението на инструкциите, като записва и проследява:

- инструкциите, които са в процес на изпълнение;
- множеството функционални единици, в който се извършва изпълнението на няколко инструкции едновременно;
- регистрите, в които функционалните единици ще запишат резултатите.



На фигурата е показан конвейер с централизирано управление на инструкциите (чрез управляващата схема – s*coreboard)* всяка извлечена инструкция, първо се допуска като се декодира и проверява за структурни зависимости (WAW). Изгражда се запис на зависимостите по данни. След това се определя кога могат да се прочетат операндите на инструкцията и да започне изпълнението ѝ. Ако инструкцията не може да се изпълни незабавно, управляващата схема следи за всяка промяна в състоянието на хардуерните елементи и решава кога да започне нейното изпълнение. Освен това s*coreboard* контролира кога резултатът от дадена инструкция може да се запише в съответния целеви регистър. По този начин, откриването и разрешаването на възникналите зависимости между инструкциите се извършва централизирано в схемата за управление s*coreboard*.

Както се вижда от фигурата в етапа *Issue* (*Допускане*) инструкциите попадат в обичайният си ред (така както са подредени в кода на програмата), след което се

разпаралелват. В следващия етап *Read operands* (*Четене на операнди*) всяка една от тях може да бъде задържана или пропусната, което води до пренареждане на инструкциите при тяхното изпълнение.

Пренареждането на инструкциите води до възникване на състезания от вида *WAR* и *WAW* каквито не могат да се възникнат в конвейера без динамично планиране. За преодоляване на тези зависимости се прилагат следните решения:

- Преодоляване на състезанията от вида *WAR* – В етапа *Write* (виж фигурата по-горе) инструкцията се задържа, докато инструкциите, предшестващи задържаната (в оригиналния код на програмата), за които е възникнало състезанието от вида *WAR*, прочетат своите операнди.
- Преодоляване на състезанията от вида *WAW* – Инструкцията се задържа в етапа *Issue*, докато другите инструкции завършат изпълнението си.
- Преодоляване на състезанията от вида *RAW* – В етапа *Read operands*, динамично се разрешават зависимостите по данни, след което инструкциите могат да бъдат изпратени за изпълнение в различен от оригиналния програмен ред.

За реализиране на посочените решения в *scoreboard* се записват и проследяват зависимостите и състоянието на операциите.

Конвейерът с централизирано управление на инструкциите (фигурата по-горе) има четири етапа, които заменят етапите *ID*, *EXE* и *WB* от стандартният конвейер (разгледан в предишните лекции). Четирите етапа са:

- ***Issue*** – Ако функционалното устройство, необходимо за изпълнение на инструкцията е свободно и няма други активни инструкции, използващи същия целеви регистър, scoreboard допуска (*issues*) инструкцията до функционалното устройство и отбелязва новото състояние във вътрешната си структура от данни. Този етап заменя част от етапа *ID* в стандартния конвейер. Проверката дали има друго активно функционално устройство, което да записва резултата от операцията в същия целеви регистър гарантира, че няма да възникне състезание от тип *WAW*. Ако съществува структурна зависимост (водеща до състезание от вида *WAW*), то допускането на инструкцията се задържа, а с това и следващите след нея, докато състезанието не отпадне. Това може да доведе до запълване с инструкции на буфера (опашката) между етапите „извличане на инструкция“ и „допускане“ (*Issue*). Ако буферът се напълни, извличането на инструкции се прекратява.

- ***Read operands*** – „Scoreboard“ следи дали изходните операнди на текущо допусната инструкция са налични, т.е. дали няма активни, допуснати по-рано инструкции, които записват в регистрите, които са входни операнди на текущо допуснатата инструкция. Ако входните операнди са налични „*scoreboard*“ сигнализира на функционалните устройства да извършат четеното на операндите от регистрите и да започнат изпълнението. По този начин, на този етап, „*scoreboard*“ разрешава динамично възникналите състезания от вида *RAW*, което позволява инструкциите да се изпращат за изпълнение в различен ред спрямо оригиналната програма. Този етап заедно с етапът „*issue*“, изпълняват функциите на етапа *ID* в стандартния конвейер.

- ***Execution*** – Функционалното устройство започва изпълнението след получаване на операндите. След завършване на операцията и получаване на резултата се сигнализиран на „*scoreboard*“, че изпълнението е приключило. Този етап заменя етапа *EXE* в стандартния конвейер. На този етап операциите с плаваща запетая отнемат не един, а множество процесорни цикъла.

- ***Write result*** – След като е сигнализирано, че изпълнението във функционалното устройство е завършило, „*scoreboard*“ проверява за състезания от вида *WAR* и задържа изпълнението на инструкцията, ако е необходимо.

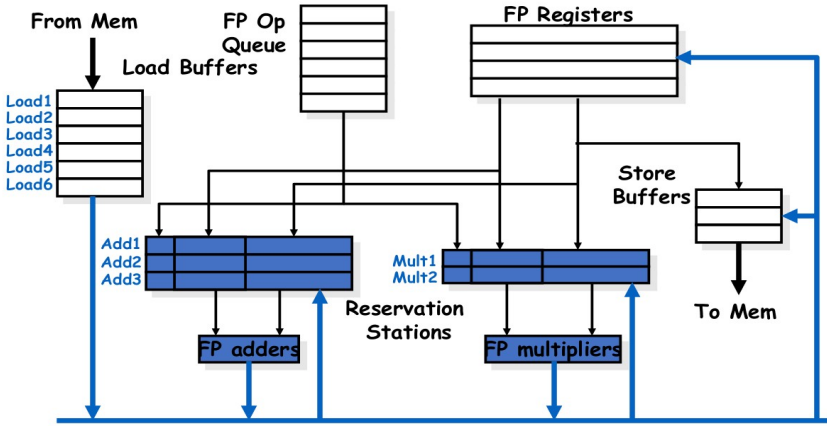
4.2. Управление на инструкциите с резервационни станции (принцип на Томасуло)

Друга хардуерна реализация за динамично планиране е тази на алгоритъма на Томасуло. Той е разработен от Робърт Томасуло през 1967 г. Целта е била да се постигне висока производителност без да е необходимо да се използват специални компилатори. Първоначално е реализиран в модула за числа с плаваща запетая на IBM System/360 Model 91. В последствие е внедрен в разработките на процесори като: Alpha 21264, UltraSparc, HP 8000, MIPS 10000, Pentium II, PowerPC 604 и други.

Предпоставка за разработката на този алгоритъм са и разликите между архитектурите на системите инструкции на IBM 360 и CDC 6600. Тези разлики са:

- инструкциите на IBM имат два регистрови операнда срещу 3 в CDC 6600;
- в архитектурата на IBM има 4 регистъра а работа с числа с плаваща запетая (FP) срещу 8 в CDC 6600;
- в системата инструкции на IBM има операции от вида памет-регистър, т.е. една част от операндите на инструкцията се намират в регистри, а друга – в паметта.

На следващата фигура е представена структурата на модул за извършване на операции с числа с плаваща запетая (FP – Floating Point), реализиращ алгоритъма на Томасуло.



Инструкциите за работа с числа с плаваща запетая се подреждат в опашката „*FP Op Queue*“ в реда на тяхното извличане. По същия начин, както това се прави и в „scoreboard“. За разлика от „scoreboard“, обаче, където всичко се извършва централизирано, тук управлението и буферирането се ***разпределят*** между функционалните устройства (суматори и умножители за числа с плаваща запетая – *FP adders* и *FP multipliers* – в син цвят). Буферите (*Add1..3* и *Mult1..2*) асоциирани с функционалните устройства се наричат „***резервационни станции***“ (Reservation Stations – RS). В тях се буферират операндите на инструкциите, които чакат да бъдат допуснати до изпълнение в съответното функционално устройство. Основната идея е че „резервационната станция“ извлича и буферира операнда тогава, когато той стане наличен. Така се елиминира нуждата операндът да бъде четен от регистъра (един от регистрите в регистровия файл – *FP Registers*), който е посочен в самата инструкция.

Резервационните станции (RS), които ще осигурят входните данни в допуснатите и чакащи за изпълнение инструкции, заменят първоначално посочените, в инструкцията, регистри. Това се извършва като стойностите в кода на инструкцията, които определят кои са входните регистри, се заменят със стойности или указатели, сочещи към резервационните станции. Този процес се нарича „***преименуване на регистри***“. Преименуването позволява да се избягват състезанията от вида ***WAR*** и ***WAW***.

Резервационните станции са повече от регистрите, което дава възможност да се елиминират състезания, които възникват от анти-зависимости и зависимости по изход, т.е. зависимости, които не могат да се открият и елиминират от компилатора.

Използването на резервационни станции, вместо единствен регистров файл, позволява: 1) откриването на състезания и управлението на изпълнението се извършва

разпределено: информацията, съхранявана в резервационните станции към всеки функционален блок, определя кога дадена инструкция може да започне своето изпълнение на това устройство; 2) резултатите от изпълнението на инструкциите, се буферират в резервационните станции (вместо да преминават през регистрите), откъдето се предават директно на функционални устройства. Общата шина за данни (в син цвят на фигурата), предава резултата към всички резервационни станции и свързаните с тях функционални устройства едновременно.

Буферите за зареждане (Load Buffer) и за съхранение (Store Buffer) съдържат данни или адреси, които *се четат от* или *се записват в* паметта. Те работят по същия начин както резервационна станция с функционални устройства.

Всяка резервационна станция има следните полета:

- *Op* – Операцията, която ще се изпълни във функционалното устройство върху входните операнди *S1* и *S2*.
- *Vj, Vk* – Стойностите на входните операнди.
- *Qj, Qk* – Резервационните станции, които ще предоставят съответните входни операнди; стойност нула показва, че входният операнд вече е наличен във *Vj* или *Vk* или е ненужен.
- *Busy* – Показва, че тази резервационна станция и придружаващата я функционална единица са заети (изпълнява се инструкция).

За да се укаже кое функционално устройство в кой регистър от регистровия файл ще записва получения резултат се поддържа структура, наречена „*Register result status*“. Номерата на позициите в тази структура отговарят на номерата на регистрите в регистровия файл. Ако към определен момент дадена позиция е празна (т.е. има стойност *0*), то това означава, че активната инструкция не изчислява резултат, предназначен за този регистър.

Алгоритъмът на Томасуло има три етапа през които преминават инструкциите. Тези етапи са:

1. ***Допускане (Issue)*** – Взема се поредната инструкция от опашката „*FP Op Queue*“. Ако има свободна резервационна станция (т.е. няма структурни състезания) инструкцията се допуска и операндите се изпращат (преименуване на регистрите) до регистрационната станция. Проверката дали има свободна регистрационна станция преди поредната инструкция от опашката да бъде допусната до станцията позволява да се избегнат състезанията от вида *WAR* и *WAW*.
2. ***Изпълнение (Execution)*** – Във функционалното устройство, асоциирано с резервационната станция, се извършва операцията, заложена в инструкцията, върху преименуваните операнди. Ако един или повече от операндите не са налични в резервационната станция, се изчаква те да бъдат изчислени и да се появят на *Common Data Bus* (шината в син цвят на горната фигура). Проверката за наличие на всички операнди преди да се извърши операцията позволява да се избегнат състезанията от вида *RAW*.
3. ***Запис на резултата (Write result)*** – Когато приключи изпълнението и резултатът от операцията е готов, той се предава на *Common Data Bus*, а от там се прехвърля и в регистрите, и във всички резервационни станции, които очакват този резултат, включително и в Store Buffers, от където се записват и в паметта, ако това е необходимо.

В една стандартна шина за данни има линии за предаване на данни и линии, по които се предава адресът на местоназначението на тези данни. В шината *Common Data Bus*, вместо адреса на местоназначението се предава адресът на функционалното устройство, което е източник на данните. Тази шина има 64 линии (бита) за данни и 4 линии (бита) за адреса на функционалното устройство. Данните се записват в приемащите устройства, след като се сравни дали идват от функционалното устройство от което се очаква да постъпят нови данни.