**Documentation**

1. **Problem Statement:**

A data analyst Mr. Ashok is working on information collected by a mobile service provider ByTel regarding monthly usage of data. The information collected is in the form of a Matrix (X axis - Total data downloaded in GB vs Y axis - Month of the year). He plans to use the Kibana (data visualization) tool to plot the data in charts. The tool prefers Months as X axis and data downloaded as Y axis. Hence he requires you to incorporate the following using CPU OS simulator based assembly language program:

* Accept the data collected in the form of a Matrix (X axis - Total data downloaded in GB vs Y axis - Month of the year) from the keyboard.
* Transpose the matrix.
* Display the transposed matrix on console.

1. **Team**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Name** | **ID NO** |
| **1** | Anirudh Menon | 2018AB04132 |
| **2** | Amit Kumar | 2018AB04046 |
| **3** | Vengal O.G.T. | 2018AB04020 |

1. **Description of solution provided:**

A matrix called ‘A’ can be of order m x n, represented as Amxn. Here m is the number or rows and n is the number of columns in the matrix. WhenAmxn is transposed, it will be of the format of Bnxm.

The input data was accepted directly into data memory of CPU-OS simulator. It was then displayed on the output console as Input Matrix. It was then transposed using assembly language operations, stored into memory and then displayed on the console of CPU-OS Simulator as Output Matrix.

### **3.1 Design:**

Multiple design approaches were evaluated to consider the matrix transpose along with various pros and cons. They are as follows:

**Approach 1**: A simple transpose of the A[i,j] to A[j,i] as typically programmed in a high level programming language like C or Java.

Problems with the approach: This approach was not possible since multidimensional array representation was not possible in the given CPU OS Simulator. Various options including coding in CPU-OS programming language in the compiler and running on the simulator was tried but did not yield good results during our research and experimentation.

**Approach-2:** Use of stack push and pop operations for storing and then moving it to transpose. This approach was more time consuming and high stack usage, since the writing logic was found to be first calculated and was considered complex.

**Approach-3 :** Store the value of the initial matrix in a single array or address location incrementing from A11, A12, A13…., A1n, A21, A22,………….. A2n,………………… Am1,………..until Amn. The stored values are then looped through to find the rows and columns and then interchanged and stored into target address location in the order of transposed matrix. That means, the first column is picked as the first row in the target address.

Approach-3 has been selected for implementation in CPU OS assignment solution.

There were three parts of the assembly code implementation. They are as follows:

1. Reading Source Data in matrix format identifying the row and column positions.
2. Transposing logic of the matrix and storing into target data memory.
3. Printing the final transpose data from the memory to console.

|  |  |  |
| --- | --- | --- |
| ***Sl No*** | ***Name of the data*** | ***Addressing mode used*** |
| *1* | *Read - Input size of matrix from 0000 and 0008 for values of row size (m) and column size (n) respectively.* | *Memory Direct Addressing* |
| *2* | *Read -Input matrix source data stored in memory starting at location 0016 and to the location of 0016+(m\*n)\*8* | *Memory Direct addressing*  *Memory InDirect Addressing*  *Register Direct Addressing Mode*  *Register Indirect Addressing Mode*  *Indexed Addressing Mode,*  *Base Register Addressing Mode* |
| *3* | *Write transposed matrix data from registers to memory in incremental manner by every word to location of 0120+(n\*m)\*8* | *Memory Direct Addressing,*  *Memory Indirect Addressing*  *Register Direct Addressing Mode*  *Register Indirect Addressing Mode*  *Index Addressing Mode,*  *Base Register Addressing Mode* |
| *4* | *Read Strings to be displayed as tags “Input Matrix” and “Output Matrix”* | *Memory Direct Addressing* |
|  |  |  |
| ***Sl No*** | ***Name of the function*** | ***Description*** |
| *1* | *Input\_matrix* | *This function is used to read Source Data in matrix format identifying the row and column positions.* |
| *2* | *Print\_Input\_matrix* | *This function is used to Print the input data from the memory to console in matrix format.* |
| *3* | *Transpose\_matrix* | *This function has transposing logic of the matrix and storing transposed matrix data into target data memory location*. |
| *4* | *Print\_transpose\_matrix* | *This function is used to Printing the final transpose data from the memory to console in matrix format.* |

1. **Procedure:**
   1. **Data Input:**

The input data is directly entered into specific Data memory address location.

Here are the details of the address location for a 3x4 matrix:

|  |  |  |
| --- | --- | --- |
| **LAddress #** | **Data Stored** | **Sample Value** |
| 0000 | m = Number of Rows of source matrix | 3 |
| 0008 | n = Number of Columns of source Matrix | 4 |
| 0016 | A11 | 11 |
| 0024 | A12 | 12 |
| 0032 | A13 | 13 |
| 0040 | A14 | 14 |
| 0048 | A21 | 15 |
| 0056 | A22 | 16 |
| 0064 | A23 | 17 |
| 0072 | A24 | 18 |
| 0080 | A31 | 19 |
| 0088 | A32 | 20 |
| 0096 | A33 | 21 |
| 0104 | A34 | 22 |
| 0224 | String for display | Input Matrix: |
| 0240 | String for display | Output Matrix: |

Table 1: Input Values

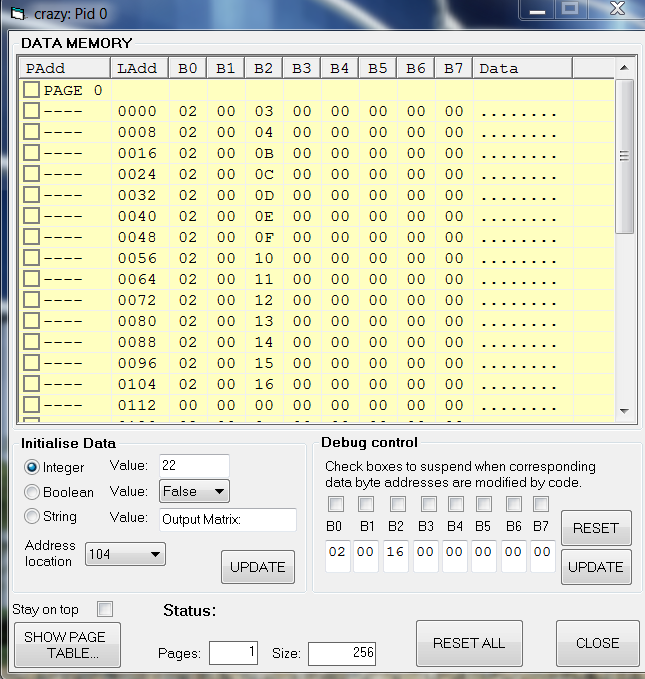


Fig 1a: Screenshot of Data Memory (Input –Matrix)

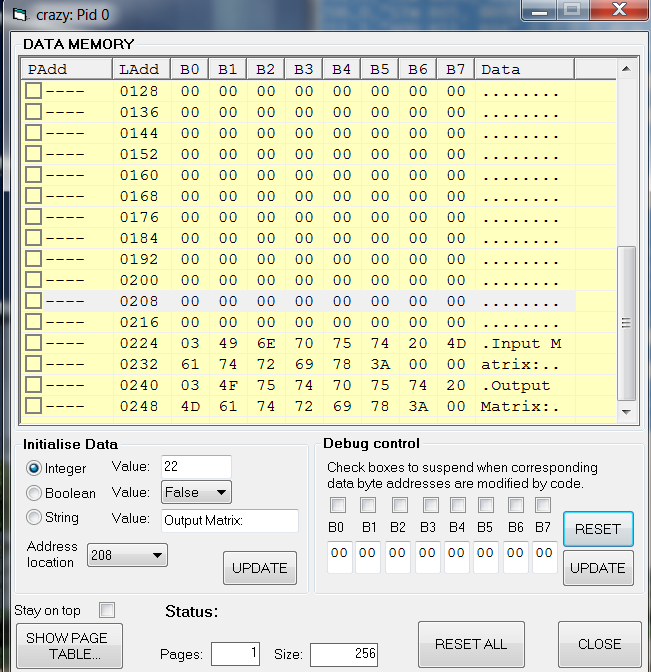


Fig 1b. Input of Strings for display

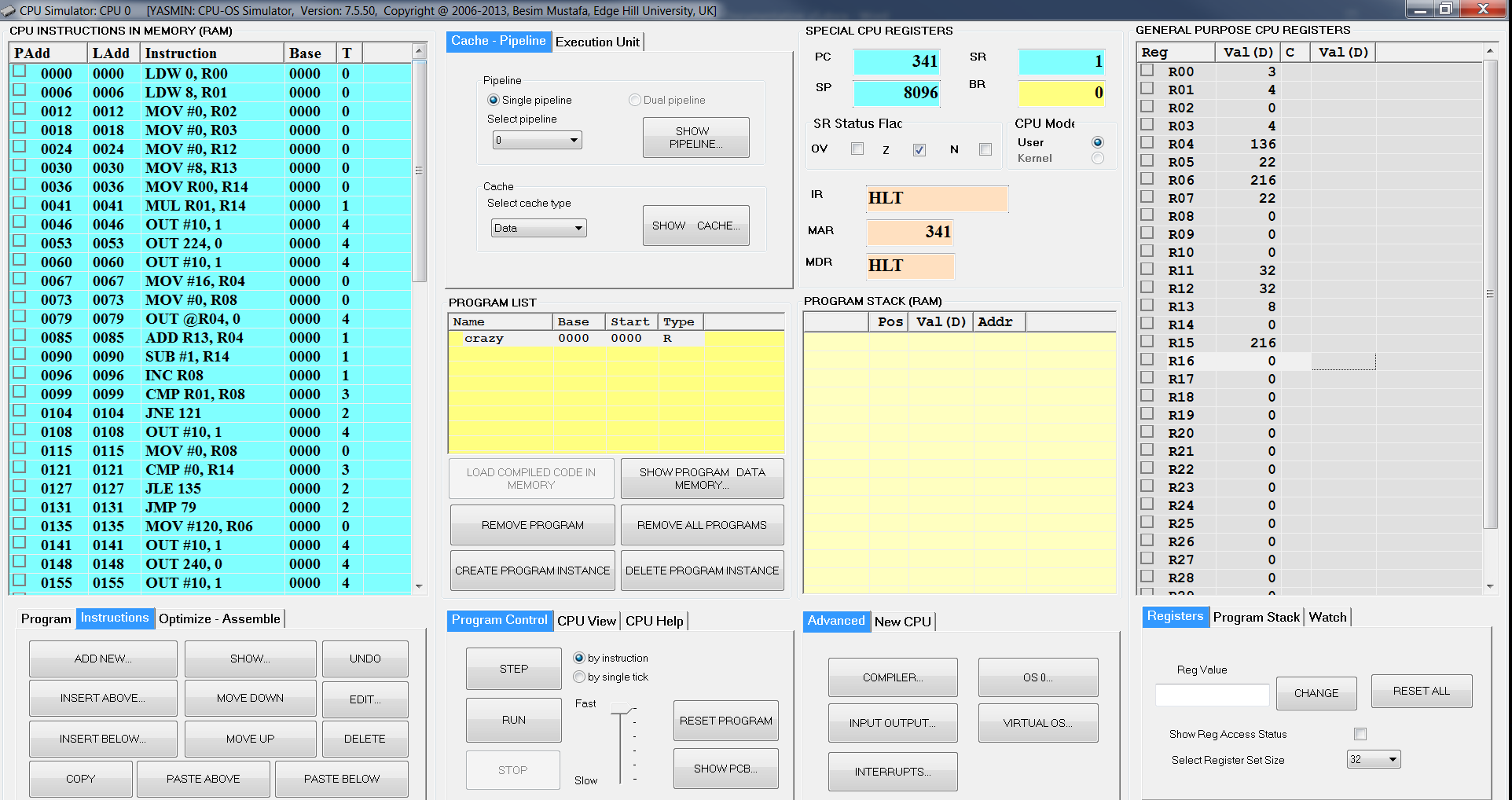
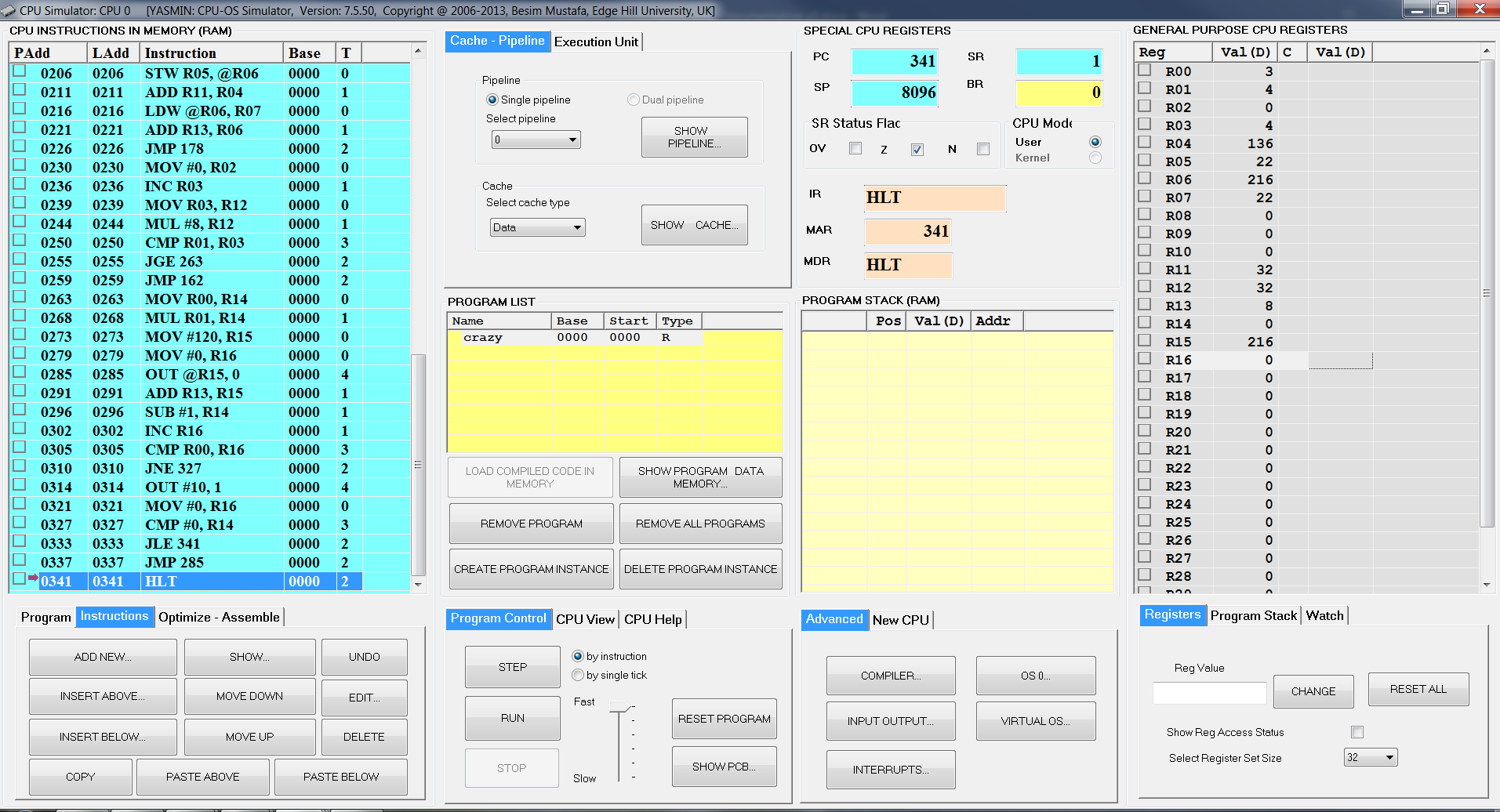


Fig 2: Screenshot of the Input Program – Part 1

Fig 2: Screenshot of the Input Program – Part 2

**Expected Output Values:**

The expected details of the address location for a 4x3 matrix that is transposed to Matrix B:

|  |  |  |
| --- | --- | --- |
| **LAddress #** | **Data Stored** | **Expected Value** |
| 0120 | B11 | 11 |
| 0128 | B12 | 15 |
| 0136 | B13 | 19 |
| 0144 | B21 | 12 |
| 0152 | B22 | 16 |
| 0160 | B23 | 20 |
| 0168 | B31 | 13 |
| 0176 | B32 | 17 |
| 0184 | B33 | 21 |
| 0192 | B41 | 14 |
| 0200 | B42 | 18 |
| 0208 | B43 | 22 |

Table 2: Expected Output Values

**4.2 Program**

|  |
| --- |
| * 0,0,"LDW 0, R00",0,0,0,0,#FALSE# * 6,0,"LDW 8, R01",0,0,0,0,#FALSE# * 12,0,"MOV #0, R02",0,0,0,0,#FALSE# * 18,0,"MOV #0, R03",0,0,0,0,#FALSE# * 24,0,"MOV #0, R12",0,0,0,0,#FALSE# * 30,0,"MOV #8, R13",0,0,0,0,#FALSE# * 36,0,"MOV R00, R14",0,0,0,0,#FALSE# * 41,1,"MUL R01, R14",0,0,0,0,#FALSE# * 46,4,"OUT #10, 1",0,0,0,0,#FALSE# * 53,4,"OUT 224, 0",0,0,0,0,#FALSE# * 60,4,"OUT #10, 1",0,0,0,0,#FALSE# * 67,0,"MOV #16, R04",0,0,0,0,#FALSE# * 73,0,"MOV #0, R08",0,0,0,0,#FALSE# * 79,4,"OUT @R04, 0",0,0,0,0,#FALSE# * 85,1,"ADD R13, R04",0,0,0,0,#FALSE# * 90,1,"SUB #1, R14",0,0,0,0,#FALSE# * 96,1,"INC R08",0,0,0,0,#FALSE# * 99,3,"CMP R01, R08",0,0,0,0,#FALSE# * 104,2,"JNE 121",0,0,0,0,#FALSE# * 108,4,"OUT #10, 1",0,0,0,0,#FALSE# * 115,0,"MOV #0, R08",0,0,0,0,#FALSE# * 121,3,"CMP #0, R14",0,0,0,0,#FALSE# * 127,2,"JLE 135",0,0,0,0,#FALSE# * 131,2,"JMP 79",0,0,0,0,#FALSE# * 135,0,"MOV #120, R06",0,0,0,0,#FALSE# * 141,4,"OUT #10, 1",0,0,0,0,#FALSE# * 148,4,"OUT 240, 0",0,0,0,0,#FALSE# * 155,0,"MOV #16, R04",0,0,0,0,#FALSE# * 161,4,"OUT #10, 1",0,0,0,0,#FALSE# * 168,1,"ADD R04, R12",0,0,0,0,#FALSE# * 173,0,"MOV R12, R04",0,0,0,0,#FALSE# * 178,3,"CMP R00, R02",0,0,0,0,#FALSE# * 183,2,"JGE 230",0,0,0,0,#FALSE# * 187,0,"MOV R01, R11",0,0,0,0,#FALSE# * 192,1,"MUL #8, R11",0,0,0,0,#FALSE# * 198,1,"INC R02",0,0,0,0,#FALSE# * 201,0,"LDW @R04, R05",0,0,0,0,#FALSE# * 206,0,"STW R05, @R06",0,0,0,0,#FALSE# * 211,1,"ADD R11, R04",0,0,0,0,#FALSE# * 216,0,"LDW @R06, R07",0,0,0,0,#FALSE# * 221,1,"ADD R13, R06",0,0,0,0,#FALSE# * 226,2,"JMP 178",0,0,0,0,#FALSE# * 230,0,"MOV #0, R02",0,0,0,0,#FALSE# * 236,1,"INC R03",0,0,0,0,#FALSE# * 239,0,"MOV R03, R12",0,0,0,0,#FALSE# * 244,1,"MUL #8, R12",0,0,0,0,#FALSE# * 250,3,"CMP R01, R03",0,0,0,0,#FALSE# * 255,2,"JGE 263",0,0,0,0,#FALSE# * 259,2,"JMP 155",0,0,0,0,#FALSE# * 263,0,"MOV R00, R14",0,0,0,0,#FALSE# * 268,1,"MUL R01, R14",0,0,0,0,#FALSE# * 273,0,"MOV #120, R15",0,0,0,0,#FALSE# * 279,0,"MOV #0, R16",0,0,0,0,#FALSE# * 285,4,"OUT @R15, 0",0,0,0,0,#FALSE# * 291,1,"ADD R13, R15",0,0,0,0,#FALSE# * 296,1,"SUB #1, R14",0,0,0,0,#FALSE# * 302,1,"INC R16",0,0,0,0,#FALSE# * 305,3,"CMP R00, R16",0,0,0,0,#FALSE# * 310,2,"JNE 327",0,0,0,0,#FALSE# * 314,4,"OUT #10, 1",0,0,0,0,#FALSE# * 321,0,"MOV #0, R16",0,0,0,0,#FALSE# * 327,3,"CMP #0, R14",0,0,0,0,#FALSE# * 333,2,"JLE 341",0,0,0,0,#FALSE# * 337,2,"JMP 285",0,0,0,0,#FALSE# * 341,2,"HLT",0,0,0,0,#FALSE# |

### **4.2 Result:**

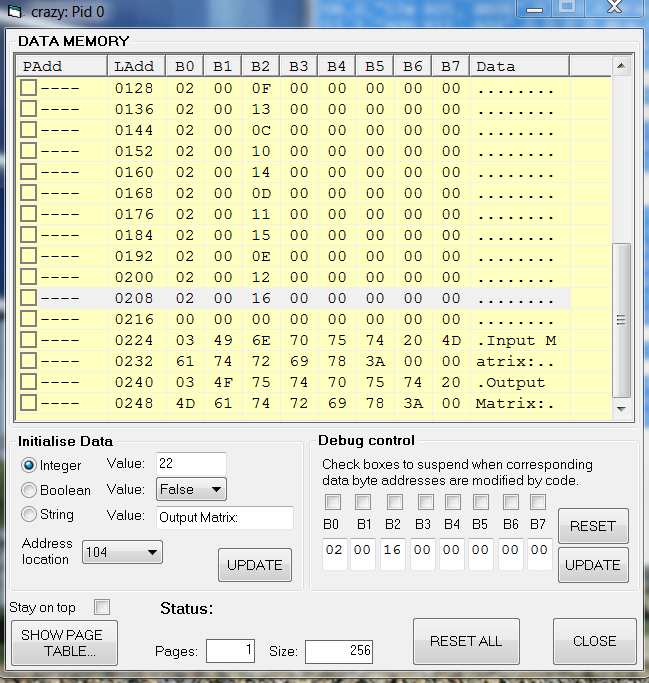


Fig 3: Screenshot of output memory location (from LAdd 120)  
The matrix is stored in the format of an array [B11, B12, B13, B21, B22, B23, B31, B32, B33, B41, B42, B43,]

The transposed matrix is displayed in the format as follows

B11, B12, B13

B21, B22, B23

B31, B32, B33

B41, B42, B43

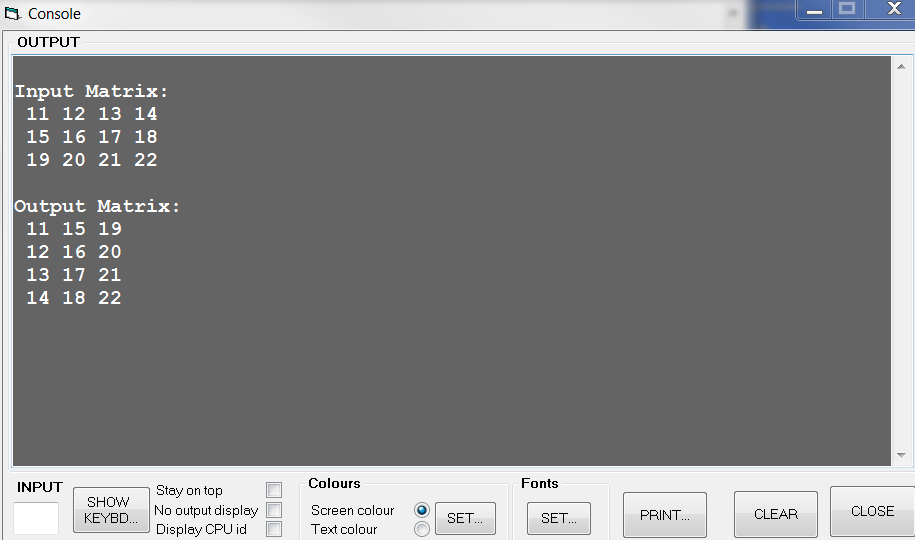


Fig 4: Screenshot of transposed matrix output

## 4.3 Observations and Workarounds:

1. **Observation:** CPU OS Simulator does not wait for the user to provide the input when in RUN mode. The usability limitations on entering data in interactive GUI for a series of data input while in RUN mode.
   1. **Potential cause:** The CPU simulator probably does not have input interrupt feature available
   2. **Workaround:**
      1. User has to enter first data in the input and then run on “STEP” mode ONLY. It is painful to run on STEP mode and then enter the data manually for every loop.
      2. Alternative solution of entering the data in the memory location directly is a simpler approach for larger data sets and when you want to execute in RUN Mode. This option was selected in our solution.
2. **Observation:** It is not possible to enter an integer bigger than one character, say “100” or “-1” in one single entry. Only single character entry is possible.
   1. **Potential cause:** This is a known limitation of CPU-OS Simulator.
   2. **Workaround:** User need to enter individual character as input and then convert into integer from ASCII and then keep multiplying the digits in individual locations to get the actual integer.
3. **Observation:** It has been found that sometimes the CPU-OS Simulator gives an error “Data Descriptor Error: Expected xx, read yy” when in program is executed in continuous “RUN” mode instead of “STEP” by instruction mode.
   1. **Potential cause:** This seems to be bug in the simulator.
   2. **Workaround:** If this is error occurs, the work around would be to restart in “STEP” mode either from where it was last throwing error or from

## **4.4 Conclusions:**

We have been successively computed and tested matrices of order 12x1, 1x12, 2x2, 4x2, 4x3, 3x3 and we believe that it can scale to ‘nxn’ provided there are sufficiently enough memory space in the device/simulator.

1. **Contribution**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl. No.** | **Name** | **ID NO** | **Contribution** |
| 01 | Anirudh Menon | 2018AB04132 | Research, Design, Test, Debug |
| 02 | Amit Kumar | 2018AB04046 | Design, Documentation |
| 03 | Vengal O.G.T. | 2018AB04020 | Research, Design, Debug, Implementation, Test, Documentation |