**Програмен модел 2**

**A. Въведение**

**Цели:**

В края на занятието вие трябва да можете да:

 използвате директно адресиране на паметта

 Създаване на различни цикли от инструкции

 Показвате текст в конзолата чрез В/И инструкции

 Създавате,влизате и излизате от подцикъл

 предавате параметри на подцикъл

**B. Процесорни (CPU) Симулатори**

Ръководството по комп.арх. са писани за симулатори, които осъществяват теоретично възможностите на архитектурата, обяснявана по време на лекциите. Симулатора предуставя визуалност на механизма на работа и дава възможност на студентите да наблюдават скритата вътрешна работа на системата, невъзможно да бъде наблюдавана по друг начин. Предимството в използването му е възможността да се експериментира и открива различна способност на технологиите, без нуждата от истинска система

**C. Основна теория**

Програмният модел на компютърната архитектура ви показва следните възможности на ниско ниво:

 набор от CPU инструкции

 CPU регистри

 Различни начини за адресиране на информация чрез инструкции

Също така се използват и връзки между гореописаните. Ниското ниво на програмиране чрез команди позволява извършването на различни действия. Трябва да прочетете допълнителна информация за да подобрите разбирането си за модерните CPU архитектури.

**D. Симулатора в детайли**

Тук се запознавате с основна информация за симулатора, за да можете да направите работещ модел.

Главният прозорец на симулатора се състои от няколко части, представляващи различна част от симулирания процесор. Показани са на снимка 1

 CPU памет за инструкции - 1

 Специални CPU регистри - 2

 CPU (основни) регистри - 3

 Програмен стек - 4

 Създаване и пускане на програма - 5

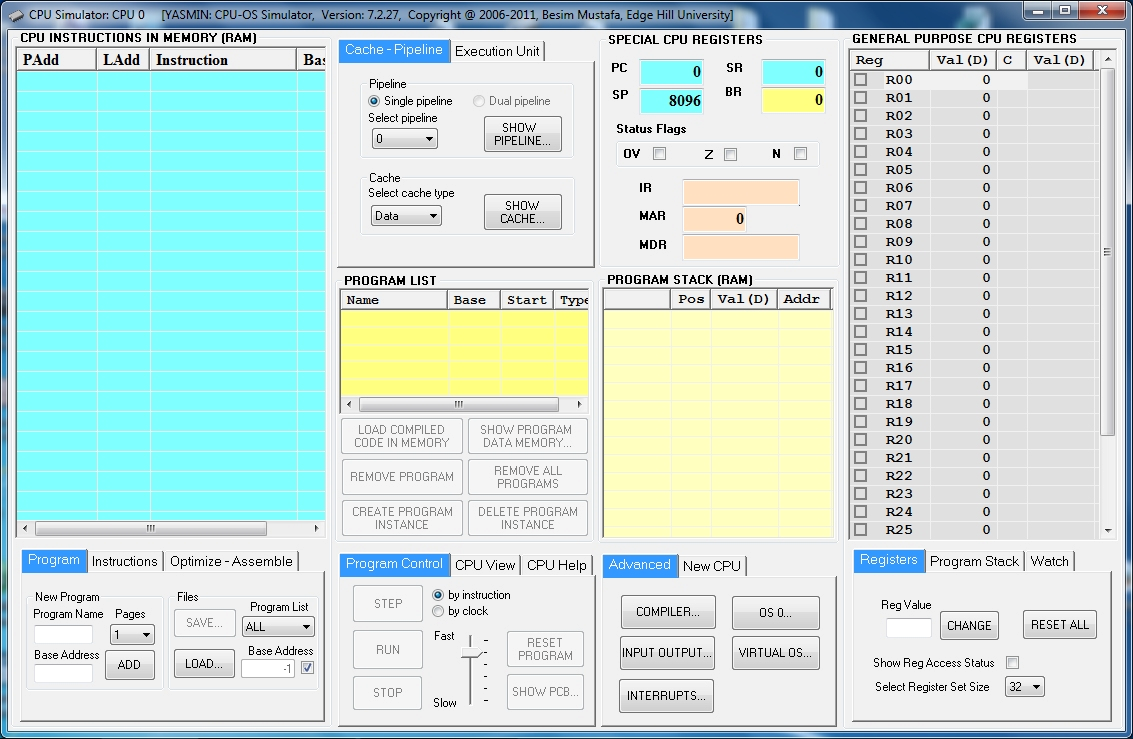
 Памет за съхранение на информация - 6

 Входно – изходна конзола - 7

CPU Instruction memory view

Add program

2



3

instructions tab

Create program tab

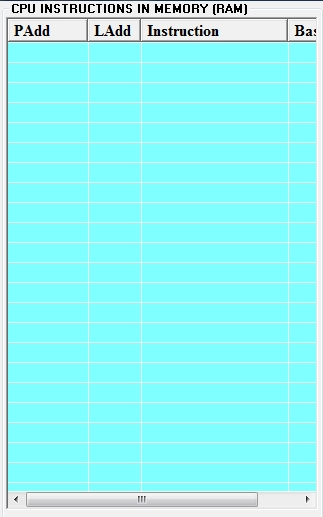
Program list view

Click to view program data memory

4

Конзола

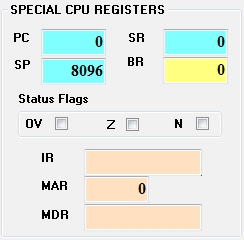
**1. CPU памет с инструкции**



Тук се съдържат програмните инструкции. Инструкциите се образуват от поредица от инструкции на ниско ниво (асемблерен формат) а не като двоичен код. Това е направено за да е кода по-четлив от хората.

Всяка инструкция има два адреса: физически (**PAdd**) и логически (**LAdd**). Тук се вижда и базисния адрес (**Base**) срещу всяка инструкция. Поредицата от инструкциипринадлежаща на една и съща програма ще има един и същ базисен адрес.

**2. Специални CPU регистри** тук са показани набор от CPU регистри, с предварително дефинирани специални функции:



**PC**: **Program Counter** съдържа адреса на следващата инструкция за изпълнение.

**IR**: **Instruction Register** моментно изпълняващата се инструкция.

**SR**: **Status Register** информация за реултата от последно изпълнената инструкция.

**SP**: **Stack Pointer** показва стойност за най-горната стойност на стека

**BR**: **Base Register** съдържа базисния адрес

**MAR**: **Memory Address Register** съдържа адреса от паметта който в момента се използва

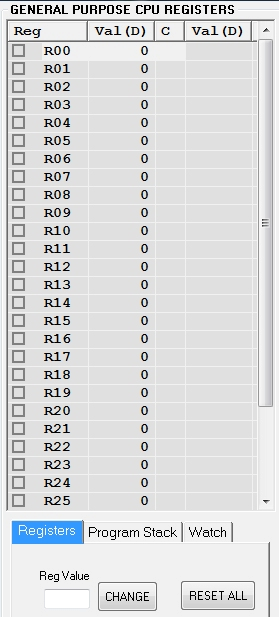
**Status bits: OV**: препълване **Z**: нула; **N**: отрицание

**3. CPU изглед на регистрите**

Показва се съдържанието на всички основни регистри, съхраняващи временни стойности по време на изпълнението на инструкциите. Регистрите са бързи блокове памет, които съдържат информация за CPU инструкциите.

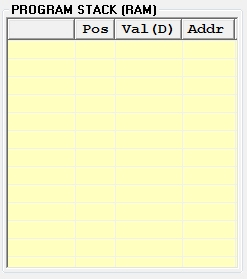
Архитектурата поддържа от 8 до 64 регистъра. Те се използват за да пазят стойности въведени от езици с високо ниво на програмиране.

Не всички архитектури имат толкова регистри. Някои имат повече (128 регистъра) а други по малко ( 8 регистъра). Във всички случаи тези регистри се използват за едни и същи дейности.



В програмата са показани регистърните имена (**Reg**), стойността (**Val**) и други опции, спомагащи дебъгването на програмата. Може да се използват и за ръчна промяна на стойностите, което е полезно при дебъгване. За ръчна смяна на стойност, селектирайте регистъра и попълнете новата стойност в кутииката, **Reg Value**, натиснете **CHANGE** бутона в **Registers** таблицата.

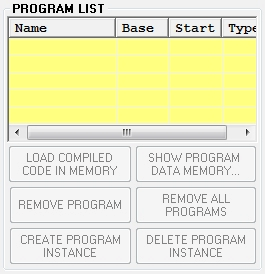
**4. Програмен стек**



Стек е другото място където се съхраняват временни стойности доакто инструкциите се изпълняват. Стека е LIFO (last‐in‐first‐out) структура от данни. Често се използва за справяне с ефикасни прекъсвания и извикване на подрутинни изнтрукции. Всяка програма си има собствен стек.

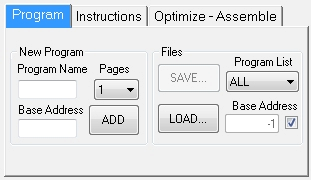
CPU инструкциите PSH и POP се използват за вмъкване и изваждане на стойности от стека.

**5. Програмен лист**



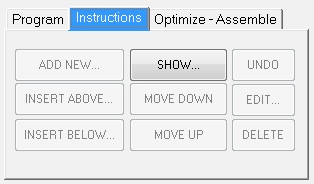
Използвайте **REMOVE PROGRAM** за да махнете програмата от листа; използвайте **REMOVE ALL PROGRAMS** за премахване на всички програми от листа. Забележете, че когато програма е премахната, инструкциите също са премахнати от **Instruction Memory View**.

**6. Създаване на програма**

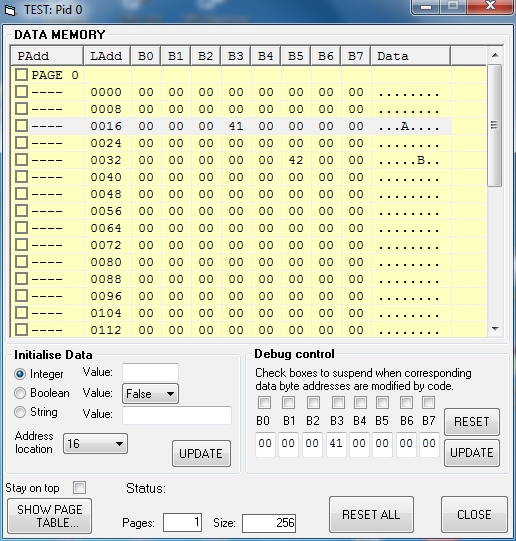


За създаване на нова програма въведето името й в **Program Name** и базисния и адрес в **Base Address** ,натиснете на бутон **ADD**. Името на новата програма ще се появи в листа.

Използвайте **ADD NEW…** за добавяне на нова инструкция; **EDIT…** за промяна на инструкция; **MOVE DOWN**/ **MOVE UP** за местене на инструкцията нагоре/надоло; **INSERT ABOVE…**/**INSERT BELOW…** за вмъкване на нова инструкция над или под съществуваща такава.



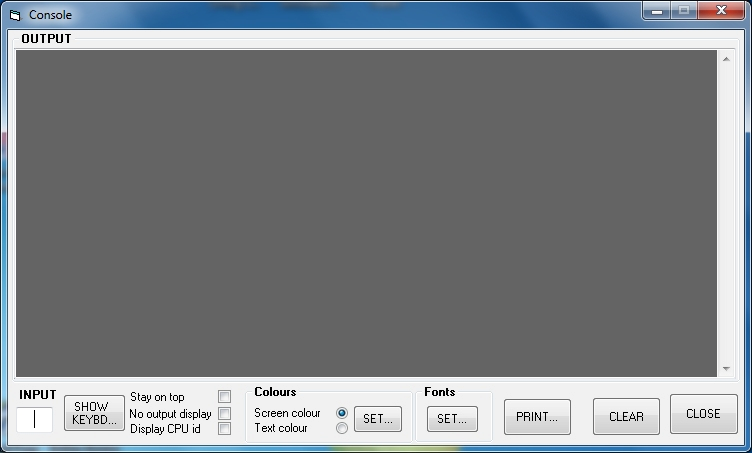
**7. Програмна информация**



CPU инструкциите които достъпват паметта с информация може да пишат или четат информация от адресираните пространства. Тази информация може да бъде видяна на страницата показана по горе. Отваряте прозореца чрез натискане на **SHOW PROGRAM DATA MEMORY…**. **Ladd** (логически адрес) колоната показва стартовия адрес. Всеки ред представлява 8 байта информация. Колони **B0 до** **B7 са байтове** 0 до 7 за всеки ред. Колоната **Data** показва визуализираните знаци отговарящи на 8-те байта. Тези които не могат да бъдат показани са изобразени с точки. Информацията е показана в шестнайсетичен формат. Например, на снимката, има ненулеви байтове на адреси 19 и 37. Те отговарят за знаците A and B.

За смяна на стойност, изберете реда/редовете със байтовете. След това използвайте информацията в **Initialize Data** рамка за да пормените стойностите на избраните байтове - **Integer**, **Boolean** или **String формати**. натиснете **UPDATE** за да запазите промените.

**8. Входно изходна конзола**



Снимката отгоре показва конзолата изплзвана от програми за изписване на съобщения и четене на данни.показва се при натискане на **INPUT OUTPUT…**. натиснете **SHOW KEYBD…** за да изкарате малка клавиатура която може да ползвате за да вкарвате данни в програми.

**E Изследване и откриване**

Лабораторните упр. са набор от действия, правени от студентите с леки напътствия и помощ. Как това ръководство е организирано? Студентите следват дадените инстуркции за да достигнат до нужната им информация, да се възползват от нея, записват си наблщдаваното и обясняват при нужда.За да могат да направят това трябва да се запознаете с информацията от секция D и към приложението в края на файла за различните процесорни инструкции които ще трябва да измислите и използвате.

Нека започнем. Като за начало трябва да сложите инструкции в **Instruction Memory View** (снимка 2)**,** симулирайки RAM в реална машина, преди да изпълните каквито и да е инструкции. За да направите това следвайте тези стъпки:

В полето **Program**  (снимка 7), въведете **Program Name**, а после **Base Address** (може да е всяко число, но за това упр нека е 100). Натиснете **ADD** . ново програмно име ще се добави в **Program List** (снимка 6). Можете да използвате **SAVE…** за запазване във файл. Също и **LOAD…** за зареждане от файл.

Готови сте за въвеждане на инструкции в симулатора. Натиснете **ADD NEW…** в полето **Instructions**  (снимка 8). Ще покаже **Instructions: CPU0** прозорец. Ипозлвате прозореца за избор и слагане на процесорни инструкции. **В ПРИЛОЖЕНИЕТО има инструкции които процесора използва както и за какво се ползват**.

1 В приложението намерете инструкцията която запазва еидн byte инф. в клетка. запишете 65 на адрес 20 (всички числа са десетинчни). Това е пример за директна адресация( **direct addressing)**. Запишете използваната инструкция:

2. Преместете числото 22 в R01 и запишете как по-доло. Изпълнете тази команда и проверете R01.

3. запишете 51 в паметта на място което е указано в R01. Това е пример за недиректна адресация (**indirect addressing)**. Обърнете внимание на “**@”** до R01 в този случай.

4. запишете какво виждате в паметта на място 20 и 22 (вижте снимка 9 за да разберете как се наблюдава паметта).

5. Нека направим цикъл. Въведете следният код. **#** префикса се използва за да се сложи реална стойност вземайки я от адрес който не я използва. **R01** е временен регистър; използвайте всеки регистър от **R00** до **R31**.

**MOV #0, R01**

**ADD #1, R01**

**CMP #5, R01**

**JNE 0**

**HLT**

6. Горният код не е завършен. **JNE** инструкцията има нужда от числова стойност за адреса на който са скочи. В случая е 0. Не винаги ще е така, затова за да направите кода гъвкавможе да използваме етикети за връзка с адрес на инструкция. Симулатора ви позволява това.

Маркирайте **MOV** инструкцията

натиснете **INSERT BELOW…**

напишете име на маркера **L0** до **ENTER LABEL** в прозореца

натиснете **ENTER LABEL**

новият код изглежда така (проените са в червено):

**MOV #0, R01**

**L0:**

**ADD #1, R01**

**CMP #5, R01**

**JNE 0**

**HLT**

Маркирайте **JNE** инструкцията

натиснете **EDIT…**

изберете **L0** от падащият списък при **Source Operand** в прозореца в който вкарвате операнди

натиснете **EDIT**

новият код изглежда така:

**MOV #0, R01**

**L0:**

**ADD #1, R01**

**CMP #5, R01**

**JNE $L0**

**HLT**

7. Както виждате **L0** представлява адреса на инструкцията която веднага следва в случая **ADD** инструкция. Сега **JNE** може да използва **L0** като адрес на който да скочи. Както и **L0** може да отговаря за всеки адрес и така кода е валиден насякаде за паметта и прави работата по лесна. знака $ показва че L0 е маркер. Кода е готов за пускане.

Натиснете на **RESET PROGRAM** в прозореца на симулатора и маркирайте **MOV** инструкцията, ч случая първата инструкция в програмата, увеличете слайдера за скорост до стойност близка до 80

натиснете **RUN**

След кратко време програмата трябва да спре. Ако се изпълнява много дълго натиснете **STOP** и вижте кода си. Поправете го и повторете горните инструкции.

Запишете стойността на **R01**

8. Сега ще промените малко програмата. Променете кода така, че цикъла да се върти докато **R01** е по-малък или равен на 3 и тествайте. След като я подкарате запишете стойността на **R01** и новият код. Върнете се на оригиналните инструкции (използвайте **UNDO**).

9. Нека направим лесна под-рутина. Въведете следния код. Лъздайте нов етикет **L1** в началото на под-рутината. Той е за стартовия адрес. Въведете бележката с **ENTER LABEL**. изберете **Direct Mem** при въвеждане на първата операндакато стойност - 24 на **OUT** инструкцията:

**L1:**

**OUT 24, 0**

**RET**

10. Горния код изкарва текста започващ от адрес 24 и връща (прегледайте **RET** инструкцията). За да работи трябва да има текст на адреса на място 24. Задайте го като следвате тези стъпки:

Натиснете **SHOW PROGRAM DATA MEMORY…**. маркирайте 0024 под **LAdd** колоната под **Initialise Data** натиснете на **String** опцията.

Въведете текст в **Value**,например. *My name is Besim*

Натиснете **UPDATE**

11. Все още суб-рутината не е готова. За да работи програмата трябва да я извика използвайки инструкцията **MSF** последвана от **CAL**. **MSF** (Mark Stack Frame) заделя място за адрес който да се върне в програмния стек. **CAL** показва адреса на низа където се намира изречението. Нека променим кода така че да показва текста и да го повтаря в цикъл.

**MOV #0, R01**

**L0:**

**ADD #1, R01**

**MSF**

**CAL $L1**

**CMP #5, R01**

**JNE $L0**

**HLT L1:**

**OUT 24, 0**

**RET**

12. КОдът е готов за пускане. За да видите текст трябва да отворите конзолния прозорец. Натиснете на **INPUT OUTPUT…** след това:

Натиснете **RESET PROGRAM**

маркирайте **MOV** инструкцията,

увеличете слайдера за скорост на 80

натиснете **RUN**

13. Сега трябва да направим лека промяна. В момента **OUT** използва директна адресация,тоест адреса 24 е част от инструкцията. Ще сложим адрес 24 в свободен регистър. След това **OUT** ще използва този регистър индиректно като изтончик на адреса от който да покаже текста. Пуснете и запишете доло модифицираният код. Използвайте **UNDO** за да върнете инструкциите преди тази модификация:

14. Нека направим нещо по-сложно. Нека направим цикъла като друга под-рутина и да го извикаме. Така ще имаме две под-рутини и едната ще вика другата. Забелечете, че **HLT** се заменя с **RET** и новите инструкции **MSF, CAL** и **HLT** са добавени заедно с новата бележка L2 най-отгоре на кода.

CAL $L2 вика суб-рутината с цикъла, а CAL $L1 показва 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**

Рестартирайте а после маркирайте първата **MSF** инструкция. Пуснете програмата и сравнете резулатат с предишния.

15. Нека стане оше по интересно. Орният код ще извърши цикъла 5 пъти и това число е фиксирано. Може да подадем числото като параметър на под-рутината (започваща с бележка L2). Ще използваме **PSH** и **POP**. Пуснете следното:

**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. обяснете как подаването на параметър работи:

17. модифицирайте кода по горе така че да се подава втори параметър на под-рутината както първият. Вторият параметър се използва за даване на стойност на R01. Запишете кода:

**Приложение 1 – Инструкции за симулатора**

|  |  |
| --- | --- |
| **Инстр.** | **Описание** |
| **Инструкции за обмен на информация** | |
| MOV | Местене на инф. в регистър; местене на регистър в регистър.  **MOV #2, R01** мести числото 2 в R01  **MOV R01, R03 мести съдържанието на** R01 в R03 |
| LDB | Зарежда байт памет от паметта в регистър.  **LDB 1022, R03** зарежда байт от адрес 1022 в регистър R03  **LDB** [**@R0**](mailto:@R02)**2, R05** зарежда байт от паметта от адреса на който е R02 |
| LDW | Зарежда дума (2 bytes)от пметта в регистъра  Също като LDB но дума (i.e. 2 bytes) се зарежда в регистъра. |
| STB | Зарежда байт от регистър в паметта  **STB R07, 2146** зарежда байт от R07 в адрес 2146 от паметта  **STB R04,** [**@R0**](mailto:@R08)**8** зарежда байт от R04 в адреса който отговаря на регистър R08  R08 |
| STW | Слага дума (2 bytes) от регистър в паметта  Същото като STB но зарежда дума в паметта |
| PSH | Слага данни най-отгоре в хардуерния стек (TOS); вкарва регистър в TOS  **PSH #6** вкарва в стека числото 6  **PSH R03** вкарва стойността на регистър R03 в стека |
| POP | Вади информация от стека в регистър  **POP R05** вади съдържанието от стека в регистър R05  **Забележка**: ако опитате POP от празен стек ще получите съобщение за грешка:  “Stack underflow”. |
| **Аритметични операции** | |
| ADD | Добавя число в регисър; добавя регистър в регистър  **ADD #3, R02** добавя 3 към стойността на регистър R02 и запазва полученото в регистър R02.  **ADD R00, R01** добавя съдържанието на R00 към R01 и запазва полученото в регистър R01. |
| SUB | Изважда число в регисър; изважда регистър в регистър |
| MUL | умножава число в регисър; умножава регистър в регистър |
| DIV | дели число в регисър; дели регистър в регистър |
| **Контролиране на обменни операции** | |
| JMP | „Скача“ на инструкция без условие да е изпълнено  **JMP 100** прескача на адрес 100 където има друга инструкция |

|  |  |
| --- | --- |
| JLT | Прескача на инструкция ако е по малко(след последното сравнение) |
| JGT | Прескача на инструкция ако е по голямо(след последното сравнение) |
| JEQ | Прескача на инструкция ако е равно(след последното равнение)  **JEQ 200** прескача на адрес 200 ако миналата инструкция показва че две числа са равни. **Z** флага става маркиран |
| JNE | Прескача на инструкция ако е не е равно(след последното сравнение) |
| MSF | Mark Stack Frame инструкцията се използва в комбинация с CAL  инструкция  **MSF** запазва място за връщания адрес от програмния стек **CAL 1456** запазва върнатия адрес в резервираното място и се премества на подчастта с адрес 1456 |
| CAL | Прескача на подчастен адрес (запазва адрес за връщане в стека)Тази инструкция се използва заедно с **MSF** . Вижте примера по горе |
| RET | Връща се от подчаст (използва вече запаметения в стека адрес за връщане) |
| SWI | Софтуерно прекъсване (използва се за помощ от ОС) |
| HLT | Симулация за задържане |
| **Сравняващи инструкции** | |
| CMP | Сравнява число с регистър;сравнява рег. с рег.  **CMP #5, R02** сравнява 5 със съдържанието на R02  **CMP R01, R03** сравнява съдържанията на R01 и R03  Бележки:  Ако R01 = R03 то флага **Z** ще е маркиран.  Ако R01 < R03 то нито един флаг няма да е селектиран.  Ако R01 > R03 то флага **N** ще е маркиран. |
| **Входни/изходни инструкции** | |
| IN | Взима входна информация (ако може) от външно В/И устройство |
| OUT | Извежда информация на В/И устройство  **OUT 16, 0** изкарва информацията от адрес 16 в конзолата (втория параметър трябва да е винаги 0) |