**A. Въведение**

**Програмен модел 3**

**Цели на заданието**

В края на лабораторното упражнение трябва да можете да:

 създавате условни инструкции към Процесора (compare и jump)

 създавате итеративни цикли от инструкции

 използвате недиректно адресиране за достъп до паметта

Това упражнение е направено за да покаже основите за работа с набор от CPU инструкции и програмирането на процесора.

**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

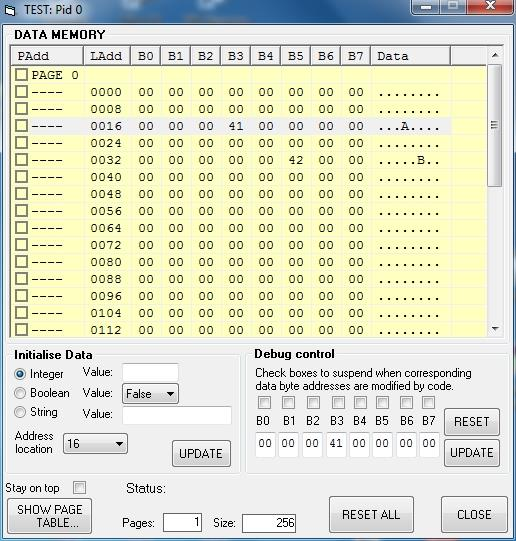
 Различни начини за адресиране на информация чрез инструкции

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. Симулаторът в детайли**

Може да видите настройките и детайлите за работа на симулатора в ръководство **Програмен модел 1** . Информацията не е повторена тук с изключение на допълнения как да видите и достъпите прегледа на паметта на програмата; с това ръководство ще достъпите тази част на симулатора. Подробности ще намерите на следващата страница.

**Изглед от паметта на програмата (data memory)**



**Снимка 1**

Инструкциите към CPU които достъпват частта от паметта, съдържаща информация, могат да четат или пишат информация в адресираните пространства. Тази информация може да бъде видяна в страниците на паметта, показани на **снимка 1**. Може да отворите този прозорец с натискане на бутона **SHOW PROGRAM DATA MEMORY…** в главния прозорец на CPU. Колонта **Ladd** (logical address) показва началния адрес на всяка линия(ред) на дисплея. Всеки ред представлява 8 байта информация. Колони от **B0** до **B7** са байтове от 0 до 7 за всеки ред. **Data** колоната показва реалните знаци отговарящи на 8-те байта. Тези байтове, отговарящи на знаци които не се показват са изобразени с точка (***.***). Байтовета с информация се показват шеснайсетично. Например на **снимка 1** на позиции 19 и 37 има ненулеви полета. Те отговарят за показване на буквите A и B.

За смяна на стойностите на който и да е байт първо изберете реда с байтовете който искате да променяте. След това в полето **Initialize Data** променете стойността на избраните байтове като цяло число, двоично или знаково (**Integer**, **Boolean** or **String** ). Натиснете бутона **UPDATE** за да направите промяната.

**E. Лабораторно упражнение – Изследвне и Откриване**

**Запишете инструкциите които вкарвате за да отговорите на въпросите в празните полета. Търсете в ПРИЛОЖЕНИЕ 1 в края на документа за подробности по желаните инструкции. Очаква се да изпълните инструкциите си на симулатора за да потвърдите истинността на отговорите си. При проблем се обръщайте към асистента.**

**A. Цикли иползващи jump и compare инструкции:**

1. Напишете условие, такова, че ако R02 > R01 то R03 се задава стойност 8. (Използвайте R01 като първа операнда R02 като втора)

2. Напишете условие, такова, че ако R02 <= R01 то R03 се задава стойност -5. (Използвайте R01 като първа операнда R02 като втора)

3. Напишете условие, такова, че ако R01 = 0 то R03 приема стойност 5 иначеR03 приема стойност R01 + 1.

4. Напишете цикъл, който се върти **5** пъти в който R02 започва от 2 .

5. Напишете цикъл, който се опвтаря докатоR04 > 0. Инициализирайте R04 на 8 за начална стойност.

6. Напишете цикъл,който се повтаря докатоR05 > R09.Инициализирайте R05 = 0 и R09 = 12.

7. Напишете процедура, която вкарва 8 и 2 най отгоре в стека. След това вади двете числа едно по едно от стека, събира ги и ги пъха обратно най-отгоре в стека.

8. Задача #1: Сложете 15 числа от 1 до 15 в стека използвайки команда **push** в цикъл. След това в друг цикъл използвайте **pop** за да вадите 2 числа от стека, съберете ги (**add)** и пъхнете **(push)** резултата в стека. Втория цикъл да се повтаря докато остане само 1 число в стека – сбора на всички останали.

**B. Инструкции за четене от и писане във паметта (RAM):**

**Следващите инструкции дават достъп до паметта на програмата. Може да се визуализира информацията от паметта зад а се изследват резултатите, следвайки примера от снимка 1 и описанието от секция D.**

9. Намерете инструкцията, която поставя (**store**) байт в паметта и поставете числото 65 в адресно пространство 20 (това използва метод за директно адресиране на паметта).

10. Преместете числото 51 в регистър R04. Поставете съдържанието на R04 в поле 21 (използва се директно регистърно поставяне в паметта).

11. Числото 22 преместете в R04. Използвайте научената информация за да сложите 58 в паметта индиректно (подсказка: използвайте ‘@’ – вижте инструкцията в приложението на края на документа) - (това е недиректен адресиращ метод).

12. Нмаерете инструкцията, която зарежда байт от паметта в регистър. Използвайте я за да заредите в паметта числото 22 в регистър R10.

**C. Свързване:**

13. Задача #2: напишете цикъл, в който 10 числа от 48 до 57 се поставят като единични байтове в паметта започвайки от адрес 24. Използвайте недиректно адресиране за поставяне на числата в паметта (точка 11 по горе).

14. Задача #3: напишете цикъл, в който числата поствени в паметта от 13 се копират в друга част от паметта започвайки от адрес 80.

**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 |

**7 |** P a g e

|  |  |
| --- | --- |
| 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) |

**8 |** P a g e