КАТЕДРА: КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ
ДИСЦИПЛИНА: КОМПЮТЪРНИ АРХИТЕКТУРИ

ЛАБОРАТОРНО УПРАЖНЕНИЕ № 4

ТЕМА: КОНВЕЙЕРНО ИЗПЪЛНЕНИЕ НА ИНСТРУКЦИИТЕ В ПРОЦЕСОРА

Цел: Да запознае студентите с начина на работа на конвейерите в процесора – какво се случва на всеки един от етапите и как се избягват състезания между инструкциите при тяхното конвейерно изпълнение.

I. Теоретична част:

В процесор без конвейер инструкциите се изпълняват последователно една след друга. За да бъде изпълнена една инструкция са необходими различни ресурси:

- памет;
- регистри;
- АЛУ (аритметични-логическо устройство).

Тези ресурси не се заемат през цялото време на изпълнение на инструкцията. Това позволява изпълнението да се раздели на етапи, така че във всеки етап да се използват различни ресурси. Докато една инструкция се намира в един етап на изпълнение и работи с даден ресурс, друга инструкция може да работи с друг ресурс в друг етап. Така, в даден момент във всеки етап може да се намира различна инструкция. В крайна сметка, вместо да се изчаква докато една инструкция премине през всички етапи, за да завърши своето изпълнение и след това да започне изпълнението на следващата, то изпълнението на няколко инструкции може да се застъпи, като броят на инструкциите, които могат се изпълняват едновременно е равен на броя на различните етапи.

Конвейерно изпълнение

Изпълнението на инструкциите се разделя в последователни зависими етапи. След като една инструкция завърши изпълнението в едни етап, тя може да премине в следващия, а следващата инструкция може да започне да се изпълнява в освободения предходен етап. По този начин броят на инструкциите, които ще се изпълнят за определен период от време се увеличава. Всички инструкции преминават през всички етапи, които имат фиксирано времетраене, така че времето за изпълнение на една инструкция не може да варира.

Етапи в конвейерното изпълнение за процесор със система инструкции MIPS

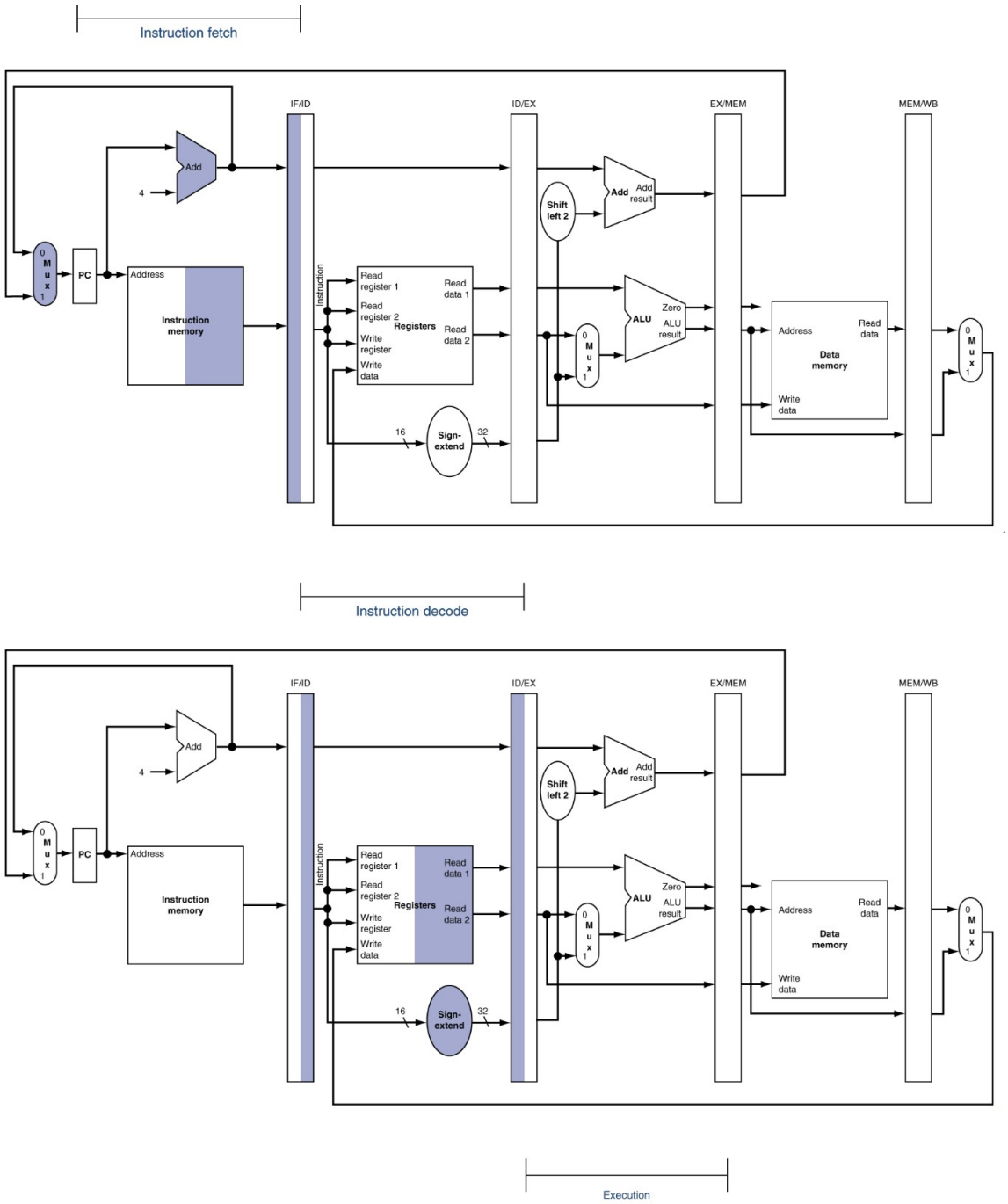
Етапите при конвейерно изпълнение в MIPS процесор са пет, като на всеки етап се изпълнява по една инструкция:

1. IF (Instruction fetch): Извличане на инструкцията от паметта;
2. ID (Instruction decode): Декодиране на инструкцията и четене от регистрите;
3. EX (Execute): Извършване на операцията или изчисляване на адрес;
4. MEM (Memory access): Достъп до операнд в паметта;
5. WB (Write back): Запис на резултата обратно в регистър.

На следващите фигури са представени в блоков вид изчислителните ресурси, които се използват на всеки един от етапите при изпълнението на инструкциите в MIPS процесор.

На етапа IF се прочита една инструкция от паметта за инструкции от адрес, чиято стойност е записана в регистъра PC (програмен брояч). Освен това PC се инкрементира с 4 (толкова байта е дължината на инструкциите в системата MIPS), за да сочи към адреса на следващата инструкция в програмата, която предстои да се изпълни.

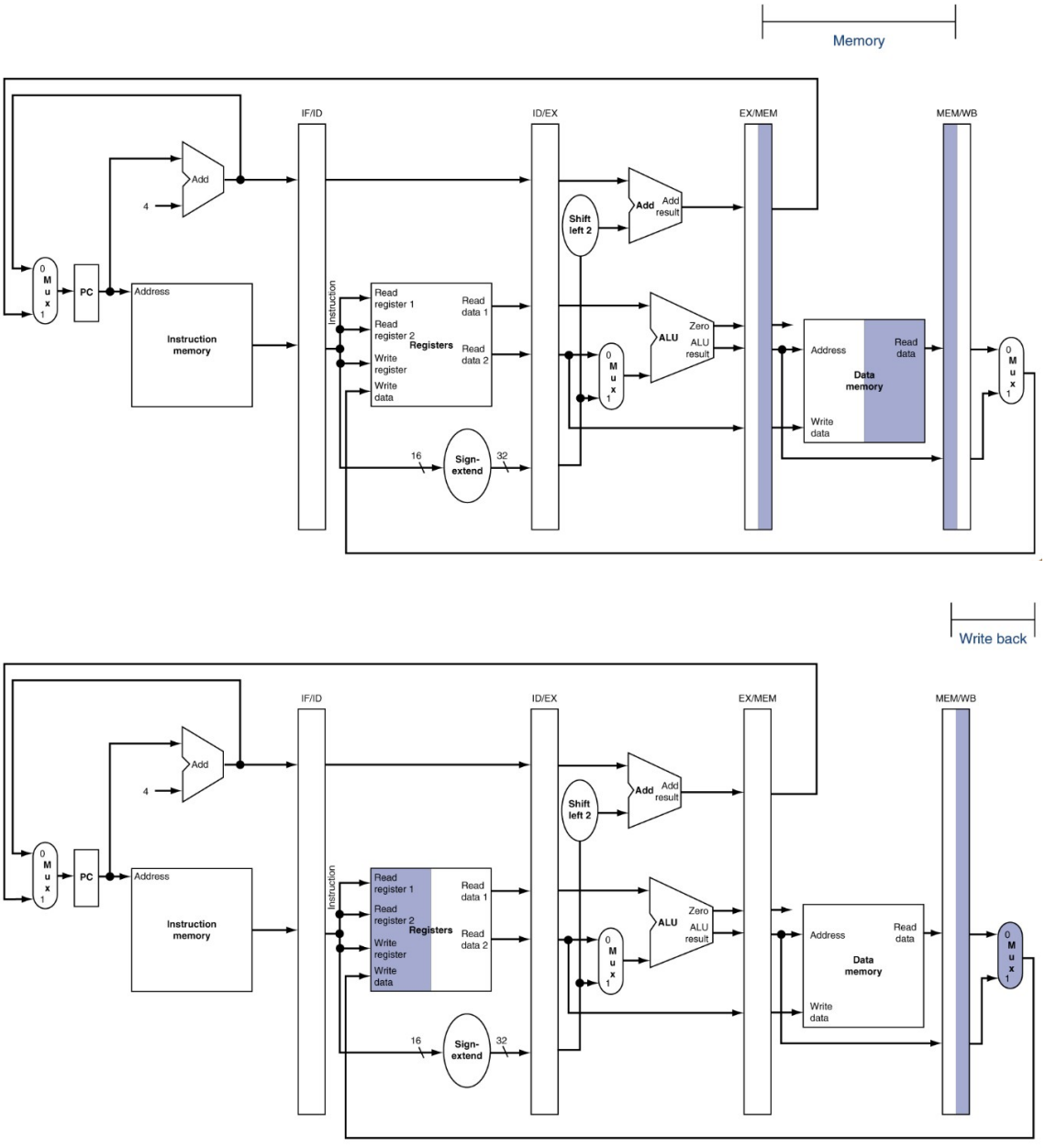
На етапа ID се декодира инструкцията, за да се определи операцията, която следва да се изпълни и се прочитат стойностите на нейните операнди, записани в регистрите с общо предназначение. Ако операнда е непосредствена стойност, зададена в самата инструкция, тя се преобразува от 16-битова в 32-битова, за да бъде с размерността на другите данни, участващи в операцията.


Instruction fetch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
16
Sign-extend
32
Data memory
Read data
Write data
Instruction decode
Execution

На етапа EXE се изпълнява операцията (аритметична или логическа), като операндите се зареждат в АЛУ и там се извършва действието. Ако операцията не е аритметична или логическа, а зареждане или съхраняване в паметта, на този етап в АЛУ се изчислява адресът, на който ще се осъществи достъпът до паметта. Ако изпълняваната инструкция е за условен преход (към определен адрес в програмата) в АЛУ се извършва сравнение на двата операнда, за да се провери условието на прехода. Допълнителният суматор в този етап (компонентът означен с **Add**) се използва, за да се добави отместването в инструкция за преход към адреса на текущо изпълняваната инструкция и така да се установи адресът, от който да продължи изпълнението на програмата.

Етапът MEM се използва само за инструкциите за зареждане или съхраняване в паметта. Операндите на аритметичните и логическите инструкции се прехвърлят без да се осъществява достъп до паметта. Етапът не се използва и при изпълнение на инструкции за преход.

Етапът WB се използва при зареждане на данни от паметта, за да могат прочетените от паметта данни да се запишат в регистър. Етапът се използва и за запис на резултата от аритметичните и логически операции обратно в регистър. Този етап не се използва при инструкции за преход или съхраняване в паметта.



Състезания между инструкциите при конвейерно изпълнение

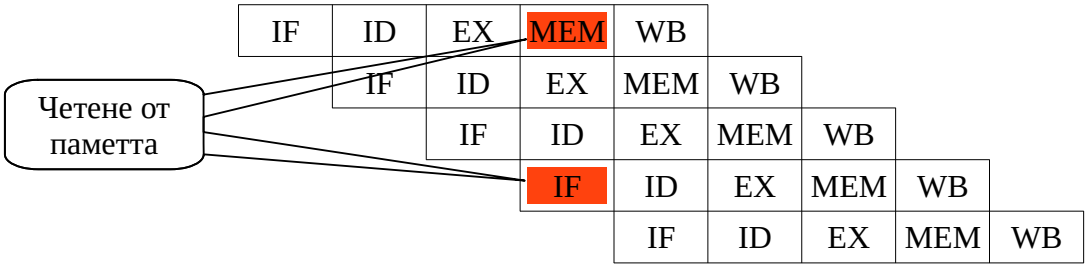
При застъпеното изпълнение могат да възникнат конфликти между инструкциите, намиращи се едновременно в конвейера. Тези конфликти са три вида и се наричат зависимости или състезания:

• Състезания за ресурси (структурни зависимости) – един ресурс се използва от две инструкции, изпълняващи се на различни етапи в конвейера.

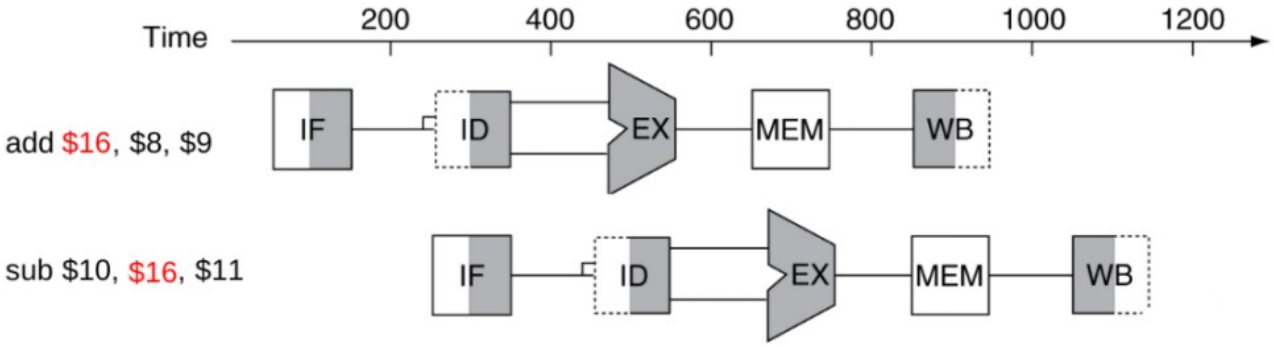
• Състезания за данни (зависимости по данни) – дължат се на зависимостта между някой инструкции по отношение на използваните операнди. Например, резултатът от една инструкция се използва като входен операнд в следващата.

• Състезания за управление (процедурни зависимости) – дължат се на инструкциите за преход, които управляват последователността на инструкциите в изпълнението на програмата.

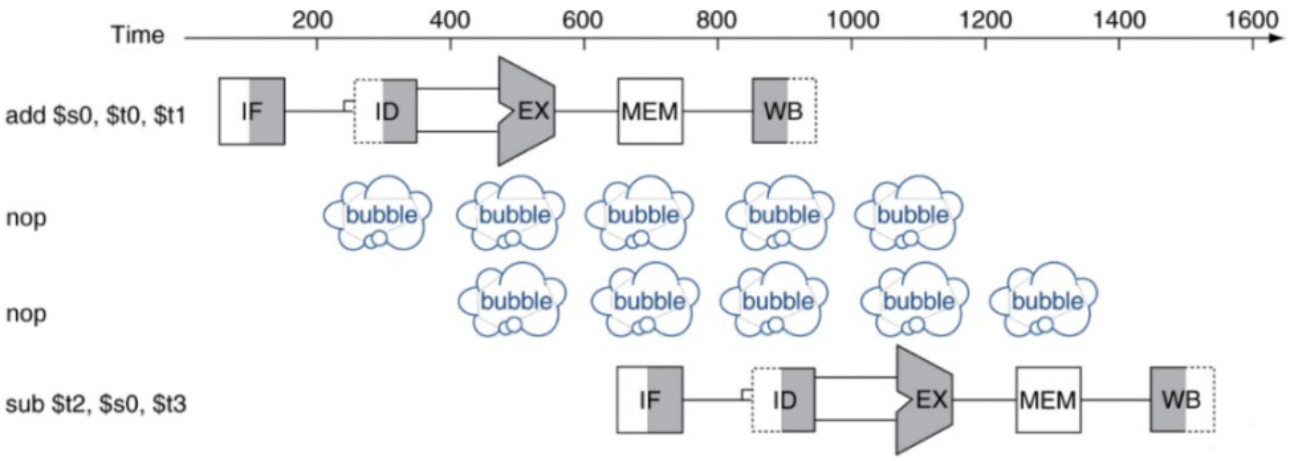
На фигурата е представен пример за структурна зависимост. В момента от време когато една инструкция осъществява достъп до паметта, за да прочете данни, друга инструкция бива извличана от паметта. Налице е едновременен достъп до паметта или още може да се каже, че се получава състезание за достъпа до паметта. Това състезание се преодолява, като се обособят два вида кеш памети – за инструкции и за данни.



На следващата фигура е представена зависимост между две инструкции по данни.

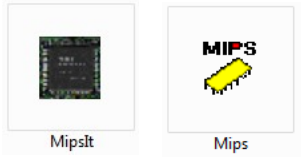


Инструкцията ADD сумира регистрите R8 и R9 и записва резултата в R16. Следващата инструкция SUB използва резултата от R16, за да извърши операцията: R10 = R16 – R11. За да бъде верен резултатът от втората инструкция е необходимо операцията изваждане да се изпълни след като резултатът от инструкцията ADD се запише в регистър R16. Ако инструкциите се изпълняват застъпено в конвейера на процесора се получава обратното: първо се изпълнява операцията изваждане за втората инструкция, а след това се извършава запис на резултата от инструкцията ADD в регистъра. Това е така, тъй като, във времето, етапът изпълнение (EXE) за инструкцията SUB е преди етапът запис в регистър (WB) за инструкцията ADD. За да се преодолее този конфликт при използването на регистъра R16 е необходимо втората инструкция да се забави, като изпълнението и започне два процесорни цикъла по-късно, както е показано на следващата фигура. За целта се добавят празни инструкции – NOP. Добавените NOP инструкции се наричат застои, на англ. „stall“ или „bubble“



II. Практическа част:

Инструментите, които се използват в практическата част са:

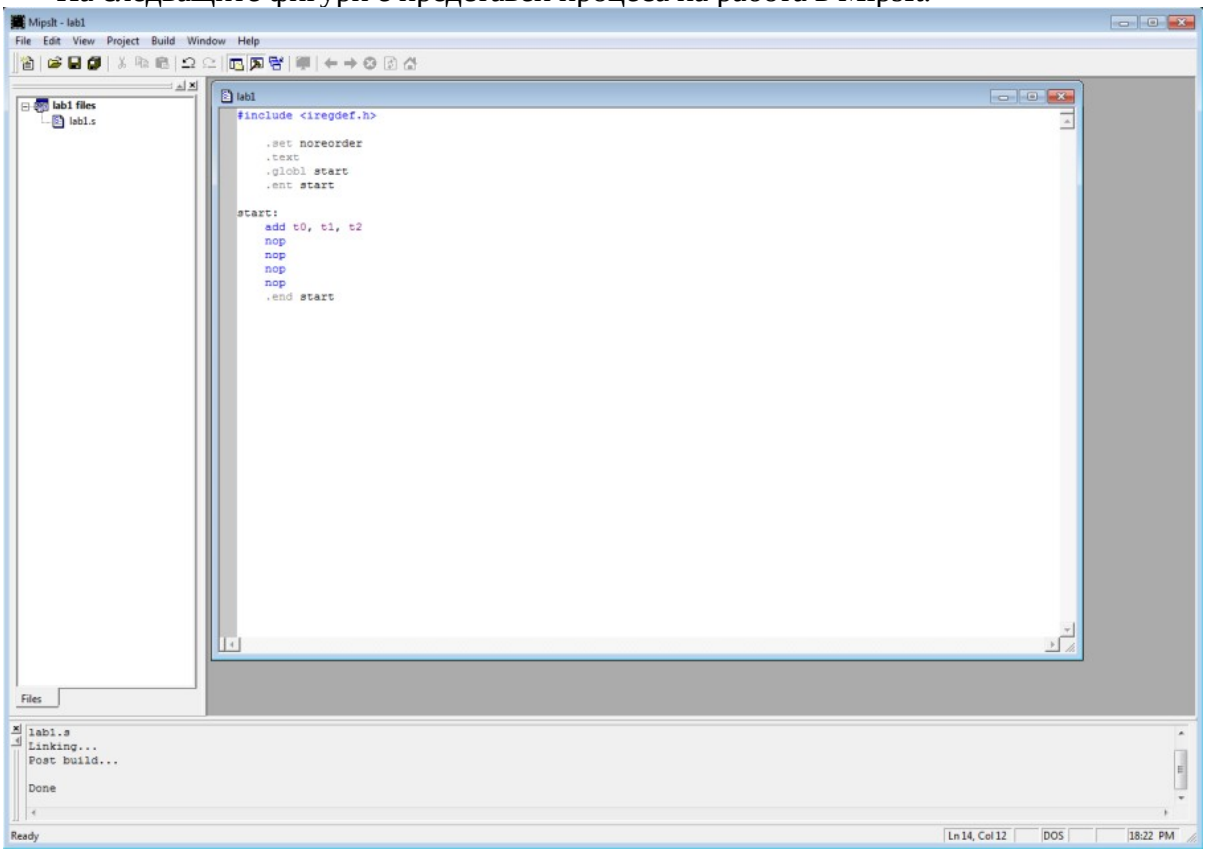


• **MipsIt.exe** – за редактиране и създаване на програми за MIPS процесор

• **MipsPipeS.exe** – за симулация на последващото им изпълнение в конвейера

MipsIt е интегрираната среда за разработка, която може да се изтегли от: https://www.eit.lth.se/fileadmin/eit/courses/eitf20/Labs2016/MipsICT.exe.zip. Трябва да се има предвид, че работи с *Windows 7* или по-ниска версия.

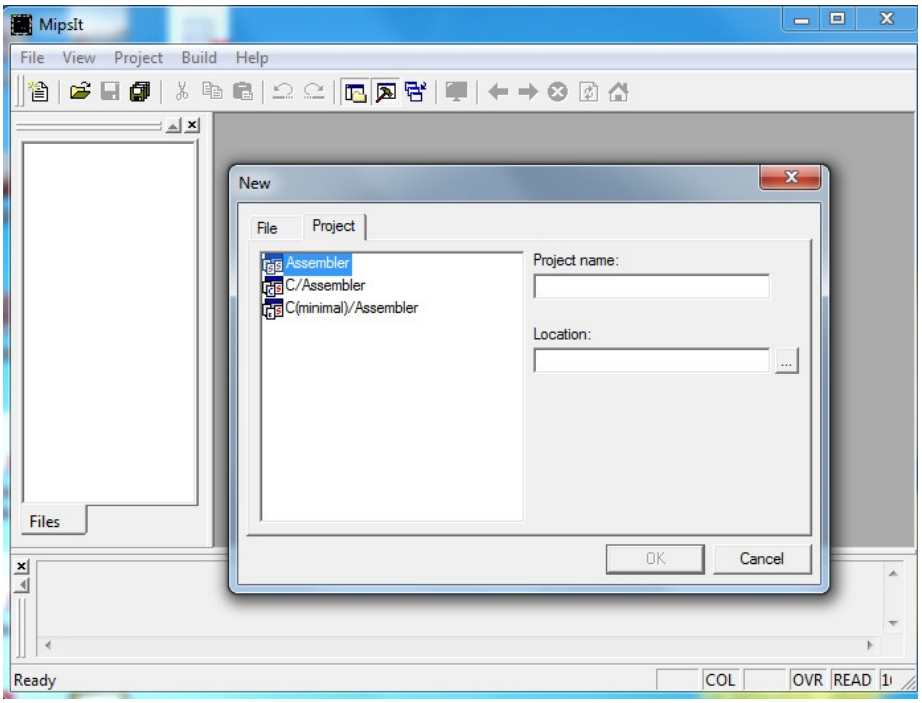
На следващите фигури е представен процеса на работа в MipsIt.



Файловете на проекта са представени в ляво. При двукратно щракване върху файл той се отваря в самостоятелен прозорец в дясно. Повечето команди имат клавишни комбинации, за да се направи работата по-ефективна. Също така има ленти с инструменти за по-бърз достъп. За да се конфигурира MipsIt се използва *File->Options...* . Може да се настрои COM

порта, компилатора, пътищата и други. При първоначално стартиране на MipsIt се извършва авто-конфигуриране с изключение на съобщението за невъзможност да се отвори COM порта. То се игнорира с натискане на OK.

Създаването на нов проект се извършва от *File -> New* и е представено на следващата фигура.



Има възможност да се избере между три вида проекти:

1. Assembler – Ако проекта ще съдържа само файлове на асемблер, както в това упражнение.
2. C/Assembler – Ако проекта ще съдържа само файлове на C или на C и асемблер.
3. C(minimal)/Assembler – Същото както предходното, но с използване на минимални библиотеки.

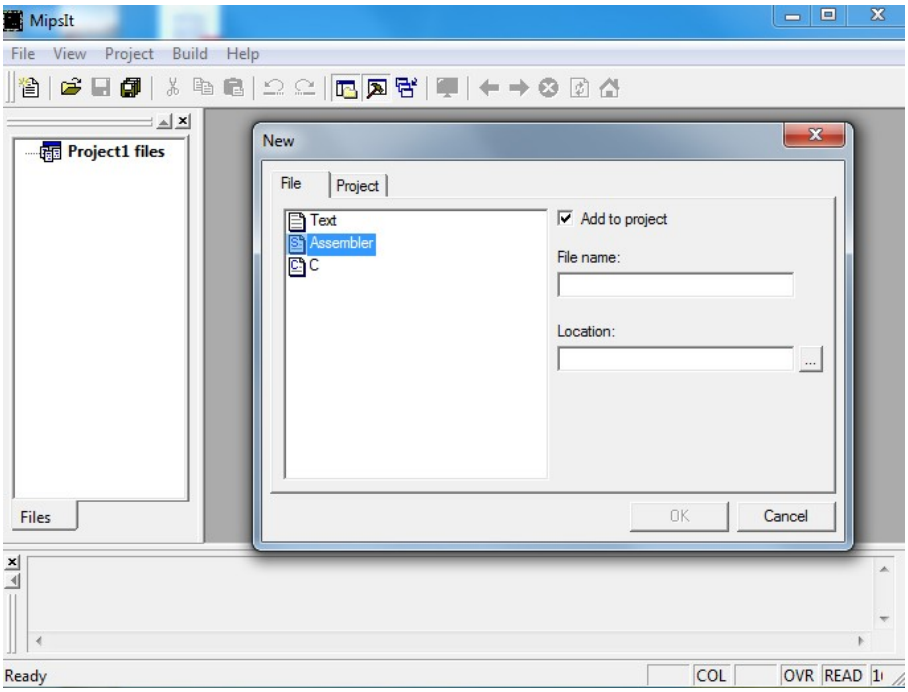
Въвежда се името на проекта и местоположението (по-желание) и се натиска OK.

Разликите между видовете проекти е в библиотеките и модулите, които ще бъдат включени. Проектите от вида C/Assembler ще свържат в изпълнима програма големи библиотеки и резултатът ще бъде по-голям изпълним файл. Това няма да работи с използвания в упражнението симулатор. Проектите от вида C(minimal)/Assembler ще свържат в изпълнимата програма само базовите минимални библиотеки. В резултат програмата ще е по-малка. Тези проекти ще могат да работят в симулатора.

След създаване на проекта е необходимо да се добавят или създадат файлове в него. Това се извършва от *Project -> Add* или *File -> New*. Създаването на нов файл в проекта е показано на следващата фигура. В това упражнение ще се използват само файлове от тип Assembler, (които имат разширение *.s*).

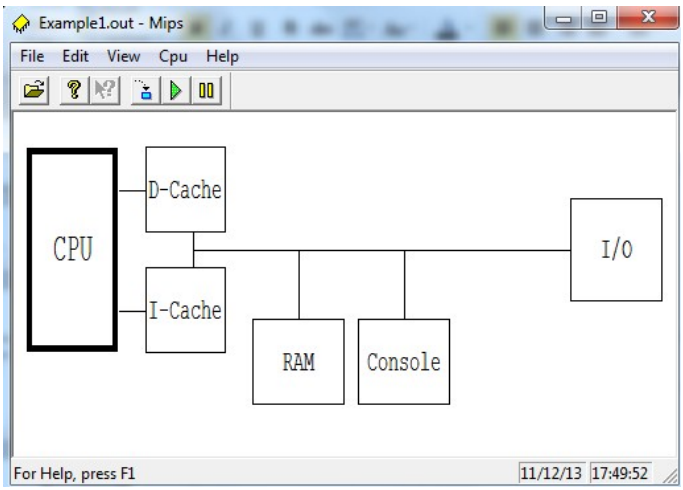
Като начало може да се въведе следния код:

```
#include <iregdef.h>
.set noreorder    # Предотвратява пренареждането на инструкциите
.text             # Дефинира началото на сегмента с инструкции
.globl start      # Дефинира етикет видим глобално
.ent start        # Дефинира етикет, който да обозначи началото на
програмата
start:
      add $8, $9, $10
      nop
      nop
      nop
      nop
.end start        # Обозначава края на програмата
```

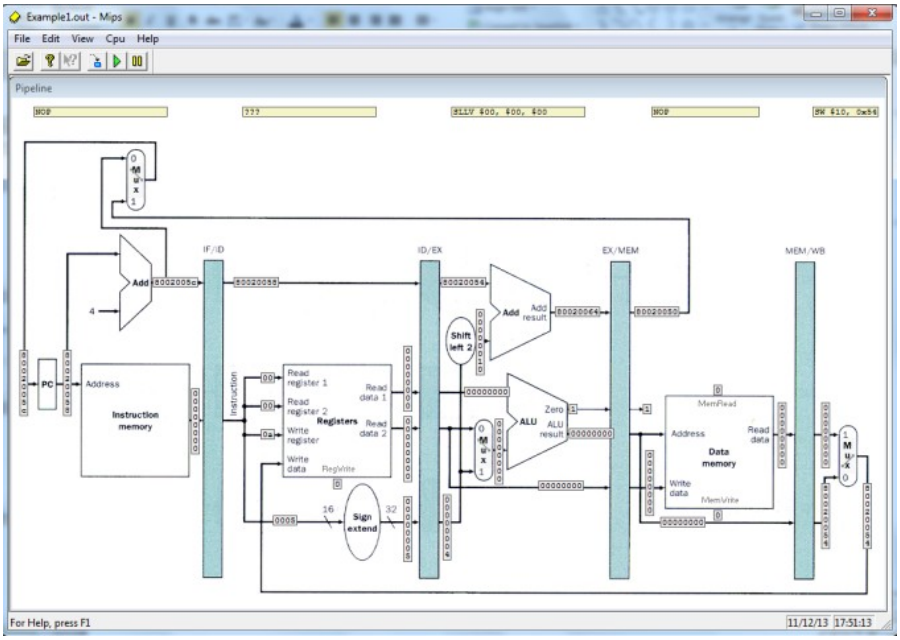


За да се подготви проектът за симулация се избира *Build* от менюто *Build* или се натиска *F7*. Всеки файл в проекта, който още не е компилиран ще бъде компилиран (или асемблиран) и накрая изпълнимият файл ще бъде свързан. Текущото състояние и резултатите от процеса на компилация могат да се видят в прозореца за съобщения в долната част. В случай, че е необходимо да се компилират отново всички файлове, дори и тези, които не са променени след последната компилация се избира *Rebuild All* от менюто *Build*. Когато компилацията и свързването с библиотеките е успешно можете да се премине към симулатора.

Сега може да се отвори програмата *MipsPipeS.exe* (в папката *C:\MipsIT\bin*), за да се направи симулация на изпълнението на програмата в конвейера на процесора. Необходимо е да се отиде отново в *MipsIt* и от менюто *Build* да се избере *Upload->To Simulator* (или да се натисне *F5*). В прозореца на симулатора се щраква върху блока CPU (както е показано на следващата фигура), за да се отвори прозореца, в който може да се проследи конвейерното изпълнение на програмата.



Следващата фигура показва прозореца, който трябва да може да се види ако са изпълнени предишните стъпки. С бутона *Step* или с командата *Cpu -> Step* програмата може да се изпълнява постъпково, за да се проследи как инструкциите преминават през етапите в конвейера.



III. Задачи за изпълнение:

Задача 1: В MipsIt променете програмата, представена по-горе като зададете стойности за регистрите $9 и $10 (може да използвате *lui* . . . инструкции преди инструкцията *add*). Компилирайте проекта и заредете в симулатора *MipsPipeS*. Изпълнете програмата постъпково и проследете какво се случва на всеки етап при преминаването на инструкциите през конвейера.

Забележка: Необходимо е да затворите и отворите отново MipsPipeS преди да можете да заредите програмата за симулация.

Отговорете на следните въпроси:

- Колко процесорни цикъла са необходими, за да може резултатът от операцията да бъде записан в регистъра *$8*? Правилен ли е резултатът? Как можете да промените кода, за да сте сигурни, че резултатът е правилен?
- На кой етап в конвейера се извършват операциите на аритметичните инструкции (например add и sub)?
- Кой етап не се използва от аритметичните инструкции? Защо?

Задача 2: Заменете инструкцията *add $8, $9, $10* в програмата с *lw $8, 0 ($9)*, компилирайте, заредете в симулатора и изследвайте изпълнението на програмата. Обърнете внимание, че трябва да дефинирате променлива с директивата *.data* в кода на асемблер, от която да заредите стойността (вижте „Индиректно и индексно адресиране“ в предишното упражнение).

Отговорете на следните въпроси:

- Каква аритметична операция извършва ALU? Защо?
- Колко процесорни цикъла са необходими, за да може в регистъра *$8* да се зареди стойност?
- Използват ли се всички етапи от конвейера? Защо?

Задача 3: Експериментирайте с инструкцията *sw $8, 4 ($9)* и се опитайте да съхраните определена стойност, която сте заредили в *$8* в паметта.

Отговорете на следните въпроси:

- Каква аритметична операция извършва ALU? Защо?
- Колко процесорни цикъла са необходими, за да се запише стойност в паметта?
- Използват ли се всички етапи от конвейера? Защо?

Задача 4: Изследвайте изпълнението на инструкцията *beq t0, t1, Dest*, по подобие на това, което направихте в предишните задачи. Имайте предвид, че някъде в кода на асемблер трябва да добавите етикет с име *Dest*.