Програмни модели за работа с процесорни инструкции (Част 1)

I. Въведение

Основната цел на настоящото упражнение е в неговия край Вие да можете да:

- Използвате директно и индиректно адресиране при достъпа до данни в паметта.
- Създавате итеративни цикли от инструкции.
- Визуализирате текст в конзола чрез използването на входно-изходни инструкции.
- Създавате подпрограми, да ги извиквате и да се връщате от тях.
- Предавате параметри на подпрограми.

II. Процесорни симулатори

Използването на симулатори спомага за по-доброто разбиране на теоретичните концепции, които се описват по време на лекциите. Симулаторите осигуряват визуално и анимирано представяне на механизмите на работа на системите и дават възможност на обучаващите се да наблюдават отвътре самата работа на тези системи, която освен че остава скрита за потребителите, е и трудно и ли дори невъзможно да се представи по друг начин. Освен това чрез използването на симулатори се позволява на обучаващите се да експериментират и изследват различните технологични аспекти на системите, без да се налага да инсталират и конфигурират реални системи.

III. Основни аспекти

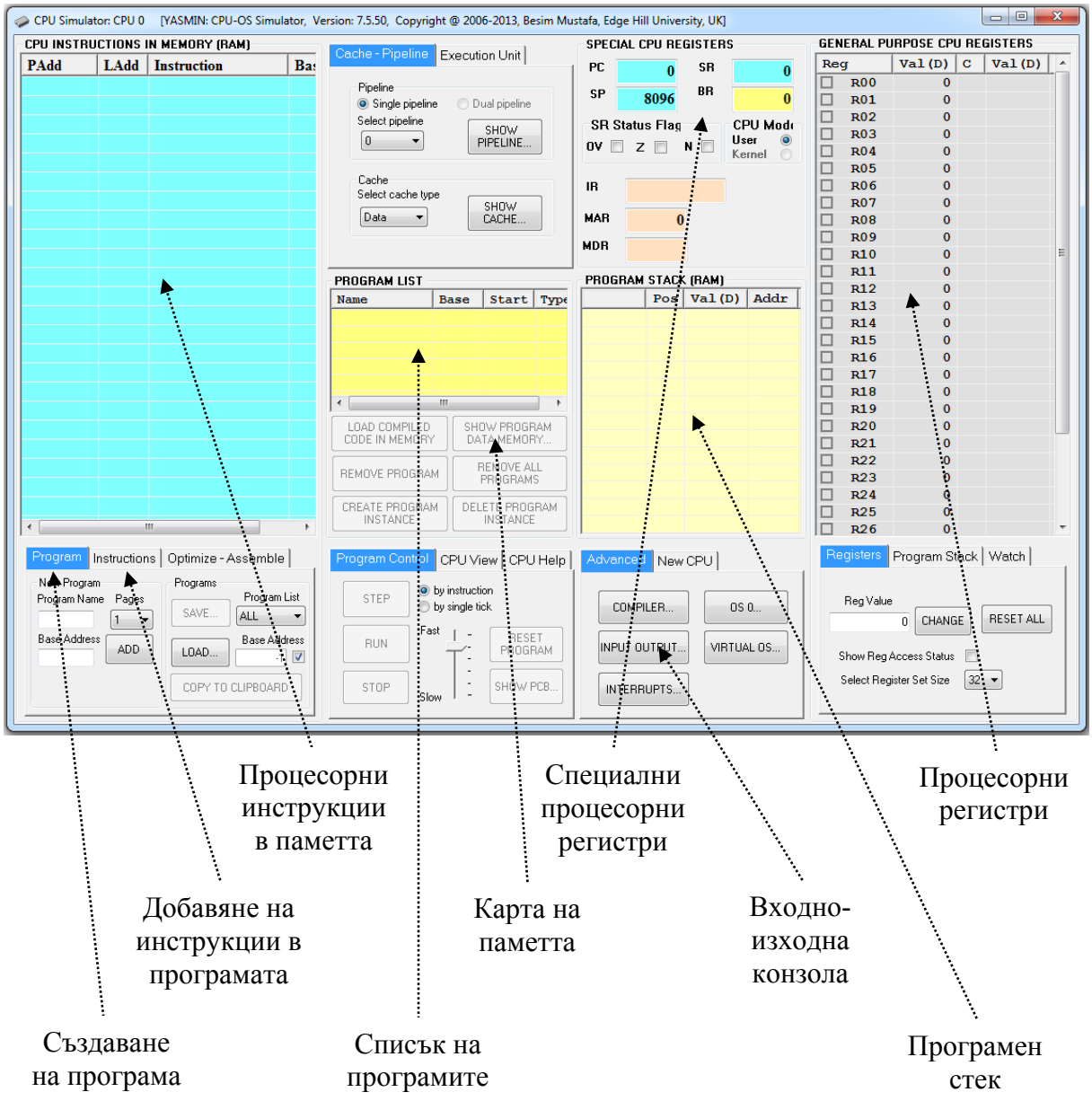
Програмният модел на компютърните архитектури дефинира архитектурните компоненти от ниско ниво, които участват пряко в работата на:

- Набора от инструкции на процесора
- Регистрите на процесора
- Различните видове адресиране на инструкциите и данните в тях

Програмният модел дефинира и взаимодействията между описаните по-горе компоненти. Всъщност именно програмния модел от ниско ниво е това, което позволява програмните изчисления.

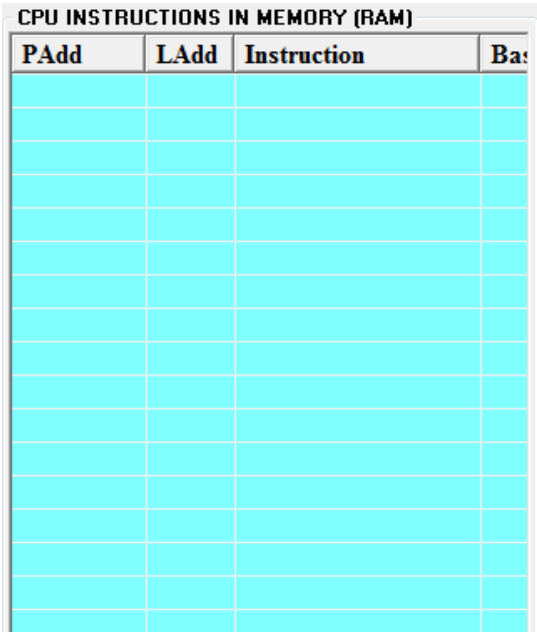
IV. Описание на симулатора

Основният програмен прозорец на симулатора е съставен от няколко елемента, които представляват различни функционални компоненти в рамките на симулирания процесор.



Елементите от симулатора, които ще се използват по време на днешното упражнение, са описани по-долу. Моля прочете внимателно тяхното описание и ги намерете преди започване на същинската част от настоящото упражнение.

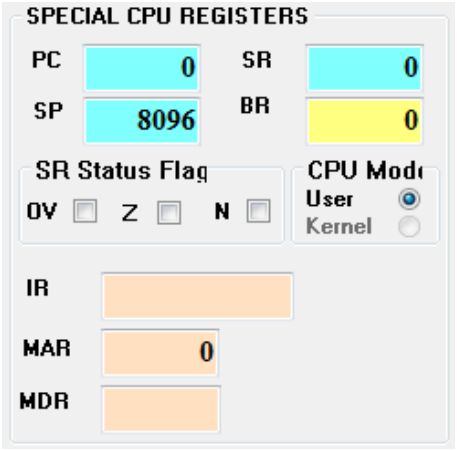
1. Процесорни инструкции в паметта



В този прозорец са показани инструкциите, от които се състои програмата. Те са представени като последователност от инструкционни мнемоники от ниско ниво (асемблерен вид), а не като бинарен код. По този начин кодът става по-ясен и по-лесно четим.

Всяка инструкция се асоциира с два адреса – физически (**PAdd**) и логически (**LAdd**). В този прозорец също се вижда и базовия адрес на всяка инструкция (**Base**). Всички инструкции от една програма имат един и същ базов адрес.

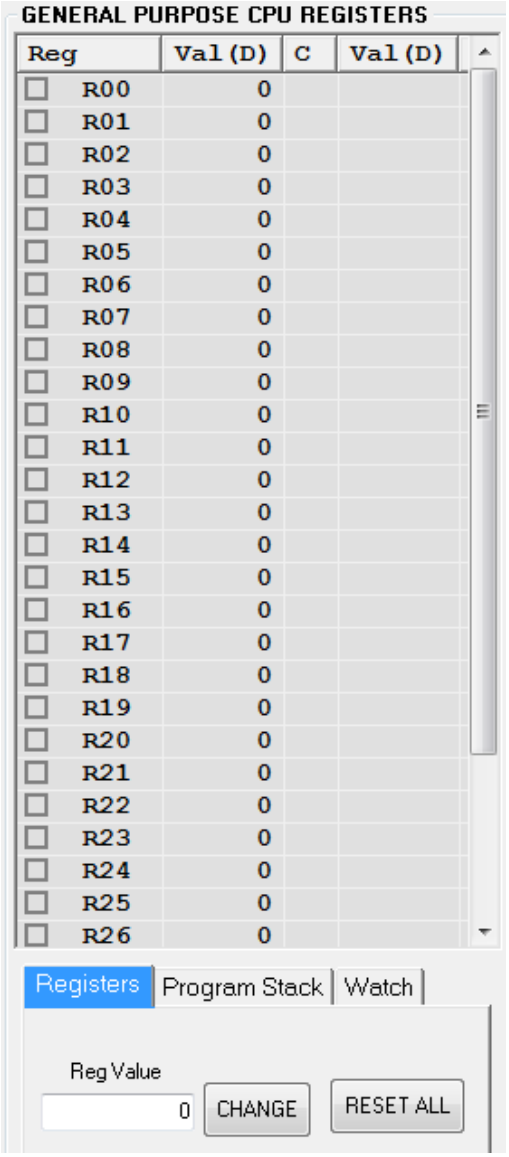
2. Специални процесорни регистри



В този прозорец са показани процесорните регистри, които имат предварително определени специализирани функции.
PC (Program Counter) – съдържа адреса на следващата инструкция, която трябва да се изпълни.
IR (Instruction Register) – съдържа инструкцията, която в момента се изпълнява.
SR (Status Register) – съдържа информация, която се отнася до резултата от последната изпълнена инструкция.
SP (Stack Pointer) – този регистър сочи към стойността, която се намира най-отгоре в програмния стек.
BR (Base Register) – съдържа текущия базов адрес.
MAR (Memory Address Register) – съдържа адреса от паметта, който в момента се достъпва.
MDR (Memory Data Register) – междинен регистър, който съдържа инструкцията, която в момента се изпълнява.
Status bits – битове за статус:

- **OV (Overflow)** – Препълване
- **Z (Zero)** – Нула
- **N (Negative)** – Отрицателен

3. Процесорни регистри

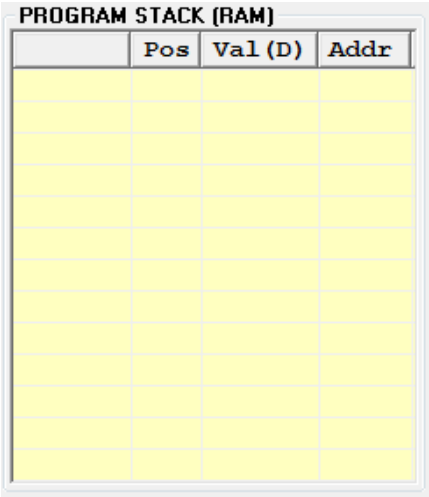


В този прозорец са показани стойностите на всички регистри с общо предназначение. Това са много бързи памети, които се използват за временно съхранение на данни по време на изпълнение на програмата. Най-често се използват за съхранение на променливи, използвани в програмните езици от високо ниво.

Броят на регистрите с общо предназначение варира в зависимост от архитектурата на процесора. Някои процесори имат повече такива регистри (напр. 128 регистъра), докато други по-малко (напр. 8 регистъра). Във всички случаи обаче те имат еднаква функция.

В този прозорец са показани имената на всички регистри с общо предназначение (**Reg**), техните текущи стойности (**Val**) и някои допълнителни елементи, които са резервирани за системата. Допълнително има възможност за ръчна промяна на стойността на всеки един от регистрите. Това става като най-напред се избере дадения регистър, след което в полето **Reg Value** се запише новата стойност и се натисне бутона **CHANGE**.

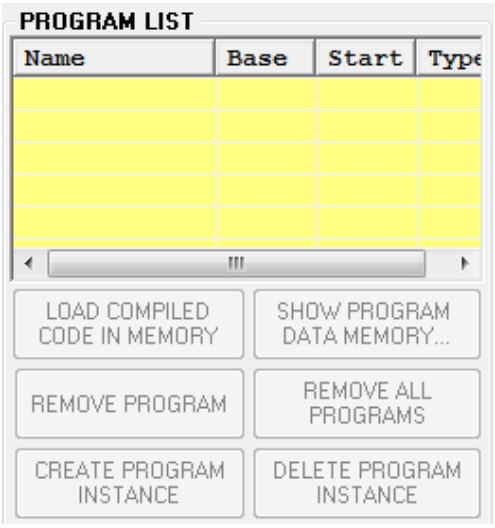
4. Програмен стек



Програмният стек е друго място, където временно се съхраняват данни по време на изпълнение на програмата. Структурата на стека е от тип LIFO (Last-In-First-Out). Той често се използва като средство за избягване на прекъсванията и извикване на подпрограми. Всяка програма има свой собствен стек.

Процесорните инструкции PSH (Push) и POP се използват за въвеждане и извличане на данни от върха на стека.

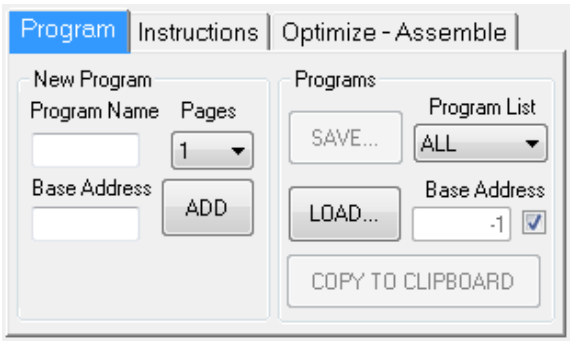
5. Списък на програмите



В този прозорец се визуализират създадените от нас програми.

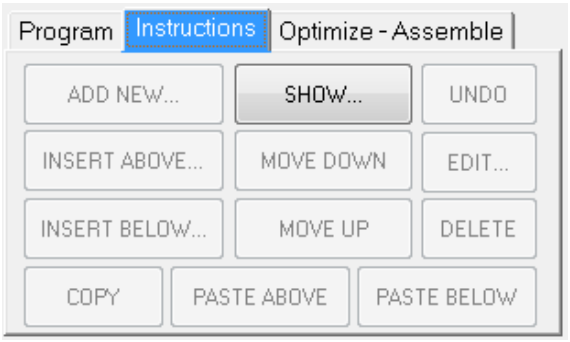
Бутонът **REMOVE PROGRAM** се използва за премахване на избрана програма от списъка, а чрез бутона **REMOVE ALL PROGRAMS** се премахват всички програми от списъка. Трябва да се има в предвид, че когато една програма се премахне от този списък, се премахват и всички нейни инструкции от Списъка с инструкциите.

6. Създаване на програма



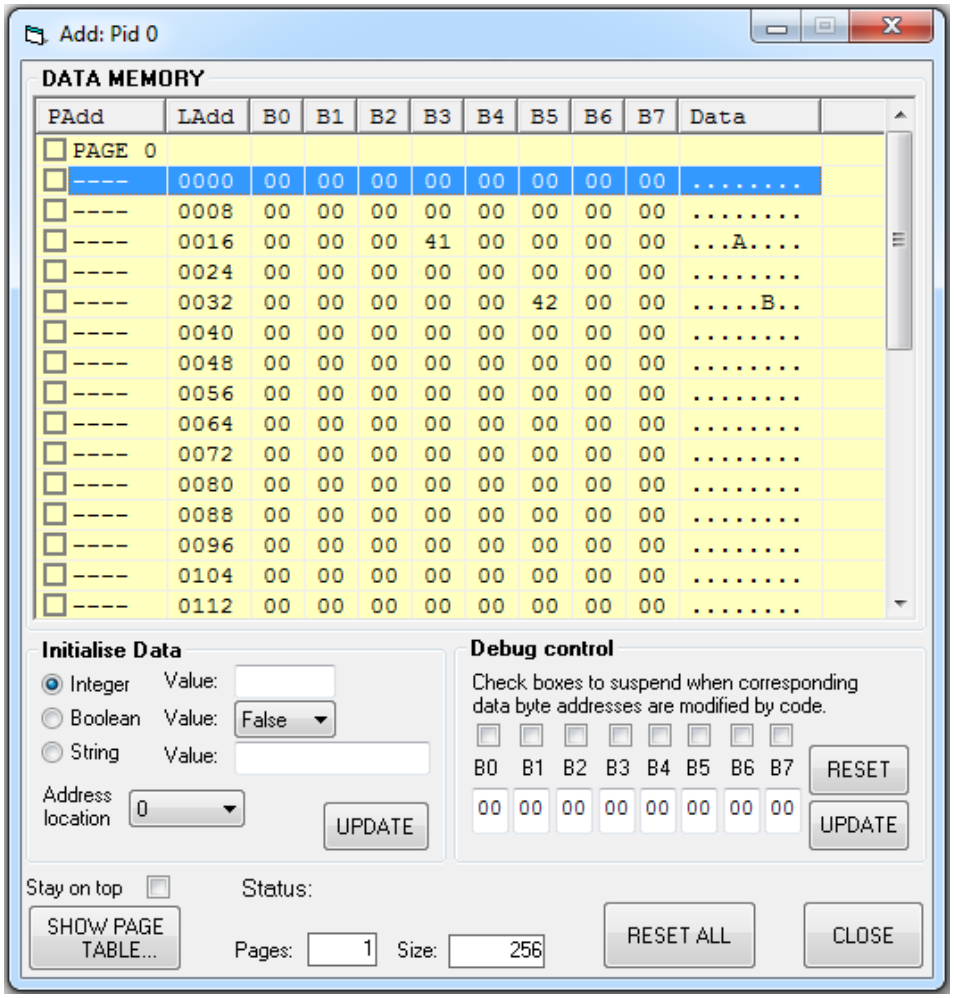
За да се създаде нова програма в полето **Program Name** се изписва нейното име, а в **Base Address** нейния базовия адрес. Чрез бутона **ADD** се създава новата програма и нейното име се появява в Списъка с програмите.

7. Добавяне и редактиране на инструкции



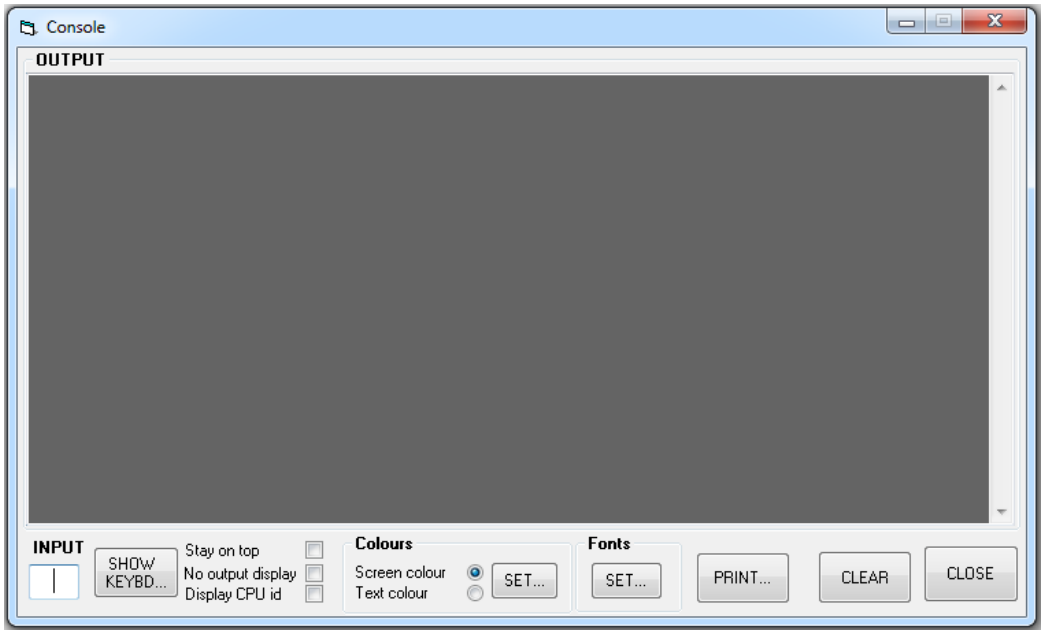
ADD NEW… – добавяне на нова инструкция в програмата
EDIT… – редактиране на избраната инструкция
MOVE DOWN/MOVE UP – преместване на избраната инструкция надолу/нагоре
INSERT ABOVE…/INSERT BELOW… – добавяне на нова инструкция над/под избраната инструкция

8. Карта на паметта



Инструкциите на процесора, които имат достъп до тази част от паметта, която съдържа данни, могат да ги четат и записват на точно определени адреси. Данните могат да се видят в прозореца, показващ страниците на паметта. Този прозорец се визуализира чрез натискането на бутона **SHOW PROGRAM DATA MEMORY…**. Колоната **LAdd** показва стартовия адрес на всеки ред от екрана. Всеки ред представлява 8 байта данни. Колоните **B0** до **B7** са байтовете от 0 до 7 на всеки ред. Колоната **Data** показва символите, които съответстват на всеки един от 8-те байта, при положение че могат да се визуализират. Байтовете, които съответстват на символи, които не могат да се визуализират, се показват като точки. Байтовете с данни се визуализират в шестнадесетичен формат. В прозореца по-горе можете да видите записаните данни в адреси 19 и 37. Те съответстват на латинските символи A и B. Симулаторът позволява ръчно да се променят всички данни, записани в паметта.

9. Входно-изходна конзола



Програмите използват входно-изходната конзола за четене и визуализиране на данни, необходими за тяхната работа. Тя се визуализира чрез натискането на бутона **INPUT OUTPUT…**

V. Описание на лабораторното упражнение

За да може да създаваме и изпълняваме инструкции най-напред трябва да създадем програма. В табът **Program** първо въвеждаме **Име на програмата**, а след това и **Базов адрес** (това може да бъде всякакво число, но за това упражнение нека бъде 100). Натискаме бутона **ADD**. Новосъздадената програма с нейното име се появява в Списъка с програмите. Чрез бутона **SAVE…** се записва самата програма и нейните инструкции във файл, а чрез бутона **LOAD…** се зарежда записаната програма в симулатора.

След като сме изпълнили стъпките по-горе може да пристъпим към създаването на инструкции в симулатора. В таба **Instructions** натискаме бутона **ADD NEW…**, след което се появява нов прозорец **Instructions: CPU0**. От там избираме процесорната инструкция и евентуално настройваме нейните параметри, като за всяка инструкция има и кратко описание. В Приложение № 1 към настоящото упражнение има списък от няколко инструкции, както и примери за тяхното използване.

Вече сме готови да започнем същинската част от днешното упражнение. Отговорите на всяка от точките по-долу записвайте в определените за това текстови полета. *Препоръчително е редовно да записвате създадената програма във файл, така че ако симулатора поради някаква причина се повреди, да можете да го рестартирате и да заредите програмата от там, докъдето сте стигнали.*

1. В Приложение № 1 към настоящото упражнение намерете инструкцията, която се използва за съхраняването на един байт данни в адрес от паметта. Използвайте я за да съхраните числото 65 в адрес 20 (всички числа са в десетичен формат). Това е пример за директно адресиране. Отворете картата на паметта и вижте какво се е записало там.

2. Създайте инструкция, която премества десетичното число 22 в регистър R01. Изпълнете я.

3. Създайте инструкция, която съхранява десетичното число 51 в адреса от паметта, който е записан в регистър R01. Това е пример за косвена (индиректна) адресация. Обърнете внимание, че се използва символа **@** преди R01.

4. Какво се е записало на позиции 20 и 22 в паметта?

5. Нека да създадем цикъл. Въведете в своята програма кода по-долу. Обърнете внимание на символа #, който се използва като префикс и означава че стойността след него има буквален смисъл. Ако такъв символ липсва, то стойността има смисъла на адрес. В кодът по-долу регистъра **R01** е напълно случайно избран. Може да се използва всеки регистър от **R00** до **R31**.

```
MOV #0, R01
ADD #1, R01
CMP #5, R01
JNE 0
HLT
```

6. Кодът по-горе все още не е напълно готов. Инструкцията **JNE** използва цифрова стойност като адрес, към който да направи прехода. В случая това е 0. В повечето случаи обаче нещата не са толкова прости, така че за да направим нашия код по-гъвкав би било добре да използваме етикет (label) като адрес за прехода.

   Изберете инструкцията **MOV**
   Натиснете бутона **INSET BELOW…**
   В отворилия се прозорец натиснете бутона **INSERT LABEL…**
   Въведете **L0** като име на етикета и потвърдете с **OK**
   Новият код трябва да изглежда по следния начин:

```
MOV #0, R01
L0:
ADD #1, R01
CMP #5, R01
JNE 0
HLT
```

   Изберете инструкцията **JNE**
   Натиснете бутона **EDIT…**
   Изберете **L0** от падащото меню, намиращо се под **Value** в **Source Operand**
   Натиснете бутона **EDIT…**
   Новият код трябва да изглежда по следния начин:

```
MOV #0, R01
L0:
ADD #1, R01
CMP #5, R01
JNE $L0
HLT
```

7. Както се вижда, етикетът **L0** има адреса на инструкцията точно под него, т.е. адреса на инструкцията **ADD**. По този начин инструкцията **JNE** може да използва **L0** като адрес за преход. Тъй като **L0** може да представлява всякакъв адрес, този код може да се използва навсякъде в програмата без ограничение. Знакът **$** означава, че **L0** е етикет. Нашият код е готов за работа.

   Натиснете бутона **RESET PROGRAM**
   Изберете инструкцията **MOV** (първата инструкция от програмата)
   Коригирайте скоростта на изпълнение на програмата от плъзгача и задайте стойност около 80
   Натиснете бутона **RUN**

След кратко време програмата трябва да спре. Ако изпълнението й продължи твърде дълго, натиснете бутона **STOP** и проверете своя код за грешки. При необходимост го редактирайте и повторете всички стъпки отначало.
Когато програмата спре, каква е стойността на регистър **R01**?

8. Сега нека малко да модифицираме нашия код. Променете го така, че цикъла да се повтаря докато стойността на регистъра **R01** е по-малка или равна на 3. Тествайте го. Запишете модифицирания код в полето по-долу, след което го върнете в първоначалния му вид (може да използвате бутона **UNDO** за това).

9. Хайде сега да създадем елементарна подпрограма. Въведете кода по-долу. Трябва да създадем нов етикет **L1** в началото на подпрограмата, който ще бъде и началния адрес на тази подпрограма. Етикетът се въвежда чрез бутона **INSERT LABEL…**, както е описано по-горе. Освен това задължително трябва да сме избрали радио бутона **Direct Mem** когато въвеждаме стойността 24 за първия операнд в инструкцията **OUT**.

```
L1:
OUT 24, 0
RET
```

10. Така създадената подпрограма единствено визуализира текста, който е записан в паметта и започва от адрес 24, след което се връща обратно в основната програма. За да може обаче да работи, трябва да има някакъв текст, който да е записан в адрес 24. Това може да се направи и ръчно:

    Натиснете бутона **SHOW PROGRAM DATA MEMORY…**
    В отворилия се прозорец изберете ред 0024
    Изберете радио бутона **String**, който се намира под **Initialise Data**
    Въведете произволен текст в текстовото поле
    Натиснете бутона **UPDATE**

11. Създадената подпрограма е безполезна сама по себе си. За да може да се използва от основната програма, тя трябва да се извика чрез инструкциите **MSF** и **CAL**. Инструкцията **MSF** (Mark Stack Frame) запазва място в програмния стек, където се записва адреса, откъдето трябва да продължи програмата след като подпрограмата завърши своята работа. Инструкцията **CAL** определя стартовия адрес на подпрограмата. Нека да модифицираме нашата програма по такъв начин, че когато извикваме подпрограмата тя да визуализира многократно текста в цикъл.

```
MOV #0, R01
L0:
ADD #1, R01
MSF
CAL $L1
CMP #5, R01
JNE $L0
HLT
L1:
OUT 24, 0
RET
```

12. Така създаденият код е готов за работа. За да може да се визуализира текста, трябва да се стартира входно-изходната конзола чрез натискането на бутона **INPUT OUTPUT…**

    Натиснете бутона **RESET PROGRAM**
    Изберете инструкцията **MOV** (първата инструкция от програмата)
    Коригирайте скоростта на изпълнение на програмата от плъзгача и задайте стойност около 80
    Натиснете бутона **RUN**

13. Трябва да направим някои малки промени в подпрограмата за да стане по-гъвкава. В момента инструкцията **OUT** използва директно адресиране, т.е. адрес 24 от паметта е част от самата програма. По-добре би било използването на косвена адресация, така както го направихме в Точка 3. Това, което трябва да направим, е да сложим числото 24 в произволен регистър. След това трябва така да модифицираме инструкцията **OUT**, че да използва този регистър като място, където е записан адреса, откъдето трябва да се прочете текста, който ще се визуализира. Тествайте програмата. Запишете частта от кода, която променихте, в полето по-долу, след което го върнете в първоначалния му вид (може да използвате бутона UNDO за това).

14. Сега нека още малко да усложним нашата програма, като заменим цикъла в нея с друга подпрограма. По този начин ще имаме две подпрограми, като едната ще извиква другата. Кодът по-долу показва именно това. Забележете че инструкцията **HLT** е заменена с **RET**, а новите инструкции **MSF**, **CAL** и **HLT** са сложени заедно с новия етикет **L2** най-отгоре в кода. **CAL $L2** извиква подпрограмата с цикъла, а **CAL $L1** извиква тази, която визуализира текста.

```
MSF
CAL $L2
HLT
L2:
MOV #0, R01
L0:
ADD #1, R01
MSF
CAL $L1
CMP #5, R01
JNE $L0
RET
L1:
OUT 24, 0
RET
```

Натиснете бутона **RESET PROGRAM**
Изберете първата инструкция **MSF**
Коригирайте скоростта на изпълнение на програмата от плъзгача и задайте стойност около 80
Натиснете бутона **RUN**
Сравнете получените резултати във входно-изходната конзола с предишните такива.

15. Така създадената програма ще направи 5 цикъла, като обаче това число е фиксирано. За по-голяма гъвкавост можем да предадем броя на циклите за изпълнение като параметър на подпрограмата, започваща от етикет **L2**. За да направим това нещо възможно, трябва да използваме инструкциите **PSH** и **POP**. Модифицирайте кода, така че да изглежда по следния начин, след което го стартирайте и вижте какво ще се визуализира в конзолата.

```
MSF
PSH #8
CAL $L2
HLT
L2:
```

```
POP R02
MOV #0, R01
L0:
ADD #1, R01
MSF
CAL $L1
CMP R02, R01
JNE $L0
RET
L1:
OUT 24, 0
RET
```

16. Наблюдавайте какво се случва по време на изпълнението на кода и обяснете как става предаването на параметъра.

17. Като последна задача за днешното упражнение модифицирайте кода по такъв начин, че и втори параметър да се предава на подпрограмата, започваща от етикет **L2**, така както се предава и първия. Вторият параметър да се използва за инициализиране на първоначалната стойност на регистър **R01**. Представете готовия код под формата на протокол за днешното упражнение.

Приложение № 1

Инструкция	Описание
Инструкции за трансфер на данни	
MOV	Премества данни в регистър Премества данни от един регистър в друг регистър **MOV #2, R01** – Премества числото 2 в регистър R01 **MOV R01, R03** – Премества съдържанието на регистър R01 в R03
LDB	Зарежда байт от паметта в регистър **LDB 1022, R03** – Зарежда байт от адрес 1022 в регистър R03 **LDB @R02, R05** – Зарежда байт от адреса, който е записан в регистър R02, в регистър R05
LDW	Зарежда дума (2 байта) от паметта в регистър Работи по същия начин като LDB, но се зарежда дума (2 байта) в регистър
STB	Съхранява байт от регистър в паметта **STB R07, 2146** – Съхранява байт от регистър R07 в адрес 2146 **STB R04, @R08** – Съхранява байт от регистър R04 в адреса, който е записан в регистър R08
STW	Съхранява дума (2 байта) от регистър в паметта Работи по същия начин като STB, но се съхранява дума (2 байта) в регистър
PSH	Въвежда данни най-отгоре в хардуерния стек Въвежда данни от регистър най-отгоре в хардуерния стек **PSH #6** – Въвежда числото 6 най-отгоре в стека **PSH R03** – Въвежда съдържанието на регистър R03 най-отгоре в стека
POP	Извлича данните, които се намират най-отгоре в хардуерния стек, и ги записва в регистър **POP R05** – Извлича данните, които се намират най-отгоре в стека, и ги записва в регистър R05 *Ако се опитаме да извлечем данни от празен стек се получава грешка „Препълване на стека (Stack overflow)“*
Аритметични инструкции	
ADD	Събира число с регистър Събира регистър с регистър **ADD #3, R02** – Събира числото 3 със съдържанието на регистър R02 и записва резултата в R02 **ADD R00, R01** – Събира съдържанието на регистър R00 със съдържанието на регистър R01 и записва резултата в R01
SUB	Изважда число от регистър Изважда регистър от регистър
MUL	Умножава число с регистър Умножава регистър с регистър

DIV	Разделя число с регистър Разделя регистър с регистър
Инструкции за контрол на трансфера	
JMP	Безусловен преход към адрес на инструкция **JMP 100** – Безусловен преход към адрес 100, където има друга инструкция
JLT	Преход към адрес на инструкция, ако е по-малко от (след последното сравнение)
JGT	Преход към адрес на инструкция, ако е по-голямо от (след последното сравнение)
JEQ	Преход към адрес на инструкция, ако е равно (след последното сравнение) **JEQ 200** – Преход към адрес 200, ако при предишното сравнение са сравнени две еднакви числа, т.е. флага Z е установен
JNE	Преход към адрес на инструкция, ако не е равно (след последното сравнение)
MSF	Използва се заедно с инструкцията CAL **MSF** – Запазва място в програмния стек, където се записва адреса за връщане **CAL 1456** – Записва адреса за връщане на запазеното място в програмния стек и прави преход към адрес 1456, където се намира подпрограмата
CAL	Запис на адрес за връщане в програмния стек и преход към адрес на подпрограма Тази инструкция се използва заедно с инструкцията **MSF** Задължително трябва да има инструкцията **MSF** преди **CAL**
RET	Връщане от подпрограма (използва адреса за връщане от стека)
SWI	Софтуерно прекъсване
HLT	Стоп на симулация
Инструкции за сравнение	
CMP	Сравнява число с регистър Сравнява регистър с регистър **CMP #5, R02** – Сравнява числото 5 със съдържанието на регистър R02 **CMP R01, R03** – Сравнява съдържанието на регистрите R01 и R03 Ако R01 = R03, тогава флага Z се установява Ако R01 < R03, тогава никои от флаговете не се установява Ако R01 > R03, тогава флага N се установява
Инструкции за вход и изход	
IN	Извлича входни данни (ако са налични) от външно входно-изходно устройство
OUT	Изпраща данни към външно входно-изходно устройство **OUT 16, 0** – Изпраща данните от адрес 16 към конзолата (втория параметър задължително трябва да бъде 0)