PONDICHERRY UNIVERSITY

Kalapet, Puducherry, 605014

SCHOOL OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CSSC 415 ACA LAB

RECORD BOOK

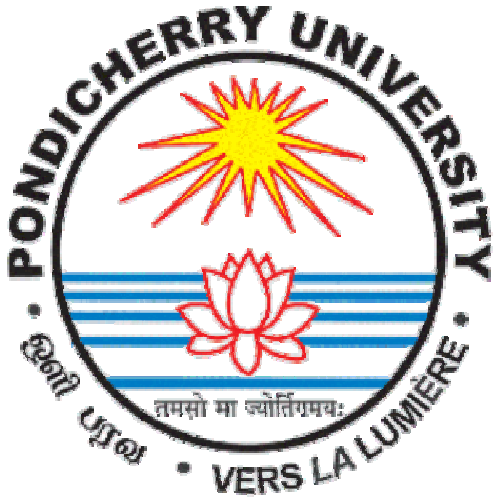
NAME :

REGISTER NUMBER :

COURSE : M.SC FIRST YEAR

DEPARTMENT : COMPUTER SCIENCE

SEMESTER : 1st



DEPARTMENT OF COMPUTER SCIENCE

PONDICHERRY UNIVERSITY

PUDUCHERRY-605014

BONAFIDE CERTIFICATE

Certified that this is a bonafide record of practical work done by

Ms. /Mr. …….. with register number ……. of First Semester Master of computer science Program in the subject of **CSSC 415 ACA-LAB** during the academic year 2020-2021.

In charge:

Date:

Submitted for M.Sc. Degree Practical examination held on ……
at the department of Computer Science, Pondicherry University, Puducherry.

Date:

Examiners:

Internal Examiner **External Examiner**

TABLE OF CONTENT

DATE: 21.12.2020	**SIMULATION OF COMPUTER COMPONENTS**
EX.NO: 01	

AIM:

Write a program for Simulation of Computer Components.

SOFTWARE USED:

MARS 4.5

THEORY:

MARS is an IDE for MIPS Assembly Language Programming. MARS is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use.

MARS features

- GUI with point-and-click control and integrated editor
- Easily editable register and memory values, similar to a spreadsheet
- Display values in hexadecimal or decimal
- Command line mode for instructors to test and evaluate many programs easily
- Floating point registers, coprocessor1 and coprocessor2. *St*andard tool: bit-level view and edit of 32-bit floating point registers.
- Variable-speed single-step execution
- "Tool" utility for MIPS control of simulated devices. *S*tandard tool: Cache performance analysis.
- Single-step backwards

The Mars program is a combined assembly language editor, assembler, simulator, and debugger for the MIPS processor. It was developed by Pete Sanderson and Kenneth Vollmar at Missouri State University.

Mars Basic Operation

Programming in assembly language involves the following activities.

- creating new program files
- opening old program files
- editing programs
- assembling programs
- running programs
- saving programs

Usually, the last five activities are repeated several times.

Using the MARS simulator provides some great benefits who wish to learn how assembly is written. It provides a powerful tool for debugging, and "watching" how code is working. We can see the true assembly code that is generated by their MAL commands.

Not all MAL commands are used "get, put, done, start, etc" however, this is not a limitation, as synthesizing TAL commands are part of the learning experience. While the current solution is being used is functional. There is no easy way to debug code in a text editor. At the very least, MARS can be used to generate the code for use on the SPIM/SAL simulators in use now. I have verified that code written with does compile and run perfectly on the SPIM/SAL sim.

PROCEDURE:

Step 1: Start MARS.

Step 2: Use File → New or Click on Create new file icon.

Step 3: Generate the code in the editor.

Step 4: Assemble the program. The provided assembly program is complete.

Step 5: Execute the program.

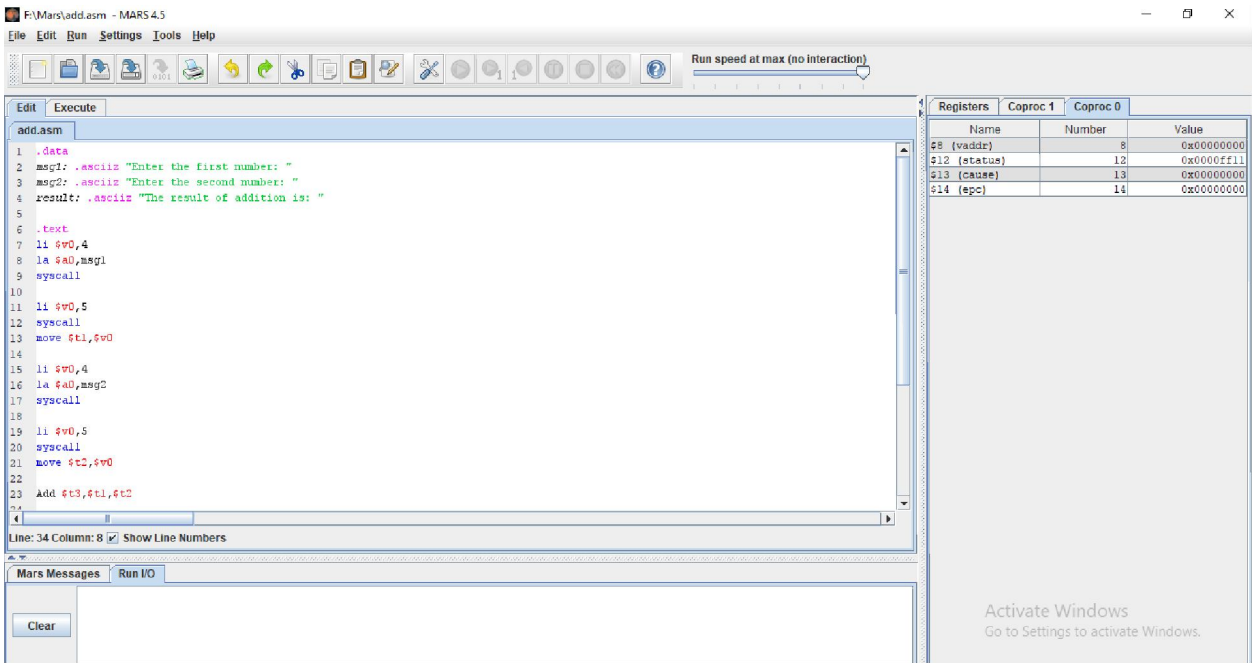
Step 6: Stop.

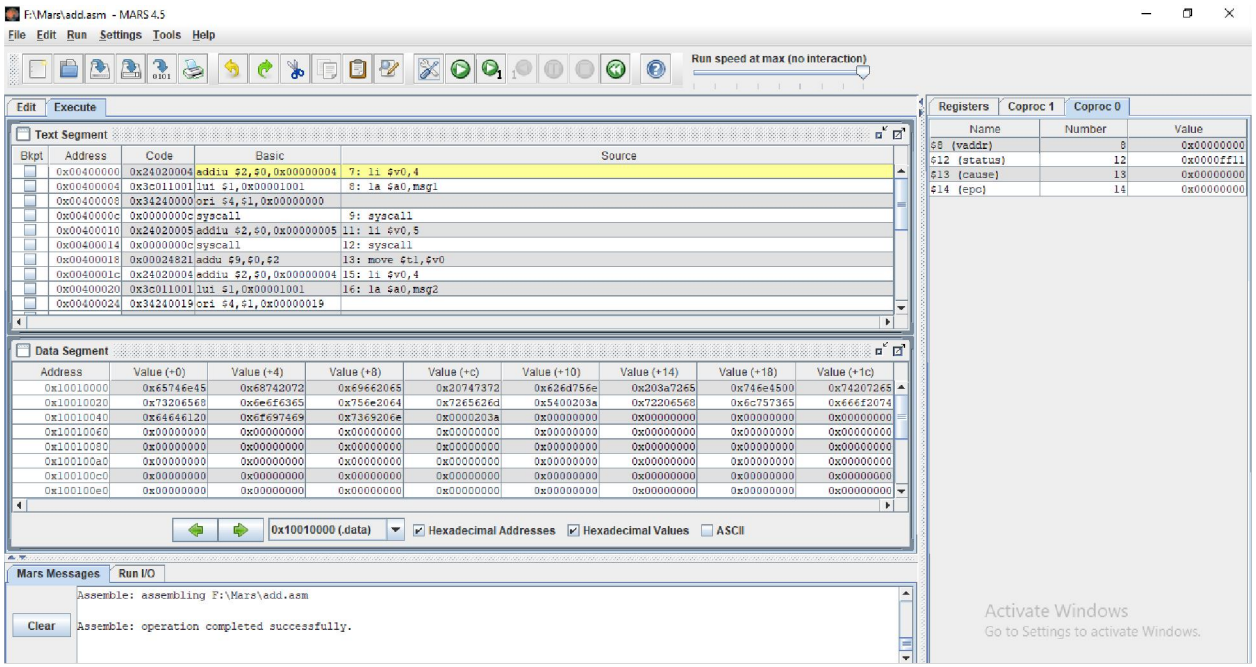
CODE:

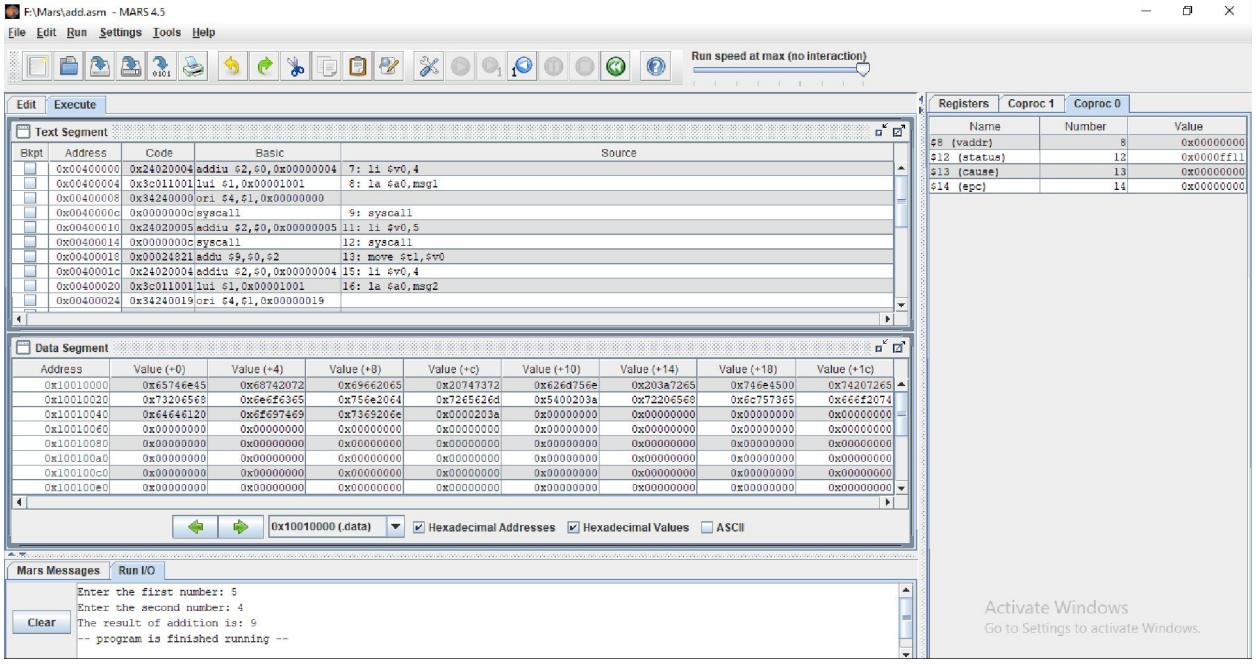
```
.data
msg1: .asciiz "Enter the first number: "
msg2: .asciiz "Enter the second number: "
result: .asciiz "The result of addition is: "
.text
li $v0,4
la $a0,msg1
syscall
li $v0,5
syscall
move $t1,$v0
li $v0,4
la $a0,msg2
syscall
li $v0,5
syscall
move $t2,$v0
Add $t3,$t1,$t2
li $v0,4
la $a0,result
```

```
syscall

li $v0,1

move $a0,$t3

syscall

li $v0,10

syscall
```

OUTPUT:







RESULT:

Thus the above simulation was executed successfully.

DATE: 28.12.2020	SIMULATION OF PIPELINE
EX.NO: 02	

AIM:

Write a program for Simulation of Pipeline.

SOFTWARE USED:

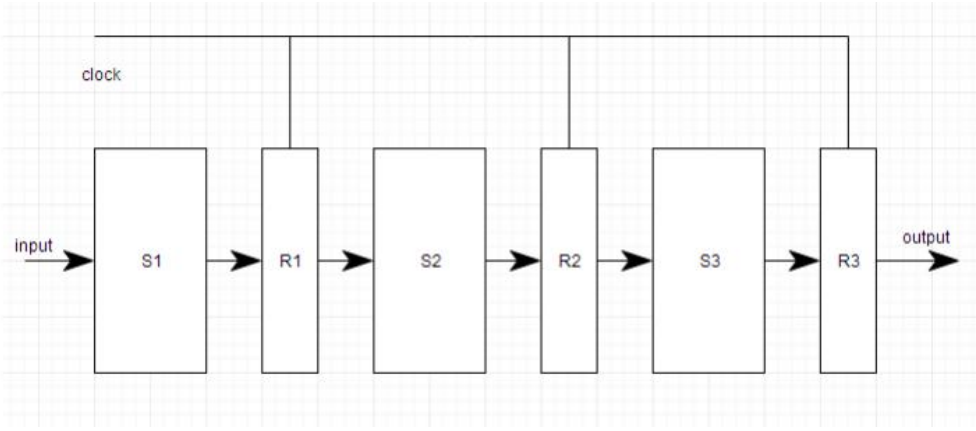
MARS 4.5

THEORY:

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as pipeline processing. Pipelining is a technique where multiple instructions are overlapped during execution.

Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end. Pipelining increases the overall instruction throughput.

In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

Types of Pipeline:

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc.

The floating point addition and subtraction is done in 4 parts:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract mantissas
4. Produce the result.

Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline

In this a stream of instructions can be executed by overlapping fetch, decode and execute phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Pipeline Stages

RISC processor has 5 stage instruction pipelines to execute all the instructions in the RISC instruction set.

Following are the 5 stages of RISC pipeline with their respective operations:

- **Stage 1 (Instruction Fetch)**
  In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
- **Stage 2 (Instruction Decode)**
  In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
- **Stage 3 (Instruction Execute)**
  In this stage, ALU operations are performed.
- **Stage 4 (Memory Access)**
  In this stage, memory operands are read and written from/to the memory that is present in the instruction.
- **Stage 5 (Write Back)**
  In this stage, computed/fetched value is written back to the register present in the instructions.

PROCEDURE:

Step 1: Start MARS.

Step 2: Use File → New or Click on Create new file icon.

Step 3: Generate the code in the editor.

Step 4: Assemble the program. The provided assembly program is complete.

Step 6: Click Tool → MIPS X-RAY.

Step 7: In MIPS X-RAY – Animation of MIPS Datapath dialog box click Connect to MIPS.

Step 8: Run the simulator.

Step 9: The simulator shows IMMEDIATE TYPE INSTRUCTION.

Step 10: Re-run the simulator.

Step 11: The simulator shows LOAD TYPE INSTRUCTION.

Step 12: Re-run the simulator.

Step 13: The simulator shows REGISTER TYPE INSTRUCTION.

Step 14: Re-run the simulator.

Step 15: The simulator shows STORE TYPE INSTRUCTION.

Step 16: Stop.

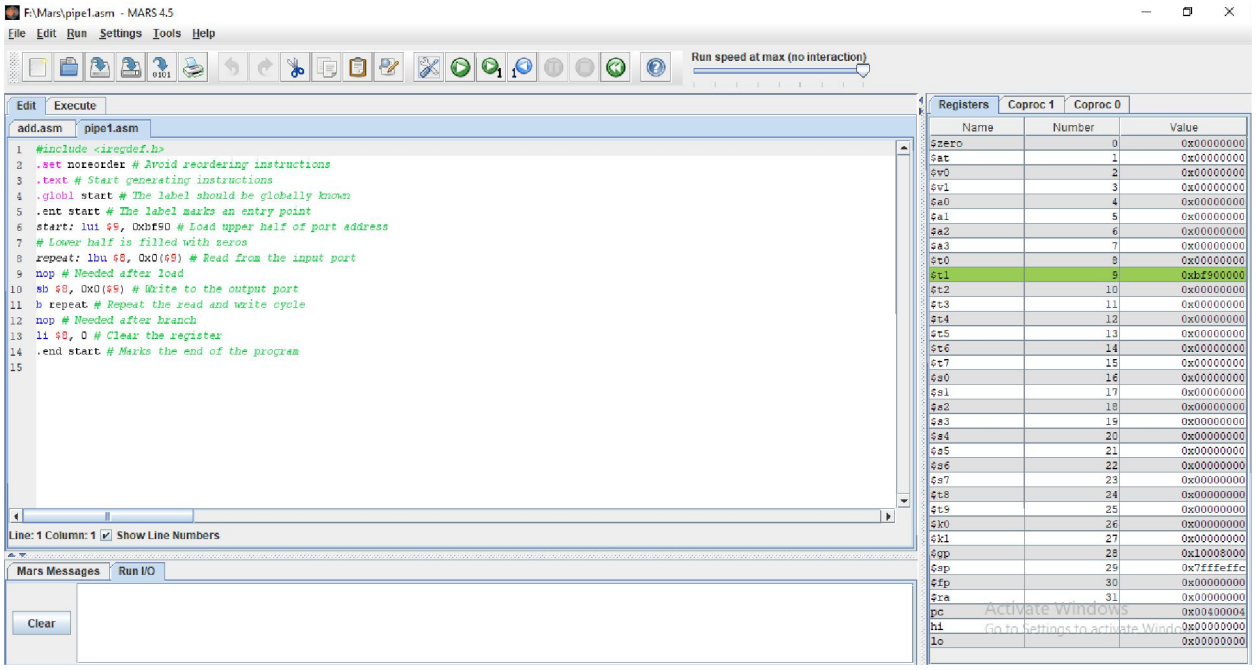
CODE:

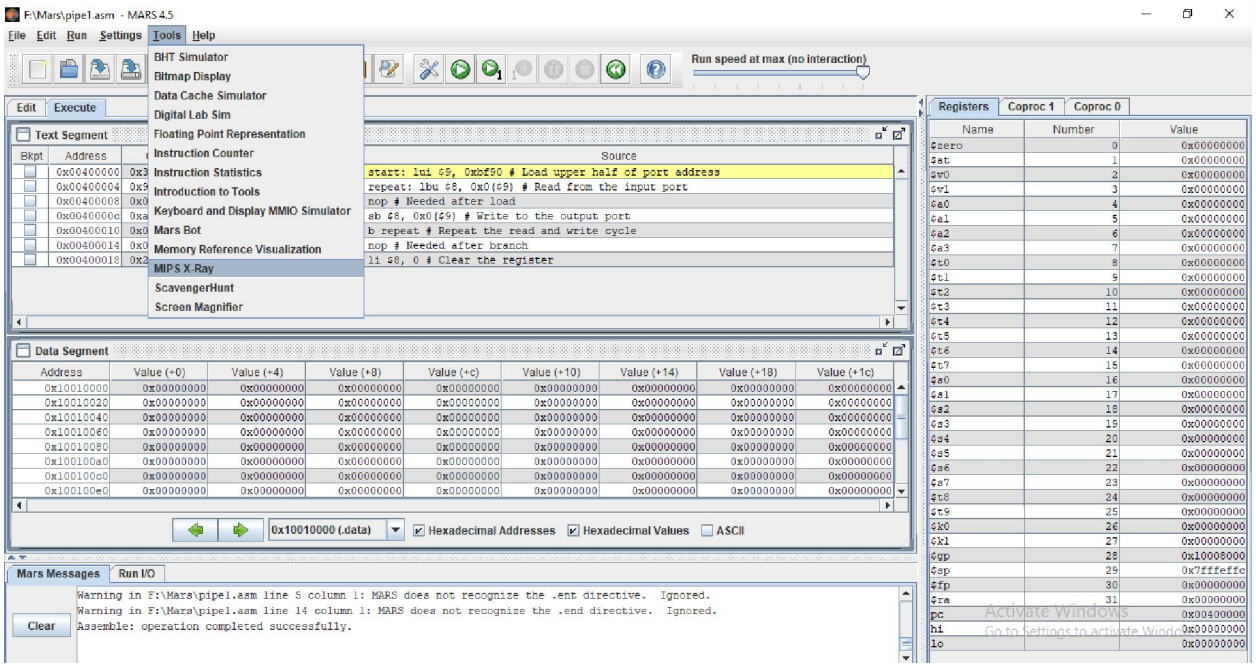
```
#include <iregdef.h>
.set noreorder                  # Avoid reordering instructions
.text                           # Start generating instructions
.globl start                    # The label should be globally known
.ent start                      # The label marks an entry point
start: lui $9, 0xbf90           # Load upper half of port address
# Lower half is filled with zeros
repeat: lbu $8, 0x0($9)         # Read from the input port
nop                             # Needed after load
sb $8, 0x0($9)                  # Write to the output port
b repeat                        # Repeat the read and write cycle
nop                             # Needed after branch
```

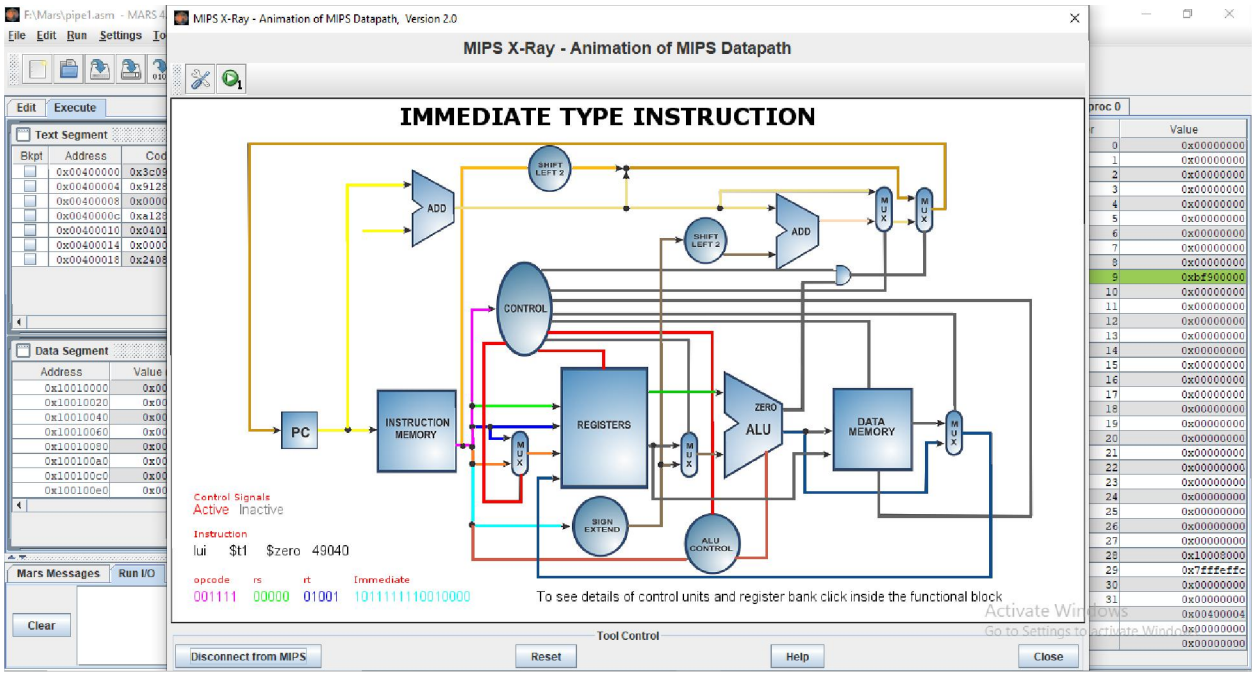
li $8, 0 # Clear the register

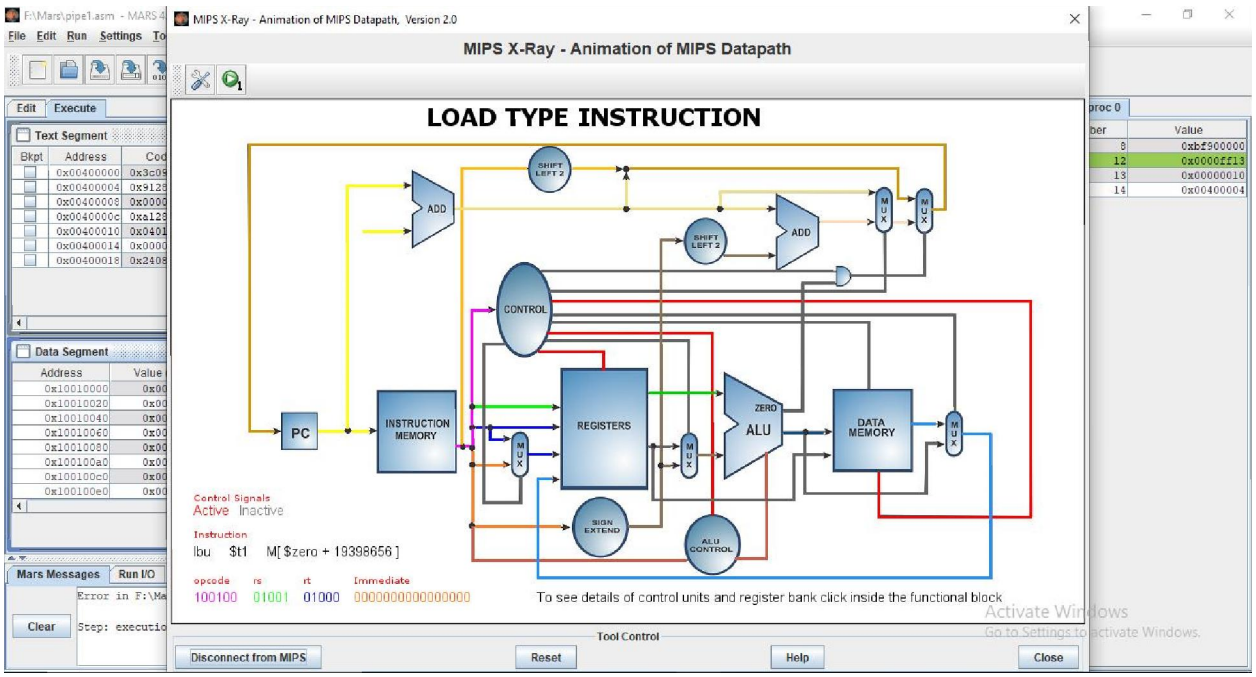
.end start # Marks the end of the program

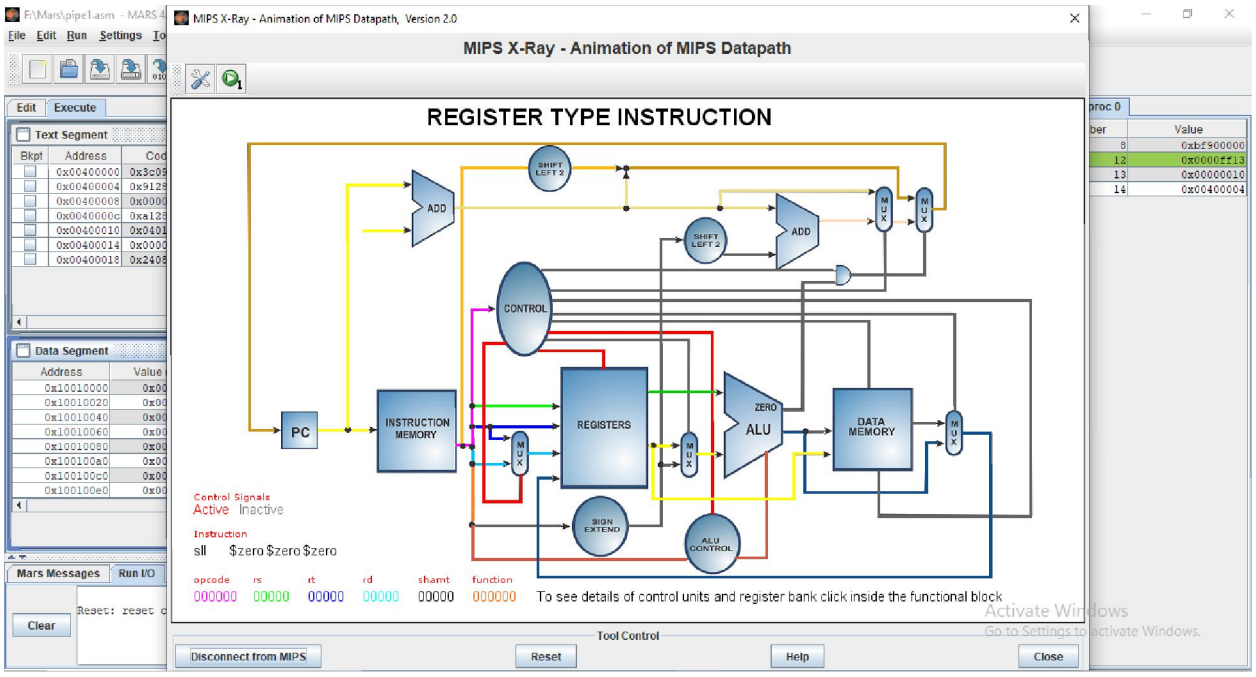
OUTPUT:

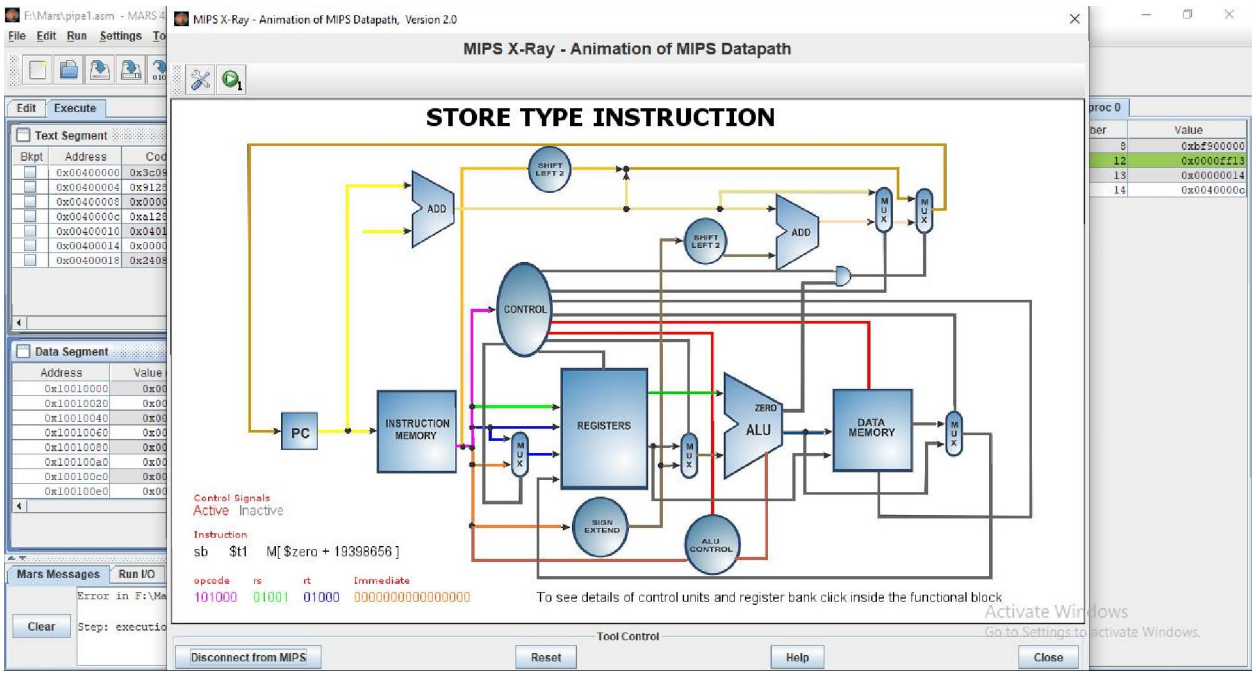





MIPS X-Ray - Animation of MIPS Datapath, Version 2.0
MIPS X-Ray - Animation of MIPS Datapath
IMMEDIATE TYPE INSTRUCTION
ADD
SHIFT LEFT 2
ADD
MUX
MUX
CONTROL
PC
INSTRUCTION MEMORY
REGISTERS
ZERO
ALU
DATA MEMORY
MUX
MUX
MUX
SIGN EXTEND
ALU CONTROL
Control Signals
Active Inactive
Instruction
lui $t1 $zero 49040
opcode rs rt Immediate
001111 00000 01001 1011111110010000
To see details of control units and register bank click inside the functional block
Tool Control
Disconnect from MIPS
Reset
Help
Close
Mars Messages
Run I/O
Clear


MIPS X-Ray - Animation of MIPS Datapath, Version 2.0
MIPS X-Ray - Animation of MIPS Datapath
LOAD TYPE INSTRUCTION
ADD
SHIFT LEFT 2
ADD
MUX
MUX
CONTROL
PC
INSTRUCTION MEMORY
REGISTERS
ZERO
ALU
DATA MEMORY
MUX
MUX
MUX
SIGN EXTEND
ALU CONTROL
Control Signals
Active Inactive
Instruction
lbu $t1 M[$zero + 19398656]
opcode rs rt Immediate
100100 01001 01000 0000000000000000
To see details of control units and register bank click inside the functional block
Tool Control
Disconnect from MIPS
Reset
Help
Close
Mars Messages
Run I/O
Clear





RESULT:

Thus the above simulation was executed successfully.

DATE: 04.01.2021	**SIMULATION OF INSTRUCTION LEVEL PARALLELISM**
EX.NO: 03	

AIM:

Write a program for Simulation of Instruction Level Parallelism.

SOFTWARE USED:

MARS 4.5

THEORY:

Instruction-level Parallelism (ILP) is a family of processor and compiler design techniques that speed up execution by causing individual machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel.

The operations involved are normal RISC-style operations, and the system is handed a single program written with a sequential processor in mind. Thus an important feature of these techniques is that like circuit speed improvements, but unlike traditional multiprocessor parallelism and massive parallel processing, they are largely transparent to users.

VLIW sand super-scalars are examples of processors that derive their benefit from instruction-level parallelism, and software pipelining and trace scheduling are example software techniques that expose the parallelism that these processors can use.

Although small amounts of ILP has been present in the highest performance uniprocessors of the past 30 years, the 1980s saw it became a much more significant force in computer design.

Several systems were built, and sold commercially, which pushed ILP far beyond where it had been before, both in terms of the amount of ILP offered and in the central role ILP played in the design of the system.

By the early 1990s, advanced microprocessor design at all major CPU manufacturers incorporated ILP, and new techniques for ILP became a popular topic at academic conferences. With all of this activity, we felt that, in contrast to a report on suggested future techniques, there would be great value in gathering, in an archival reference, reports on experience with real ILP systems and reports on the measured potential of ILP. Thus this special issue of the Journal of Supercomputing.

A typical ILP processor has the same type of execution hardware as a normal RISC machine. The differences between a machine with ILP and one without is that there may be more of that hardware, for example several integer adders instead of just one, and that the control will allow, and possibly arrange, simultaneous access to whatever execution hardware is present.

Consider the execution hardware of a simplified ILP processor consisting of four functional units and a branch unit connected to a common register file. Typically ILP execution hardware allows multiple-cycle operations to be pipelined, so we may assume that a total of four operations can be initiated each cycle. If in each cycle the longest latency operation is issued, this hardware could have 10 operations "in flight" at once, which would give it a maximum possible speed-up of a factor of 10 over a sequential processor with similar execution hardware. This execution hardware resembles that of several VLIW processors that have been built and used commercially, though it is more limited in its amount of ILP. Several superscalar processors now being built also offer a similar amount of ILP.

There is a large amount of parallelism available even in this simple processor. The challenge is to make good use of it-we will see that with the technology available today, an ILP processor is unlikely to achieve nearly as much as a factor of 10 on many classes of programs, though scientific programs, and others, can yield far more than that on a processor which has more functional units.

Instruction Level Parallelism (ILP) is used to refer to the architecture in which multiple operations can be performed parallelly in a particular process, with its own set of resources – address space, registers, identifiers, state, program counters. It refers to the

compiler design techniques and processors designed to execute operations, like memory load and store, integer addition, float multiplication, in parallel to improve the performance of the processors. Examples of architectures that exploit ILP are VLIWs, Superscalar Architecture.

ILP processors have the same execution hardware as RISC processors. The machines without ILP have complex hardware which is hard to implement. A typical ILP allows multiple-cycle operations to be pipelined.

Architecture:

Instruction Level Parallelism is achieved when multiple operations are performed in single cycle, that is done by either executing them simultaneously or by utilizing gaps between two successive operations that is created due to the latencies.

Now, the decision of when to execute an operation depends largely on the compiler rather than hardware. However, extent of compiler's control depends on type of ILP architecture where information regarding parallelism given by compiler to hardware via program varies. The classification of ILP architectures can be done in the following ways –

1. **Sequential Architecture:**
   Here, program is not expected to explicitly convey any information regarding parallelism to hardware, like superscalar architecture.
2. **Dependence Architecture:**
   Here, program explicitly mentions information regarding dependencies between operations like dataflow architecture.
3. **Independence Architecture:**
   Here, program gives information regarding which operations are independent of each other so that they can be executed instead of the 'nop's.

In order to apply ILP, compiler and hardware must determine data dependencies, independent operations, and scheduling of these independent operations, assignment of functional unit, and register to store data.

PROCEDURE:

Step 1: Start MARS.

Step 2: Use File → New or Click on Create new file icon.

Step 3: Generate the code in the editor.

Step 4: Assemble the program. The provided assembly program is complete.

Step 6: Click Tool → MIPS X-RAY.

Step 7: In MIPS X-RAY – Animation of MIPS Datapath dialog box click Connect to MIPS.

Step 8: Run the simulator.

Step 9: The simulator shows LOAD TYPE INSTRUCTION.

Step 10: Re-run the simulator.

Step 11: The simulator shows IMMEDIATE TYPE INSTRUCTION.

Step 12: Re-run the simulator.

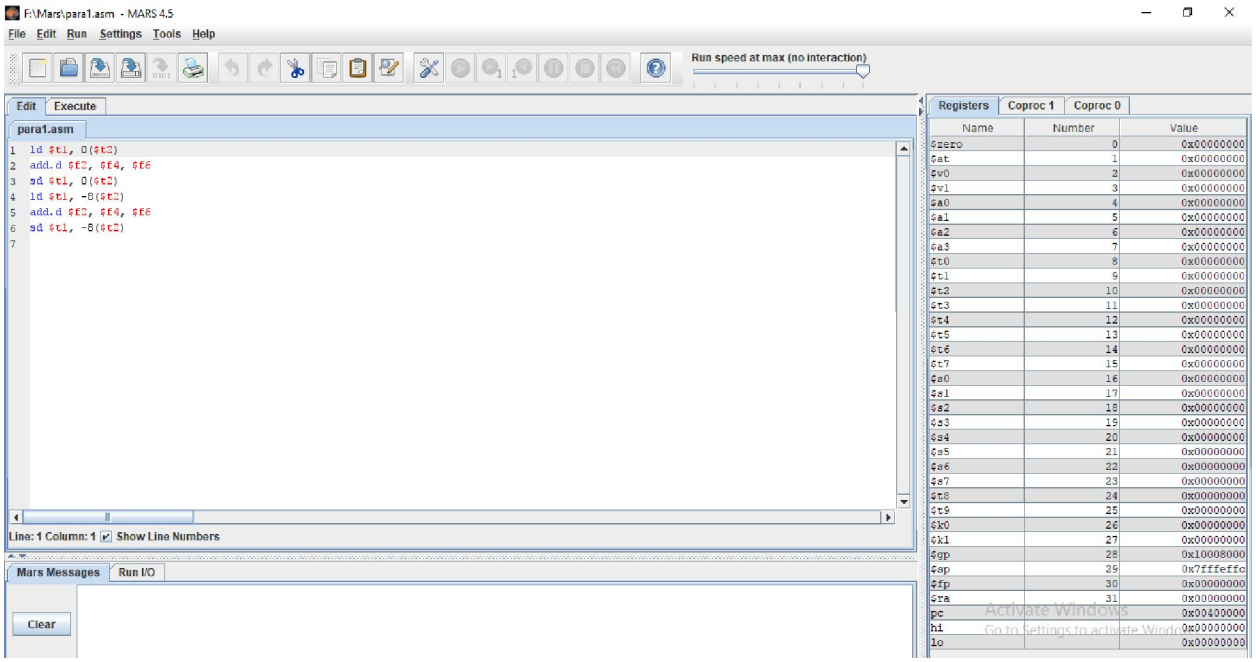
Step 13: The simulator shows REGISTER TYPE INSTRUCTION.

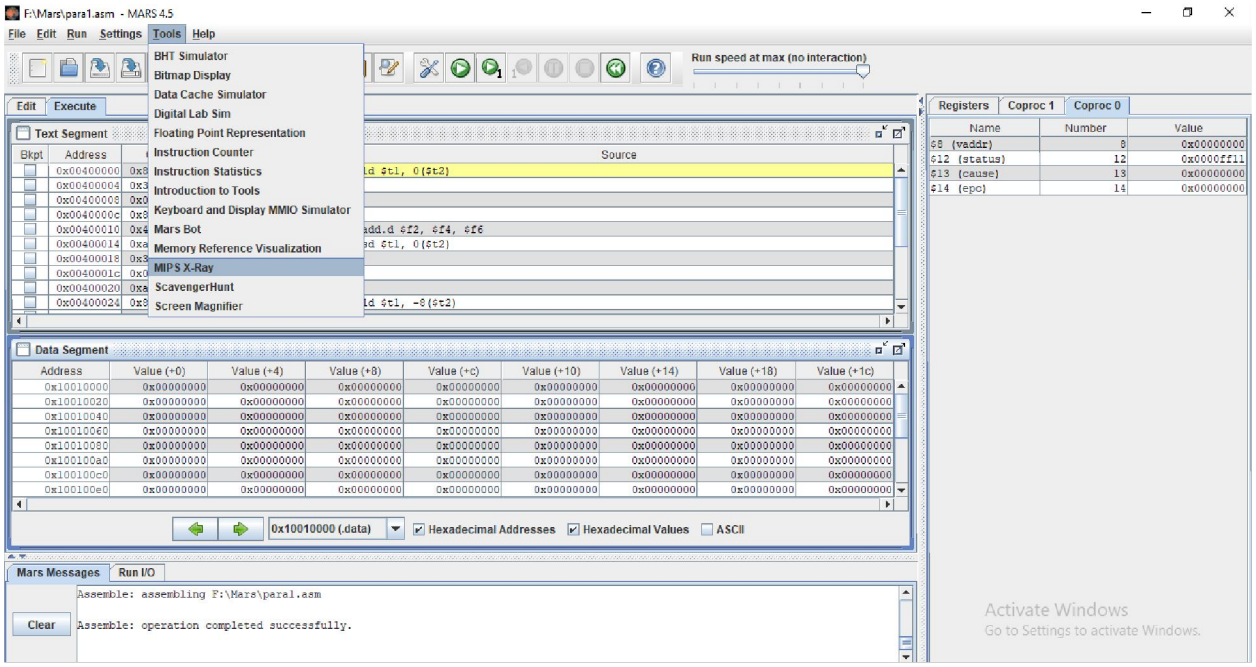
Step 14: Stop.

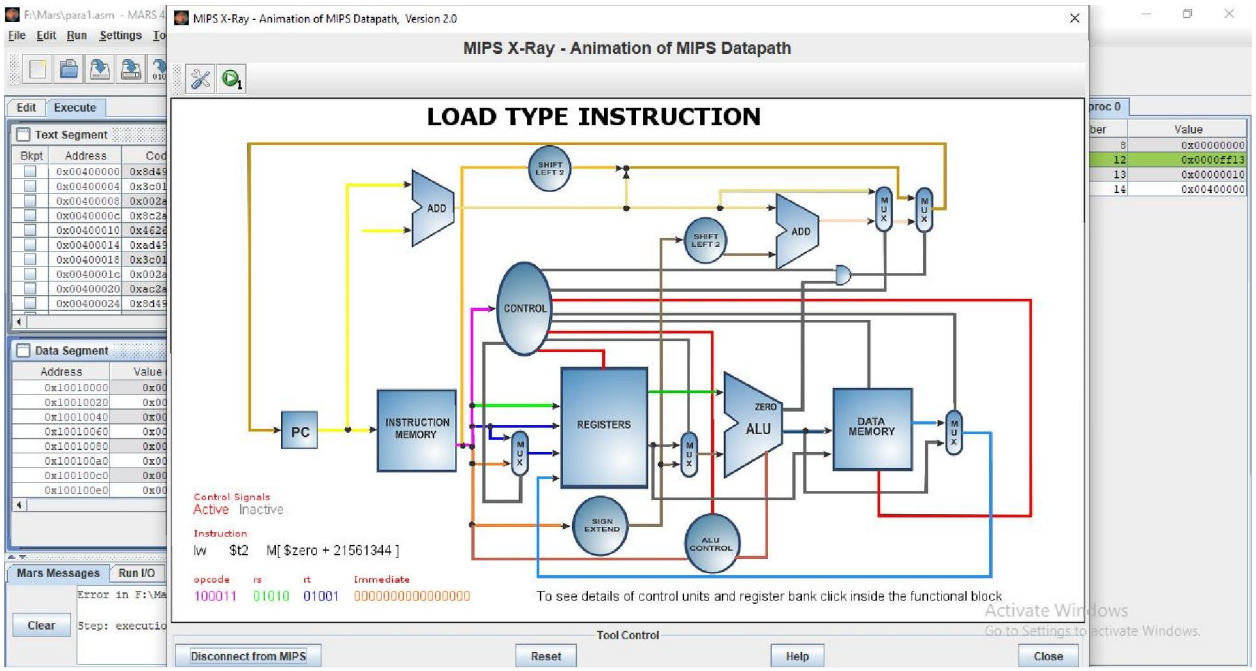
CODE:

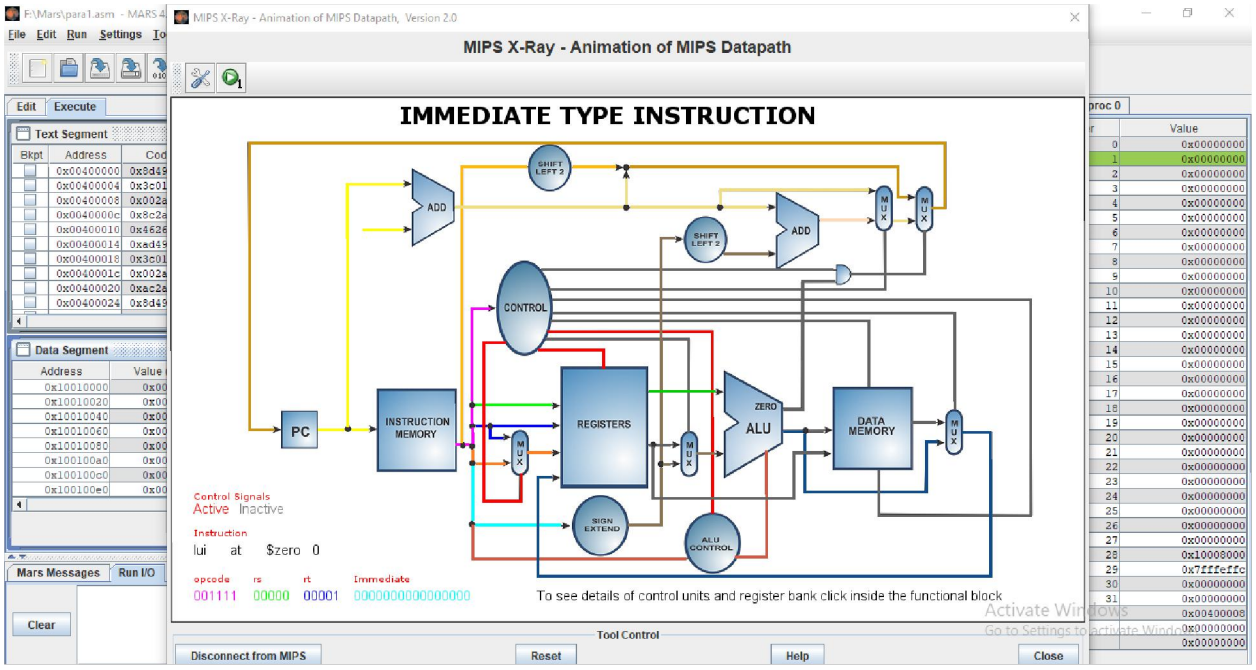
```
ld $t1, 0($t2)
add.d $f2, $f4, $f6
sd $t1, 0($t2)
ld $t1, -8($t2)
add.d $f2, $f4, $f6
sd $t1, -8($t2)
```

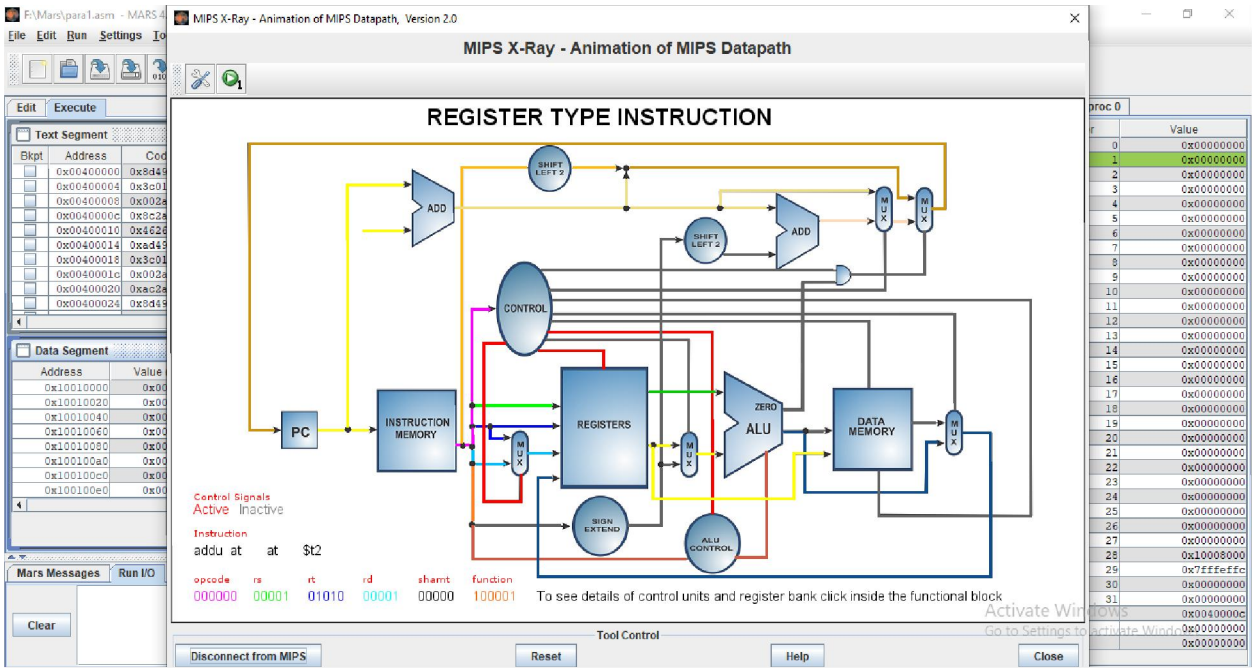
OUTPUT:


F:\Mars\para1.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
para1.asm
1 ld $t1, 0($t2)
2 add.d $f2, $f4, $f6
3 sd $t1, 0($t2)
4 ld $t1, -8($t2)
5 add.d $f2, $f4, $f6
6 sd $t1, -8($t2)
7
Line: 1 Column: 1 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0
Name Number Value
$zero 0 0x00000000
$at 1 0x00000000
$v0 2 0x00000000
$v1 3 0x00000000
$a0 4 0x00000000
$a1 5 0x00000000
$a2 6 0x00000000
$a3 7 0x00000000
$t0 8 0x00000000
$t1 9 0x00000000
$t2 10 0x00000000
$t3 11 0x00000000
$t4 12 0x00000000
$t5 13 0x00000000
$t6 14 0x00000000
$t7 15 0x00000000
$s0 16 0x00000000
$s1 17 0x00000000
$s2 18 0x00000000
$s3 19 0x00000000
$s4 20 0x00000000
$s5 21 0x00000000
$s6 22 0x00000000
$s7 23 0x00000000
$t8 24 0x00000000
$t9 25 0x00000000
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x10008000
$sp 29 0x7fffeffc
$fp 30 0x00000000
$ra 31 0x00000000
pc 0x00400000
hi 0x00000000
lo 0x00000000
Activate Windows
Go to Settings to activate Windows.


F:\Mars\para1.asm - MARS 4.5
File Edit Run Settings Tools Help
BHT Simulator
Bitmap Display
Data Cache Simulator
Digital Lab Sim
Floating Point Representation
Instruction Counter
Instruction Statistics
Introduction to Tools
Keyboard and Display MMIO Simulator
Mars Bot
Memory Reference Visualization
MIPS X-Ray
ScavengerHunt
Screen Magnifier
Run speed at max (no interaction)
Edit Execute
Text Segment
Bkpt Address Source
Data Segment
Address Value (+0) Value (+4) Value (+8) Value (+c) Value (+10) Value (+14) Value (+18) Value (+1c)
0x10010000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010020 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010040 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010060 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010080 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100a0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100c0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100e0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII
Mars Messages Run I/O
Assemble: assembling F:\Mars\para1.asm
Assemble: operation completed successfully.
Clear
Registers Coproc 1 Coproc 0
Name Number Value
$8 (vaddr) 8 0x00000000
$12 (status) 12 0x0000ff11
$13 (cause) 13 0x00000000
$14 (epc) 14 0x00000000
Activate Windows
Go to Settings to activate Windows.


MIPS X-Ray - Animation of MIPS Datapath, Version 2.0
MIPS X-Ray - Animation of MIPS Datapath
LOAD TYPE INSTRUCTION
ADD
SHIFT LEFT 2
ADD
MUX
MUX
CONTROL
PC
INSTRUCTION MEMORY
REGISTERS
ZERO
ALU
DATA MEMORY
MUX
MUX
MUX
SIGN EXTEND
ALU CONTROL
Control Signals
Active Inactive
Instruction
lw $t2 M[$zero + 21561344]
opcode rs rt Immediate
100011 01010 01001 0000000000000000
To see details of control units and register bank click inside the functional block
Tool Control
Disconnect from MIPS
Reset
Help
Close
Activate Windows
Go to Settings to activate Windows.


MIPS X-Ray - Animation of MIPS Datapath, Version 2.0
MIPS X-Ray - Animation of MIPS Datapath
IMMEDIATE TYPE INSTRUCTION
ADD
SHIFT LEFT 2
ADD
MUX
MUX
CONTROL
PC
INSTRUCTION MEMORY
REGISTERS
ZERO
ALU
DATA MEMORY
MUX
MUX
MUX
SIGN EXTEND
ALU CONTROL
Control Signals
Active Inactive
Instruction
lui at $zero 0
opcode rs rt Immediate
001111 00000 00001 0000000000000000
To see details of control units and register bank click inside the functional block
Tool Control
Disconnect from MIPS
Reset
Help
Close
Activate Windows
Go to Settings to activate Windows.



RESULT:

Thus the above simulation was executed successfully.

DATE: 18.01.2021	**SIMULATION OF CACHE MEMORY**
EX.NO: 04	

AIM:

Write a program for Simulation of Cache Memory

SOFTWARE USED:

MARS 4.5

THEORY:

Although load/store architecture forces by its design less memory intensive operations (data has to be held in general purpose registers and are moved to memory only ifit is really necessary) it doesn't mitigate the core problem that is slow memory in comparison with the rest of the CPU. For this purpose caches are used.

Cache is specialized data storage with fast access times that tries to serve as memory copy. It has limited size so it can't be complete copy but storing at least some of the data that were lately used improves overall memory access times. When memory is accessed through cache then there are two possible results. Either requested data were lately used and are still stored in cache or they were not recently accessed and are available only in memory itself. First case is called cache hit and second one is called cache miss. Commonly cache hits are resolved in just a single CPU tick. When cache miss is encountered it commonly takes considerably more time than cache hit**.**

Cache itself is constructed using value store paired with additional meta-information. At minimum cache has to have memory address identifier, called tag, of that specific value stored. On top of that it has to have bits signaling if it contains valid value and in some cases also dirty bit is required (will be addressed when write-back policies are considered). When there is read request then cache checks if it has value with validity bit set and tag matching address and if so then it provides given value instead of accessing memory itself. Having only one value store makes inefficient cache. Because of that caches are constructed from multiple of such value stores. There are few ways those can be grouped together to create bigger cache.

One way is to just add more separate value stores. This describes parameter called Degree of associability or number of ways through cache. When cache is accessed then it goes through all of its values and looks for valid one with tag matching with address. When there is no match then it looks for first one that is not valid and uses that one to get value from memory and storing it there. When all values are valid then it applies replacement policy and replaces one (changing value and corresponding tag). If there is no expansion of this cache (as described in following paragraphs) then it is called fully associative cache.

Another way is to just increase size of value storage. In that case we can store multiple words in a single store. Words stored on top of directly requested one are those that are on addresses right next to it. This divides memory to blocks of words that are loaded to cache together. It also shortens required size of stored tag because in that case we don't have to look for exact address match but just for address that exactly identifies memory block. Parameter specifying number of words to be used is called Block size.

When all value stores are marked as valid and tag from none of them matches the needed address (cache miss is encountered) then there is need for replacement. One of values in cache has to be replaced with value from memory from requested address. Unfortunately it is not directly defined which one should be replaced in case of higher than one degree of associativity. There are multiple algorithms to choose which one should be replaced.

Cache entry to store data is chosen randomly in case of random algorithm. It is one of the simplest possible algorithms as there is no need to store any additional information.

It is question what should be done with value that is currently stored there in case of replacement. There are two possible approaches. One allows immediate override. Other one requires write to memory**.**

PROCEDURE:

Step 1: Start MARS.

Step 2: Use File → New or Click on Create new file icon.

Step 3: Generate the code in the editor.

Step 4: Assemble the program. The provided assembly program is complete.

Step 5: Click Tool →data cache simulator

Step 6: In data cache simulator – click Connect to MIPS

Step 7: Run the simulator.

Step 8: The simulator shows DATA MAPPING

Step 9: Re-run the simulator.

Step 10: The simulator shows FULLY ASSOCIATIVE

Step 11: Re-run the simulator.

Step 12: The simulator shows N-WAY SET ASSOCIATIVE

Step 13: Stop.

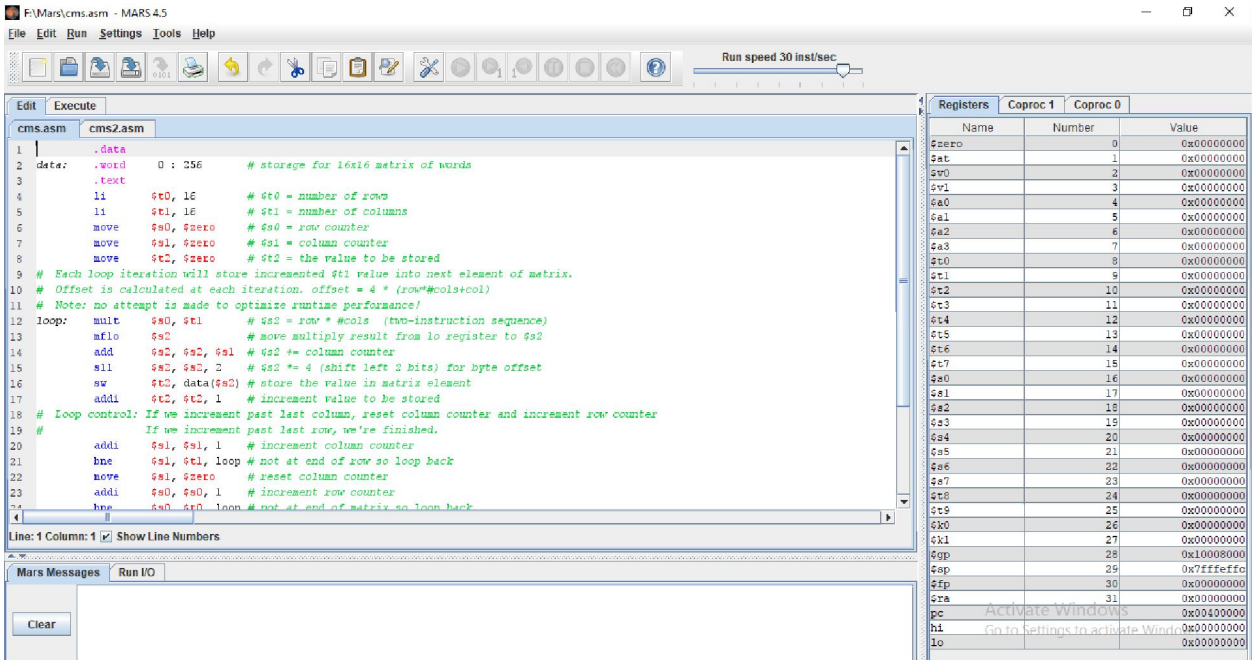
CODE:

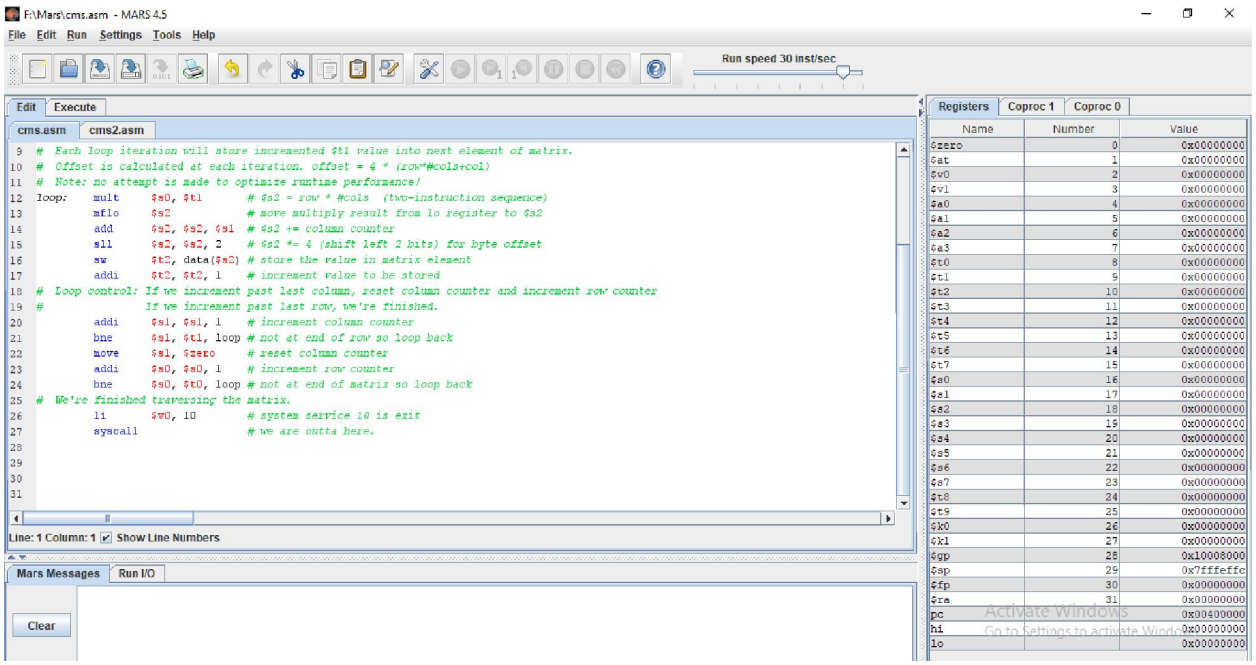
```
#  Row-major order traversal of 16 x 16 array of words.
#  Pete Sanderson
#  31 March 2007
#  To easily observe the row-oriented order, run the Memory Reference
#  Visualization tool with its default settings over this program.
#  You may, at the same time or separately, run the Data Cache Simulator
#  over this program to observe caching performance.  Compare the results
#  with those of the column-major order traversal algorithm.
#  The C/C++/Java-like equivalent of this MIPS program is:
#     int size = 16;
#     int[size][size] data;
#     int value = 0;
#     for (int row = 0; col < size; row++) {
```

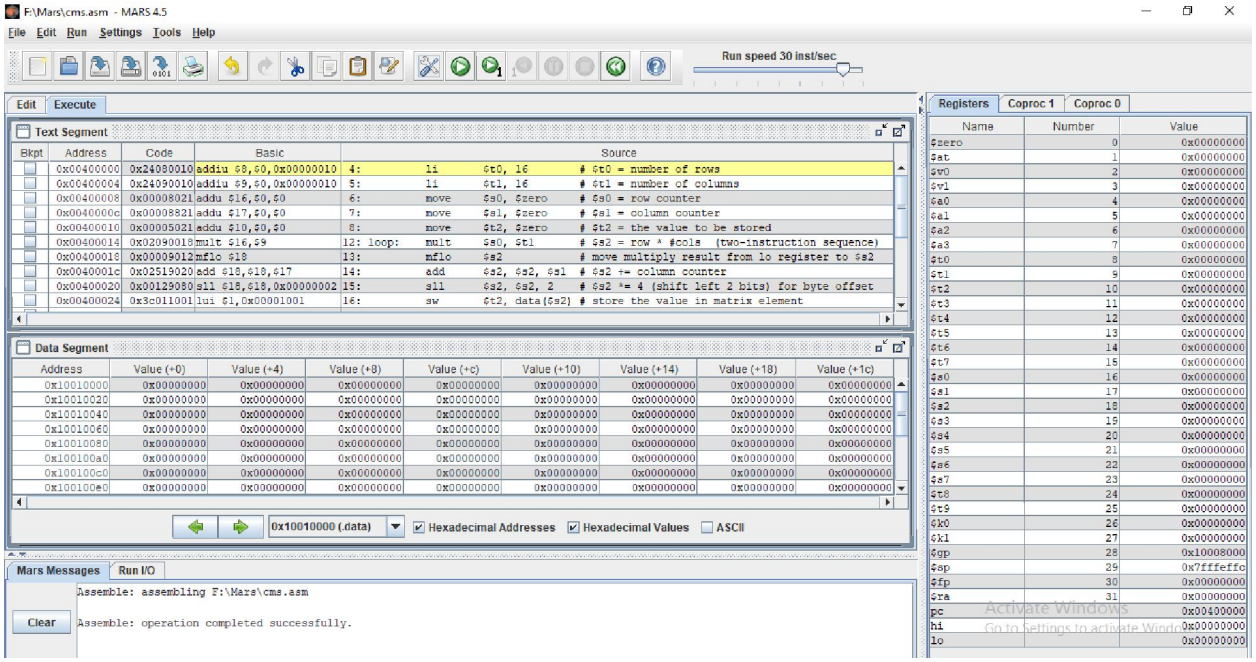
```
#       for (int col = 0; col < size; col++) }
#         data[row][col] = value;
#         value++;
#       }
#    }
#  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change this,
#       three statements need to be changed.
#       1. The array storage size declaration at "data:" needs to be changed from
#         256 (which is 16 * 16) to #columns * #rows.
#       2. The "li" to initialize $t0 needs to be changed to new #rows.
#       3. The "li" to initialize $t1 needs to be changed to new #columns.
#
        .data
data:    .word     0 : 256       # storage for 16x16 matrix of words
        .text
        li       $t0, 16        # $t0 = number of rows
        li       $t1, 16        # $t1 = number of columns
        move     $s0, $zero     # $s0 = row counter
        move     $s1, $zero     # $s1 = column counter
        move     $t2, $zero     # $t2 = the value to be stored
#  Each loop iteration will store incremented $t1 value into next element of matrix.
#  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
#  Note: no attempt is made to optimize runtime performance!
loop:    mult     $s0, $t1       # $s2 = row * #cols  (two-instruction sequence)
        mflo     $s2            # move multiply result from lo register to $s2
        add      $s2, $s2, $s1  # $s2 += column counter
        sll      $s2, $s2, 2    # $s2 *= 4 (shift left 2 bits) for byte offset
        sw       $t2, data($s2) # store the value in matrix element
        addi     $t2, $t2, 1    # increment value to be stored
#  Loop control: If we increment past last column, reset column counter and increment row
counter
```

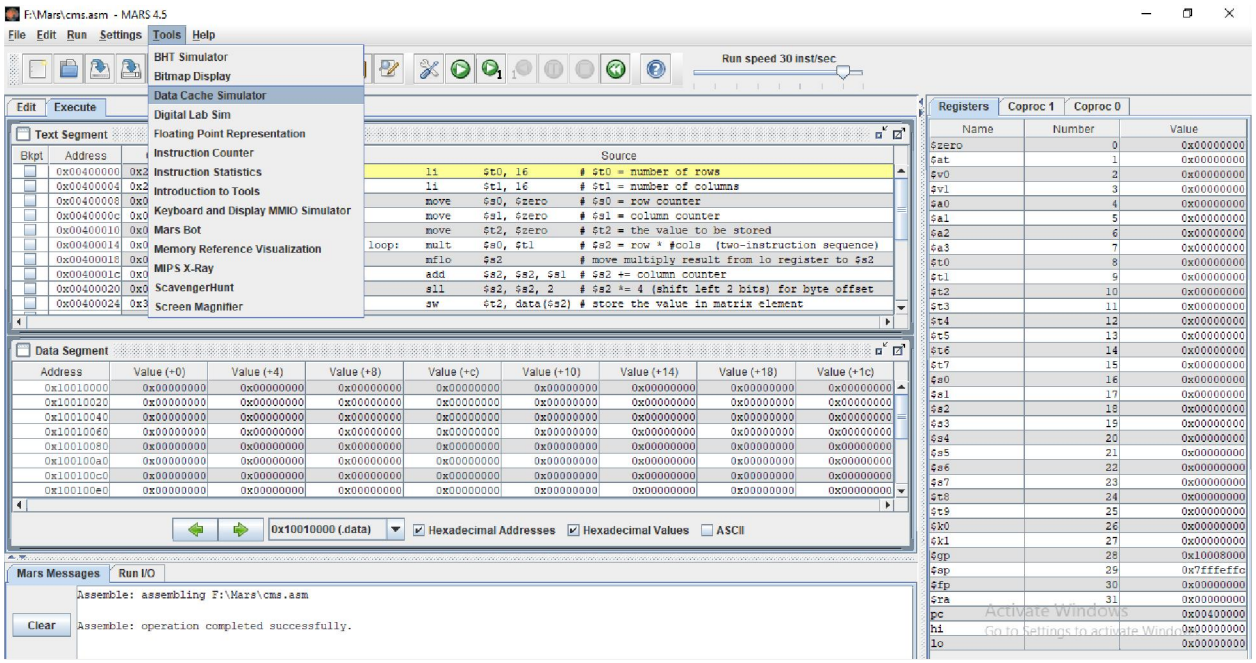
```
#               If we increment past last row, we're finished.
        addi    $s1, $s1, 1     # increment column counter
        bne     $s1, $t1, loop # not at end of row so loop back
        move    $s1, $zero      # reset column counter
        addi    $s0, $s0, 1     # increment row counter
        bne     $s0, $t0, loop # not at end of matrix so loop back
#  We're finished traversing the matrix.
        li      $v0, 10         # system service 10 is exit
        syscall                 # we are outta here.
```

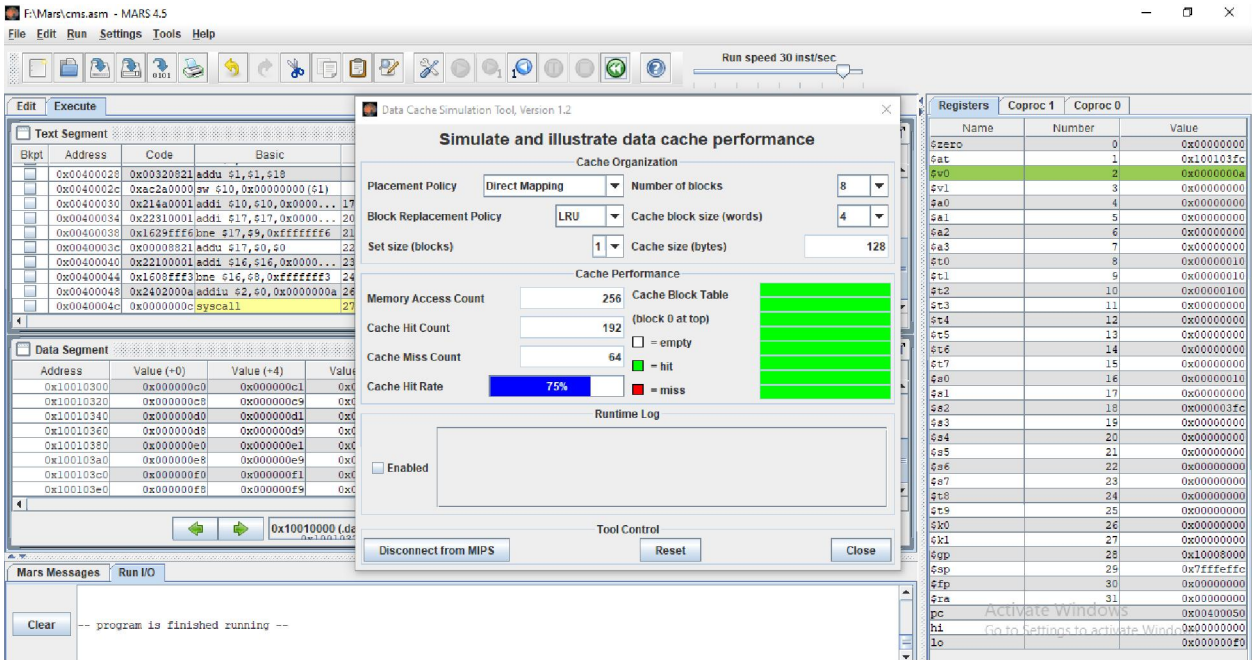
OUTPUT:

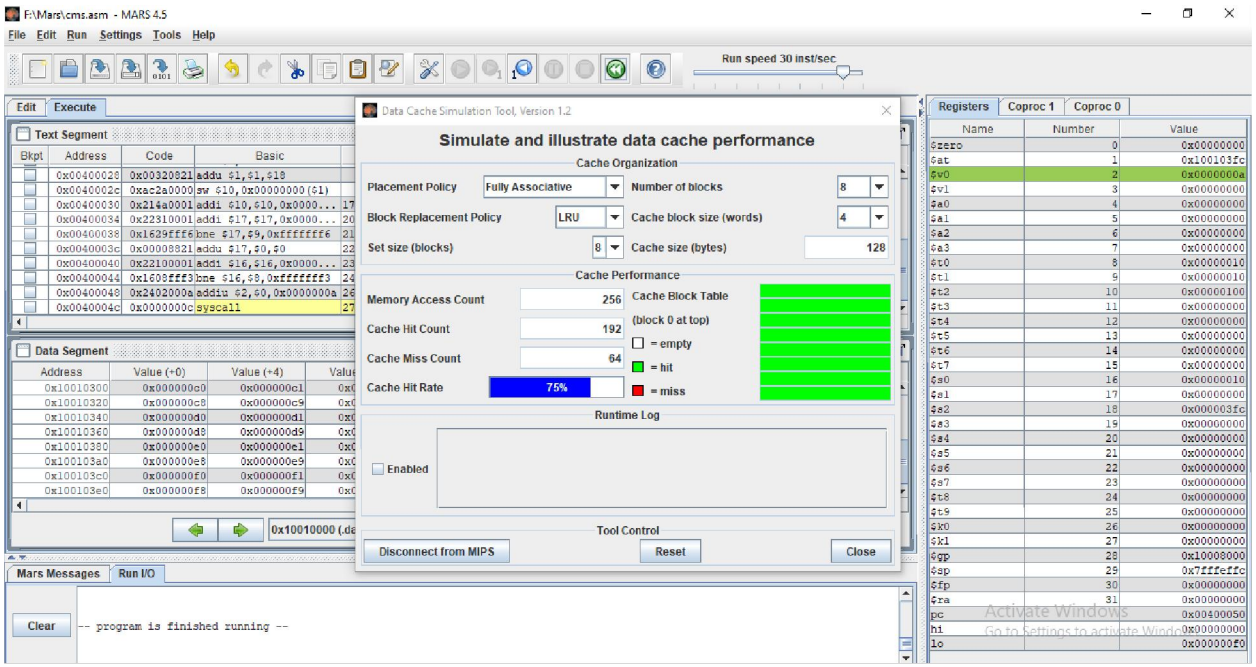


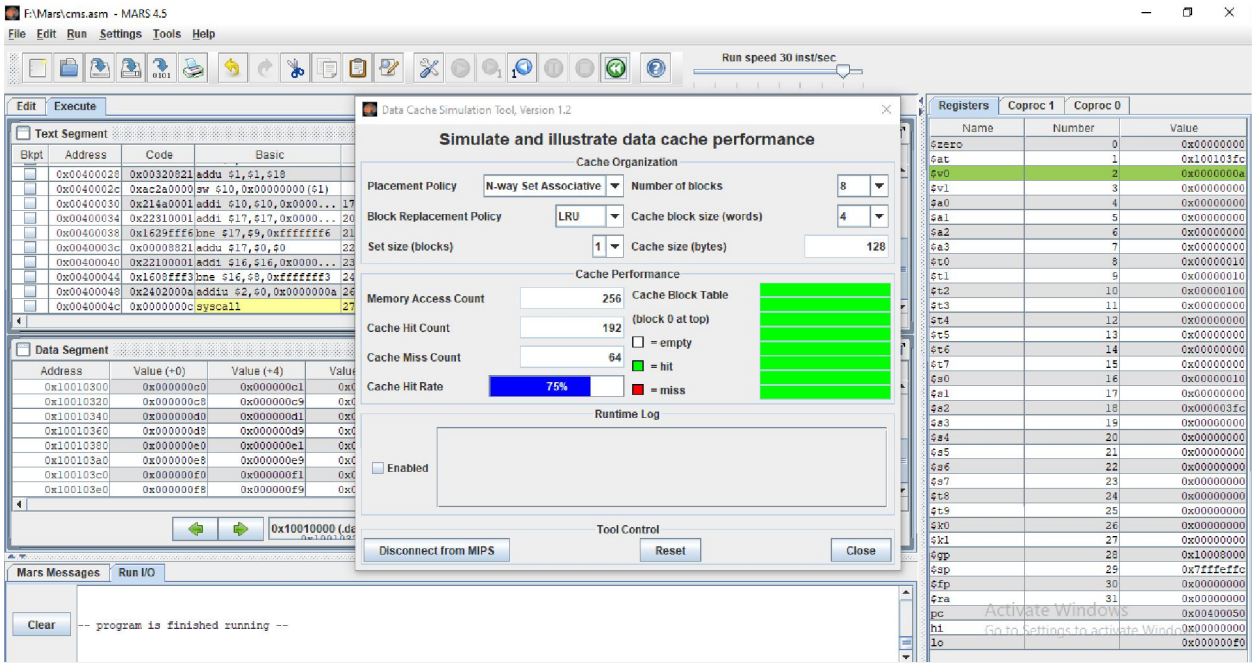

F:\Mars\cms.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed 30 inst/sec
Edit Execute
cms.asm cms2.asm
9 # Each loop iteration will store incremented $t1 value into next element of matrix.
10 # Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
11 # Note: no attempt is made to optimize runtime performance!
12 loop: mult $s0, $t1 # $s2 = row * #cols (two-instruction sequence)
13 mflo $s2 # move multiply result from lo register to $s2
14 add $s2, $s2, $s1 # $s2 += column counter
15 sll $s2, $s2, 2 # $s2 *= 4 (shift left 2 bits) for byte offset
16 sw $t2, data($s2) # store the value in matrix element
17 addi $t2, $t2, 1 # increment value to be stored
18 # Loop control: If we increment past last column, reset column counter and increment row counter
19 # If we increment past last row, we're finished.
20 addi $s1, $s1, 1 # increment column counter
21 bne $s1, $t1, loop # not at end of row so loop back
22 move $s1, $zero # reset column counter
23 addi $s0, $s0, 1 # increment row counter
24 bne $s0, $t0, loop # not at end of matrix so loop back
25 # We're finished traversing the matrix.
26 li $v0, 10 # system service 10 is exit
27 syscall # we are outta here.
Line: 1 Column: 1 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0
Name Number Value
$zero 0 0x00000000
$at 1 0x00000000
$v0 2 0x00000000
$v1 3 0x00000000
$a0 4 0x00000000
$a1 5 0x00000000
$a2 6 0x00000000
$a3 7 0x00000000
$t0 8 0x00000000
$t1 9 0x00000000
$t2 10 0x00000000
$t3 11 0x00000000
$t4 12 0x00000000
$t5 13 0x00000000
$t6 14 0x00000000
$t7 15 0x00000000
$s0 16 0x00000000
$s1 17 0x00000000
$s2 18 0x00000000
$s3 19 0x00000000
$s4 20 0x00000000
$s5 21 0x00000000
$s6 22 0x00000000
$s7 23 0x00000000
$t8 24 0x00000000
$t9 25 0x00000000
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x10008000
$sp 29 0x7fffeffc
$fp 30 0x00000000
$ra 31 0x00000000
pc 0x00400000
hi 0x00000000
lo 0x00000000


F:\Mars\cms.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed 30 inst/sec
Edit Execute
Text Segment
Bkpt Address Code Basic Source
0x00400000 0x24080010 addiu $8,$0,0x00000010 4: li $t0, 16 # $t0 = number of rows
0x00400004 0x24090010 addiu $9,$0,0x00000010 5: li $t1, 16 # $t1 = number of columns
0x00400008 0x00008021 addu $16,$0,$0 6: move $s0, $zero # $s0 = row counter
0x0040000c 0x00008821 addu $17,$0,$0 7: move $s1, $zero # $s1 = column counter
0x00400010 0x00005021 addu $10,$0,$0 8: move $t2, $zero # $t2 = the value to be stored
0x00400014 0x02090018 mult $16,$9 12: loop: mult $s0, $t1 # $s2 = row * #cols (two-instruction sequence)
0x00400018 0x00009012 mflo $18 13: mflo $s2 # move multiply result from lo register to $s2
0x0040001c 0x02519020 add $18,$18,$17 14: add $s2, $s2, $s1 # $s2 += column counter
0x00400020 0x00129080 sll $18,$18,0x00000002 15: sll $s2, $s2, 2 # $s2 *= 4 (shift left 2 bits) for byte offset
0x00400024 0x3c011001 lui $1,0x00001001 16: sw $t2, data($s2) # store the value in matrix element
Data Segment
Address Value (+0) Value (+4) Value (+8) Value (+c) Value (+10) Value (+14) Value (+18) Value (+1c)
0x10010000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010020 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010040 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010060 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010080 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100a0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100c0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100e0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII
Mars Messages Run I/O
Assemble: assembling F:\Mars\cms.asm
Assemble: operation completed successfully.
Clear
Registers Coproc 1 Coproc 0
Name Number Value
$zero 0 0x00000000
$at 1 0x00000000
$v0 2 0x00000000
$v1 3 0x00000000
$a0 4 0x00000000
$a1 5 0x00000000
$a2 6 0x00000000
$a3 7 0x00000000
$t0 8 0x00000000
$t1 9 0x00000000
$t2 10 0x00000000
$t3 11 0x00000000
$t4 12 0x00000000
$t5 13 0x00000000
$t6 14 0x00000000
$t7 15 0x00000000
$s0 16 0x00000000
$s1 17 0x00000000
$s2 18 0x00000000
$s3 19 0x00000000
$s4 20 0x00000000
$s5 21 0x00000000
$s6 22 0x00000000
$s7 23 0x00000000
$t8 24 0x00000000
$t9 25 0x00000000
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x10008000
$sp 29 0x7fffeffc
$fp 30 0x00000000
$ra 31 0x00000000
pc 0x00400000
hi 0x00000000
lo 0x00000000


F:\Mars\cms.asm - MARS 4.5
File Edit Run Settings Tools Help
BHT Simulator
Bitmap Display
Data Cache Simulator
Digital Lab Sim
Floating Point Representation
Instruction Counter
Instruction Statistics
Introduction to Tools
Keyboard and Display MMIO Simulator
Mars Bot
Memory Reference Visualization
MIPS X-Ray
ScavengerHunt
Screen Magnifier
Run speed 30 inst/sec
Edit Execute
Text Segment
Data Segment
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
Assemble: assembling F:\Mars\cms.asm
Assemble: operation completed successfully.
Clear


F:\Mars\cms.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed 30 inst/sec
Data Cache Simulation Tool, Version 1.2
Simulate and illustrate data cache performance
Cache Organization
Placement Policy
Direct Mapping
Number of blocks
Block Replacement Policy
LRU
Cache block size (words)
Set size (blocks)
Cache size (bytes)
128
Cache Performance
Memory Access Count
256
Cache Hit Count
192
Cache Miss Count
64
Cache Hit Rate
75%
Cache Block Table
(block 0 at top)
= empty
= hit
= miss
Runtime Log
Enabled
Tool Control
Disconnect from MIPS
Reset
Close
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
-- program is finished running --
Clear





RESULT:

Thus the above simulation was executed successfully.

DATE: 29.01.2021	SIMULATION OF MULTIPROCESSOR
EX.NO: 05	

AIM:

Write a program for the Simulation of multiprocessor

SOFTWARE:

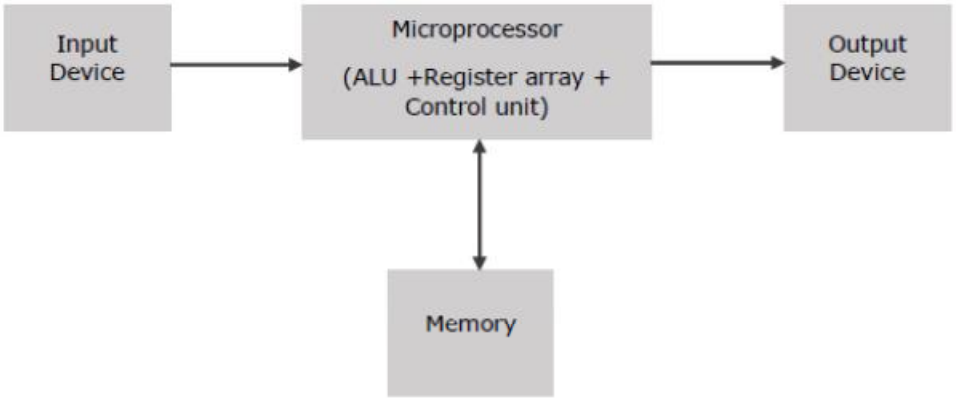
GNUSIM8085

THEORY:

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it.

Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device. Register array consists of registers identified by letters like B, C, D, E, H, L and accumulator. The control unit controls the flow of data and instructions within the computer.

Block Diagram of a Basic Microcomputer



Microprocessor

The microprocessor follows a sequence: Fetch, Decode, and then Execute.Initially, the instructions are stored in the memory in a sequential order. The microprocessor fetches those instructions from the memory, then decodes it and executes those instructions till STOP instruction is reached. Later, it sends the result in binary to the output port. Between these processes, the register stores the temporarily data and ALU performs the computing functions.

List of Terms Used in a Microprocessor

Here is a list of some of the frequently used terms in a microprocessor −

- **Instruction Set** − It is the set of instructions that the microprocessor can understand.
- **Bandwidth** − It is the number of bits processed in a single instruction.
- **Clock Speed** − It determines the number of operations per second the processor can perform. It is expressed in megahertz (MHz) or gigahertz (GHz).It is also known as Clock Rate.
- **Word Length** − It depends upon the width of internal data bus, registers, ALU, etc. An 8-bit microprocessor can process 8-bit data at a time. The word length ranges from 4 bits to 64 bits depending upon the type of the microcomputer.
- **Data Types** − The microprocessor has multiple data type formats like binary, BCD, ASCII, signed and unsigned numbers.

Features of a Microprocessor

Here is a list of some of the most prominent features of any microprocessor −

- **Cost-effective** − The microprocessor chips are available at low prices and results its low cost.
- **Size** − The microprocessor is of small size chip, hence is portable.
- **Low Power Consumption** − Microprocessors are manufactured by using metaloxide semiconductor technology, which has low power consumption.

- **Versatility** − The microprocessors are versatile as we can use the same chip in a number of applications by configuring the software program.
- **Reliability** − The failure rate of an IC in microprocessors is very low, hence it is reliable.

8085 MICROPROCESSOR

8085 is pronounced as "eighty-eighty-five" microprocessor. It is an 8-bit microprocessor designed by Intel in 1977 using NMOS technology.

It has the following configuration −

- 8-bit data bus
- 16-bit address bus, which can address upto 64KB
- A 16-bit program counter
- A 16-bit stack pointer
- Six 8-bit registers arranged in pairs: BC, DE, HL
- Requires +5V supply to operate at 3.2 MHZ single phase clock

It is used in washing machines, microwave ovens, mobile phones, etc.

8085 Microprocessor – Functional Units

8085 consists of the following functional units −

Accumulator

It is an 8-bit register used to perform arithmetic, logical, I/O & LOAD/STORE operations. It is connected to internal data bus & ALU.

Arithmetic and logic unit

As the name suggests, it performs arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8-bit data.

General purpose register

There are 6 general purpose registers in 8085 processor, i.e. B, C, D, E, H& L. Each register can hold 8-bit data.

These registers can work in pair to hold 16-bit data and their pairing combination is like B-C, D-E & H-L.

Program counter

It is a 16-bit register used to store the memory address location of the next instruction to be executed. Microprocessor increments the program whenever an instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed.

Stack pointer

It is also a 16-bit register works like stack, which is always incremented/decremented by 2 during push & pop operations.

Temporary register

It is an 8-bit register, which holds the temporary data of arithmetic and logical operations.

Flag register

It is an 8-bit register having five 1-bit flip-flops, which holds either 0 or 1 depending upon the result stored in the accumulator.These are the set of 5 flip-flops −

- Sign (S)
- Zero (Z)
- Auxiliary Carry (AC)
- Parity (P)
- Carry (C)

Its bit position is shown in the following table −

D7	D6	D5	D4	D3	D2	D1	D0
S	Z		AC		P		CY

Instruction register and decoder

It is an 8-bit register. When an instruction is fetched from memory then it is stored in the Instruction register. Instruction decoder decodes the information present in the Instruction register.

Timing and control unit

It provides timing and control signal to the microprocessor to perform operations. Following are the timing and control signals, which control external and internal circuits −

- Control Signals: READY, RD', WR', ALE
- Status Signals: S0, S1, IO/M'
- DMA Signals: HOLD, HLDA
- RESET Signals: RESET IN, RESET OUT

Interrupt control

As the name suggests it controls the interrupts during a process. When a microprocessor is executing a main program and whenever an interrupt occurs, the microprocessor shifts the control from the main program to process the incoming request. After the request is completed, the control goes back to the main program.There are 5 interrupt signals in 8085 microprocessor: INTR, RST 7.5, RST 6.5, RST 5.5, TRAP.

Serial Input/output control

It controls the serial data communication by using these two instructions: SID (Serial input data) and SOD (Serial output data).

Address buffer and address-data buffer

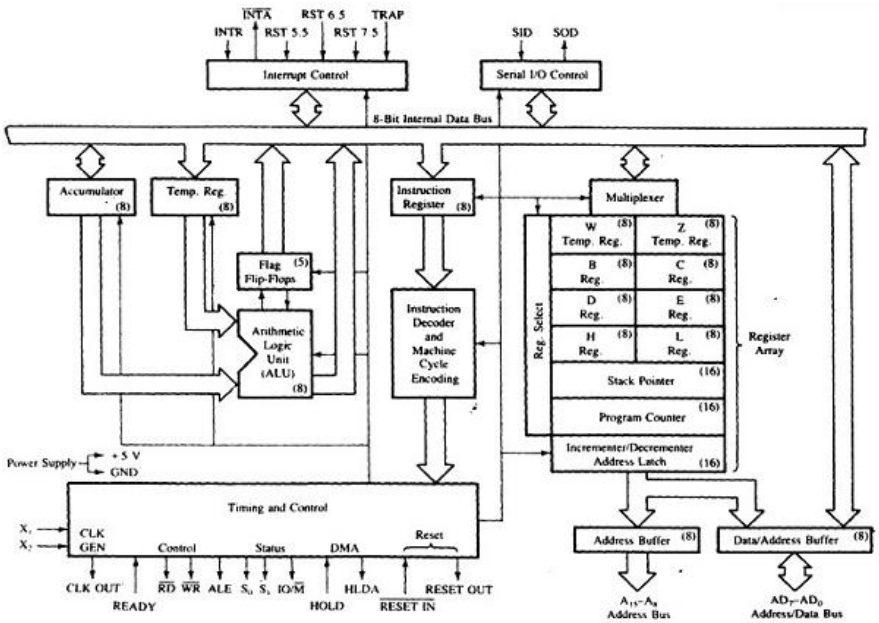
The content stored in the stack pointer and program counter is loaded into the address buffer and address-data buffer to communicate with the CPU. The memory and I/O chips are connected to these buses; the CPU can exchange the desired data with the memory and I/O chips.

Address bus and data bus

Data bus carries the data to be stored. It is bidirectional, whereas address bus carries the location to where it should be stored and it is unidirectional. It is used to transfer the data & Address I/O devices.

8085 Architecture

We have tried to depict the architecture of 8085 with this following image −



PROCEDURE:

Step 1: Start GNU 8085 SIMULATOR .

Step 2: Use File → New or Click on Create new file icon.

Step 3: Generate the code in the editor.

Step 4: Assemble the program. The provided assembly program is complete.

Step 5: Stop.

CODE:

Addition of two numbers :

```
lda var1
movb,a
lda var2
add b
sta var3
hlt
var1: db 04h
var2: db 09h
var3: db 00h
```

Multiply two 8 bit numbers without shifting :

```
lxi h,
var; multiplicand
mvi d,00h
move,m
inx h
movc,m; multiplier as counter for repeated addition
mvi h,00h
mvi l,00h
back: dad d
dcr c
jnz back
```

```
shld result
hlt
var: db 08h
var2: db 07h
result: db 00h
result2: db 00h
```

Division of 8bit number :

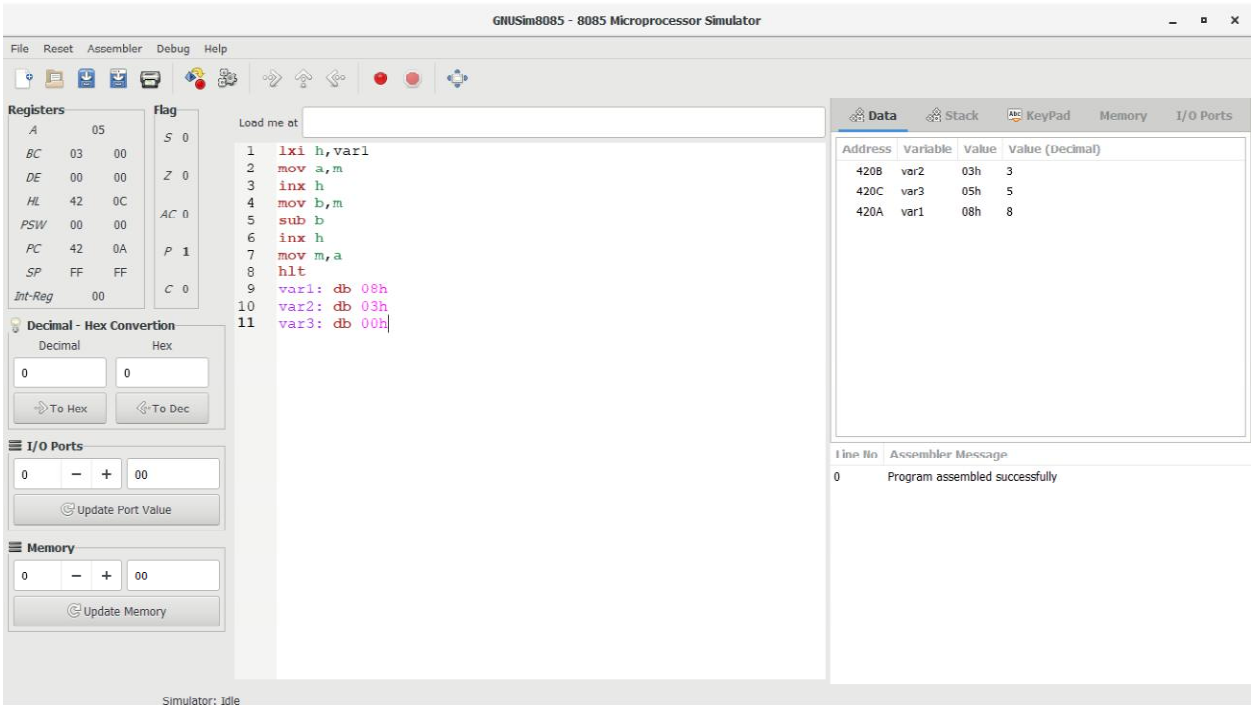
```
lhldvar;dividend
lda var2;divisor
movb,a
mvi c,08h
back: dad h
mova,h
sub b
jc forward
movh,a
inr l
forward: dcr c
jnz back
shld var3
hlt
var: db 0ch
var1: db 00h
var2: db 05h
var3: db 00h
var4: db 00h
```

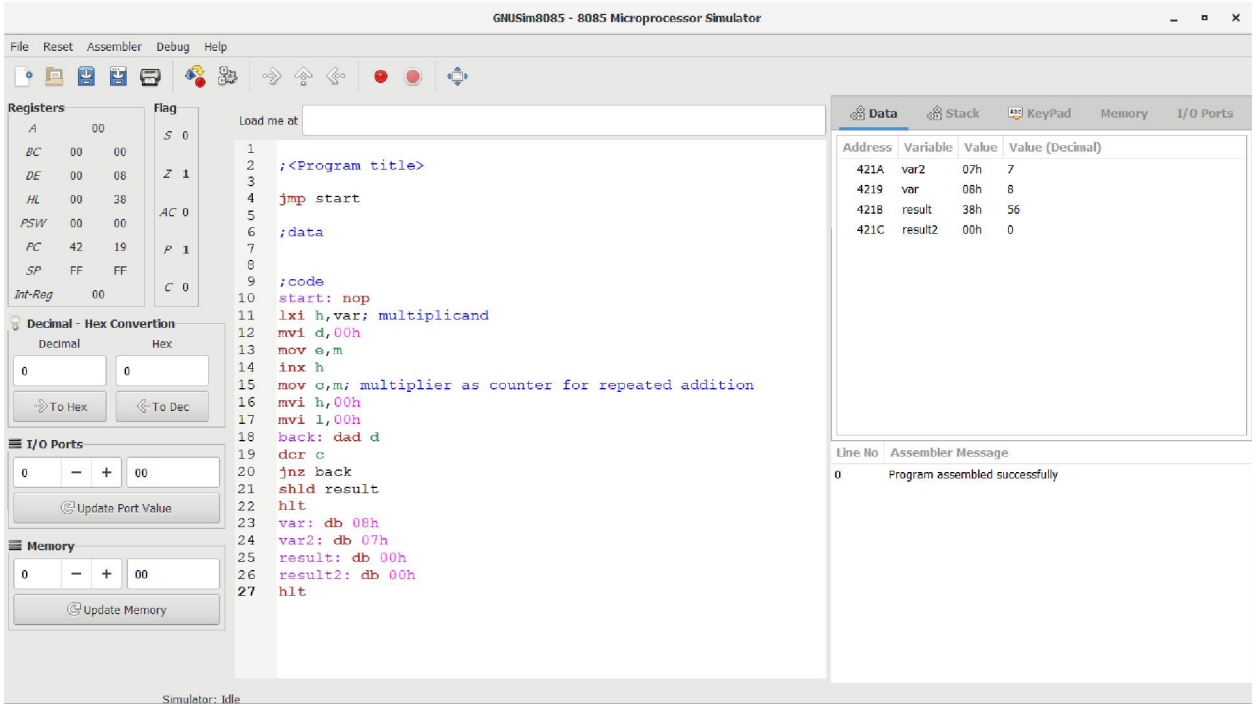
Factorial of the number :

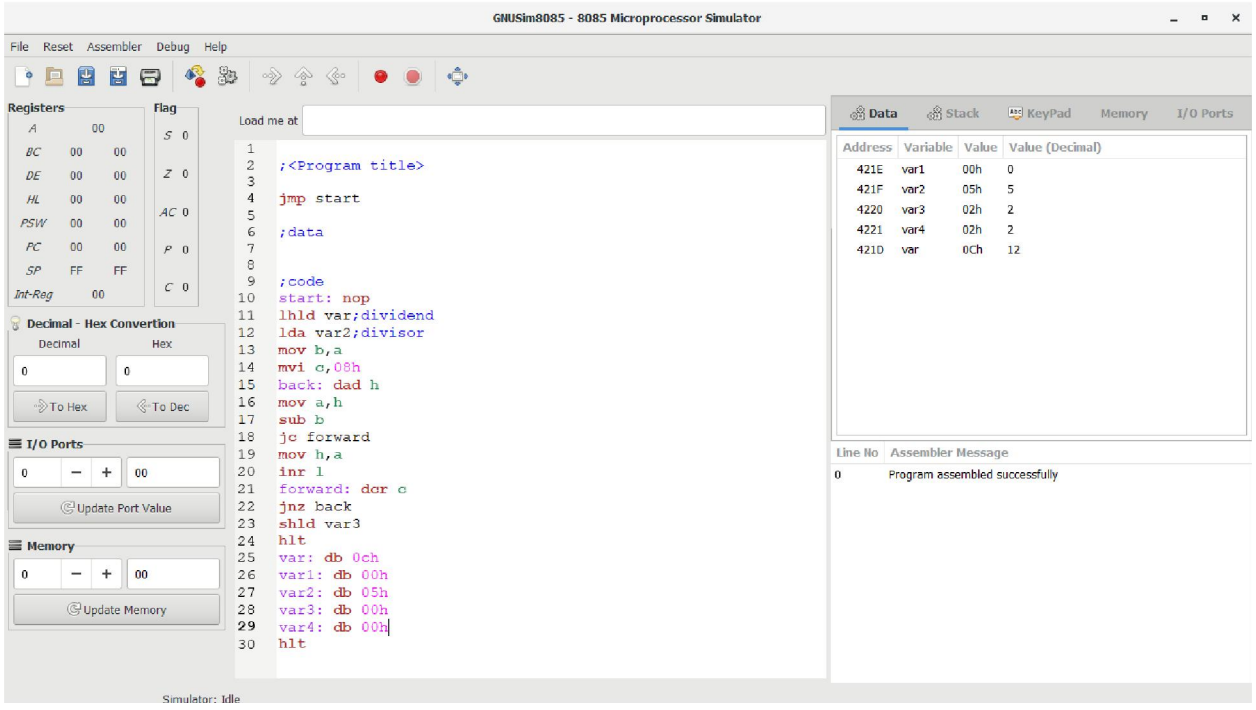
```
lxi sp,27ffh
lda var2
cpi 02h
jc last
mvi d,00h
move,a
dcr a
```

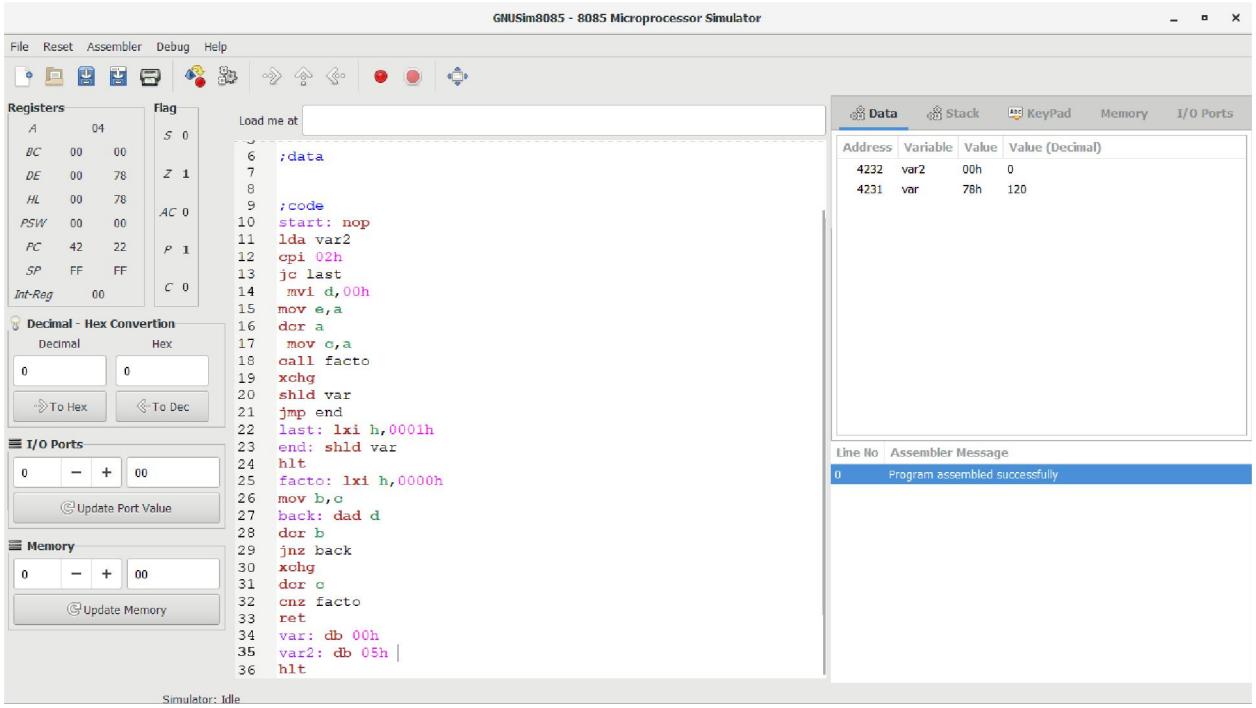
```
movc,a
call facto
xchg
shldvar
jmp end
last: lxi h,0001h
end: shldvar
hlt
facto: lxi h,0000h
movb,c
back: dad d
dcr b
jnz back
xchg
dcr c
cnz facto
ret
var: db 00h
var2: db03h ; input the number 3 here, donot give number more than 5
```

OUTPUT:




GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Registers
Flag
Load me at
Data Stack KeyPad Memory I/O Ports
Address Variable Value Value (Decimal)
421A var2 07h 7
4219 var 08h 8
4218 result 38h 56
421C result2 00h 0
Decimal - Hex Convertion
I/O Ports
Memory
Update Port Value
Update Memory
To Hex
To Dec
Line No Assembler Message
0 Program assembled successfully
;<Program title>
jmp start
;data
;code
start: nop
lxi h,var; multiplicand
mvi d,00h
mov e,m
inx h
mov c,m; multiplier as counter for repeated addition
mvi h,00h
mvi l,00h
back: dad d
dcr c
jnz back
shld result
hlt
var: db 08h
var2: db 07h
result: db 00h
result2: db 00h
hlt
Simulator: Idle


GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Registers
Flag
Load me at
Data Stack KeyPad Memory I/O Ports
Address Variable Value Value (Decimal)
421E var1 00h 0
421F var2 05h 5
4220 var3 02h 2
4221 var4 02h 2
4210 var 0Ch 12
Decimal - Hex Convertion
I/O Ports
Memory
Update Port Value
Update Memory
To Hex
To Dec
Line No Assembler Message
0 Program assembled successfully
;<Program title>
jmp start
;data
;code
start: nop
lhld var;dividend
lda var2;divisor
mov b,a
mvi c,08h
back: dad h
mov a,h
sub b
jc forward
mov h,a
inr l
forward: dcr c
jnz back
shld var3
hlt
var: db 0ch
var1: db 00h
var2: db 05h
var3: db 00h
var4: db 00h
hlt
Simulator: Idle



RESULT:

Thus the above simulation was executed successfully

DATE: 05.02.2021	SIMULATION OF VECTOR PROCESSOR
EX.NO: 06	

AIM:

To understand the Simulation of Vector Processor.

THEORY:

There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems require vast number of computations on multiple data items, that will take a conventional computer(with scalar processor) days or even weeks to complete.

Such complex instructions, which operates on multiple data at the same time, requires a better way of instruction execution, which was achieved by Vector processors.

Scalar CPUs can manipulate one or two data items at a time, which is not very efficient. Also, simple instructions like ADD A to B, and store into C are not practically efficient.

Addresses are used to point to the memory location where the data to be operated will be found, which leads to added overhead of data lookup. So until the data is found, the CPU would be sitting ideal, which is a big performance issue.

Hence, the concept of Instruction Pipeline comes into picture, in which the instruction passes through several sub-units in turn. These sub-units perform various independent functions, for example: the first one decodes the instruction, the second sub-unit fetches the data and the third sub-unit performs the math itself. Therefore, while the data is fetched for one instruction, CPU does not sit idle, it rather works on decoding the next instruction set, ending up working like an assembly line.

Vector processor, not only use Instruction pipeline, but it also pipelines the data, working on multiple data at the same time.

A normal scalar processor instruction would be ADD A, B, which leads to addition of two operands, but what if we can instruct the processor to ADD a group of numbers(from 0 to n memory location) to another group of numbers(lets say, n to k memory location). This can be achieved by vector processors.

In vector processor a single instruction, can ask for multiple data operations, which saves time, as instruction is decoded once, and then it keeps on operating on different data items.

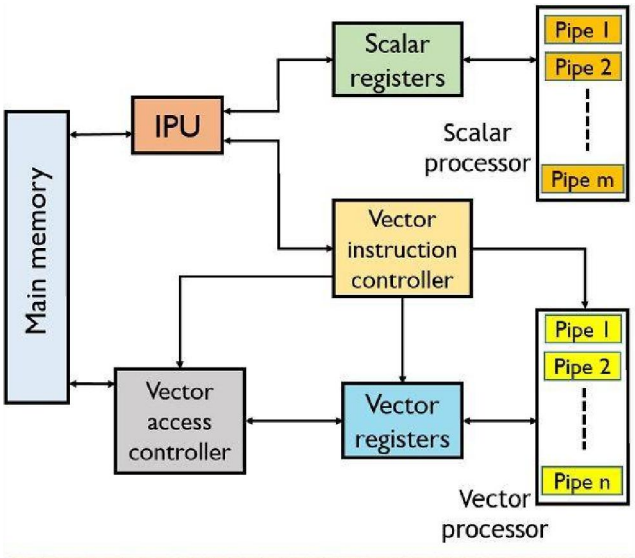
Applications of Vector Processors

Computer with vector processing capabilities are in demand in specialized applications. The following are some areas where vector processing is used:

1. Petroleum exploration.
2. Medical diagnosis.
3. Data analysis.
4. Weather forecasting.
5. Aerodynamics and space flight simulations.
6. Image processing.
7. Artificial intelligence.

Architecture and Working

The figure below represents the typical diagram showing vector processing by a vector computer:



Functional Diagram of Vector Computer

The functional units of a vector computer are as follows:

- IPU or instruction processing unit
- Vector register
- Scalar register
- Scalar processor
- Vector instruction controller
- Vector access controller
- Vector processor

Let us now understand the overall operation performed by the vector computer.

As it has several functional pipes thus it can execute the instructions over the operands. We know that both data and instructions are present in the memory at the desired memory location. So, the instruction processing unit i.e., IPU fetches the instruction from the memory.

Once the instruction is fetched then IPU determines either the fetched instruction is scalar or vector in nature. If it is scalar in nature, then the instruction is transferred to the scalar register and then further scalar processing is performed.

While, when the **instruction is a vector** in nature then it is fed to the vector instruction controller. This vector instruction controller first decodes the vector instruction then accordingly determines the address of the vector operand present in the memory.

Then it gives a signal to the **vector access controller** about the demand of the respective operand. This vector access controller then fetches the desired operand from the memory. Once the operand is fetched then it is provided to the instruction register so that it can be processed at the vector processor.

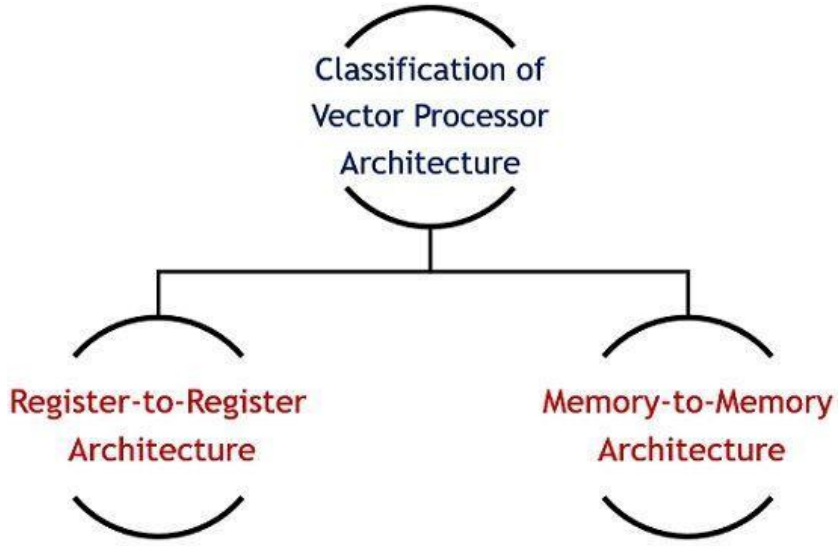
At times when multiple vector instructions are present, then the vector instruction controller provides the multiple vector instructions to the task system. And in case the task system shows that the vector task is very long then the processor divides the task into subvectors.

These sub-vectors are fed to the vector processor that makes use of several pipelines in order to execute the instruction over the operand fetched from the memory at the same time.

The various vector instructions are scheduled by the vector instruction controller.

Classification of Vector Processor

The classification of vector processor relies on the ability of vector formation as well as the presence of vector instruction for processing. So, depending on these criteria, vector processing is classified as follows:



Register to Register Architecture

This architecture is highly used in vector computers. As in this architecture, the fetching of the operand or previous results indirectly takes place through the main memory by the use of registers.

The several vector pipelines present in the vector computer help in retrieving the data from the registers and also storing the results in the desired register. These vector registers are user instruction programmable.

This means that according to the register address present in the instruction, the data is fetched and stored in the desired register. These vector registers hold fixed length like the register length in a normal processing unit.

Some examples of a supercomputer using the register to register architecture are **Cray – 1**, **Fujitsu** etc.

Memory to Memory Architecture

Here in memory to memory architecture, the operands or the results are directly fetched from the memory despite using registers. However, it is to be noted here that the address of the desired data to be accessed must be present in the vector instruction.

This architecture enables the fetching of data of size 512 bits from memory to pipeline. However, due to high memory access time, the pipelines of the vector computer requires higher startup time, as higher time is required to initiate the vector instruction.

Some examples of supercomputers that possess memory to memory architecture are **Cyber 205**, **CDC** etc.

Advantages of Vector Processor

- Vector processor uses vector instructions by which code density of the instructions can be improved.
- The sequential arrangement of data helps to handle the data by the hardware in a better way.
- It offers a reduction in instruction bandwidth.

RESULT:

Thus the process of above simulation is understood.

DATE: 12.02.2021	SIMULATION OF THREAD LEVEL PARALLELISM
EX.NO: 07	

AIM:

To understand the Simulation of Thread Level Parallelism.

THEORY:

Thread level parallelism is a form of parallel computing for multiple processors using a technique for distributing execution of processes and threads across different parallel processor nodes. It contrasts to data parallelism as another form of parallelism.

In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. Different execution threads communicate with one another usually to pass data as they work.

As a simple example, if we are running code on a 2-processor system (CPUs "a" & "b") in a parallel computing environment and we want to do tasks "A" and "B", it is possible to tell CPU "a" to do task "A" and CPU "b" to do task 'B" simultaneously (at the same time), in order to reduce the runtime of the execution.

Thread level parallelism is used by multi-user and multitasking operating systems, and applications depending on processes and threads, unlike data processing applications.

Thread level parallelism is an important alternative to instruction level parallelism, primarily because it could be more cost-effective to exploit than instruction level parallelism. There are many important applications where thread level parallelism occurs naturally, as it does in many server applications. Similarly, a number of applications naturally exploit ***data level parallelism***, where the same operation can be performed on multiple data. We shall discuss about exploiting data level parallelism in a later module.

Since ILP and TLP exploit two different types of parallel structure in a program, it is a natural option to combine these two types of parallelism. The datapath that has already been designed has a number of functional units remaining idle because of the insufficient ILP caused by stalls and dependences. This can be utilized to exploit TLP and thus make the functional units

busy. There are predominantly two strategies for exploiting TLP along with ILP – ***Multithreading*** and its variants, viz., ***Simultaneous Multi Threading (SMT)*** and ***Chip Multi Processors (CMP)***. In the case of SMT, multiple threads share the same large processor which reduces under-utilization and does efficient resource allocation. In the case of CMPs, each thread executes on its own mini processor, which results in a simple design and low interference between threads. We will discuss about both these approaches.

Multithreading: Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. In order to enable this, the processor duplicates the independent state of each thread – a separate copy of the register file, a separate PC, and a separate page table. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

There are two main approaches to multithreading – Fine grained and Coarse grained. ***Fine-grained multithreading*** switches between threads on each instruction, causing the execution of multiple threads to be interleaved. This interleaving is normally done in a round-robin fashion, skipping any threads that are stalled at that time. In order to support this, the CPU must be able to switch threads on every clock cycle. The main advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. But it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading switches threads only on costly stalls, such as level two cache misses. This allows some time for thread switching and is much less likely to slow the processor down, since instructions from other threads will only be issued, when a thread encounters a costly stall. Coarse-grained multithreading, however, is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grain multithreading. Because a CPU with coarse-grained

multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen and then fill in instructions from the new thread. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous Multithreading: This is a variant on multithreading. When we only issue instructions from one thread, there may not be enough parallelism available and all the functional units may not be used. Instead, if we issue instructions from multiple threads in the same clock cycle, we will be able to better utilize the functional units. This is the concept of simultaneous multithreading. We try to use the resources of a multiple issue, dynamically scheduled superscalar to exploit TLP on top of ILP. The dynamically scheduled processor already has many HW mechanisms to support multithreading –

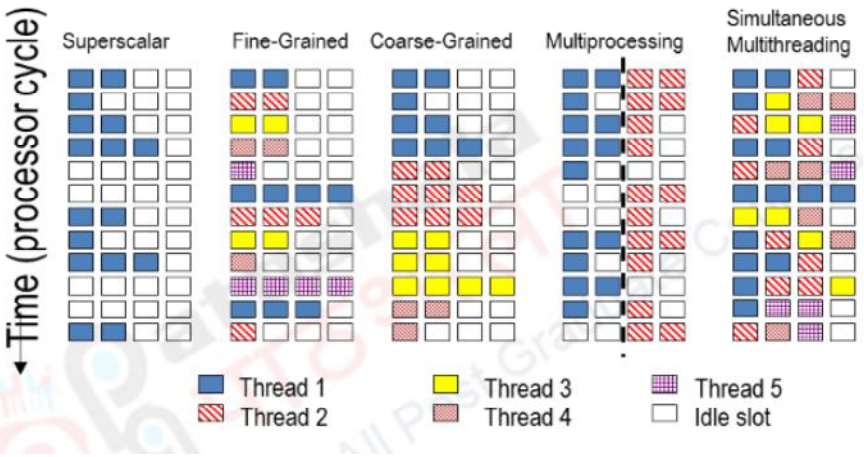
- a large set of virtual registers that can be used to hold the register sets of independent threads
- register renaming to provide unique register identifiers, so that instructions from multiple threads can be mixed in the data-path without confusing sources and destinations across threads and
- out-of-order completion that allows the threads to execute out of order, and get better utilization of the HW.

Thus, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them. The resolution of the dependences will be handled by the dynamic scheduling capability. We need to add a renaming table per thread and keep separate PCs. The independent commitment of each thread can be supported by logically keeping a separate reorder buffer for each thread. The diagram shows the difference between the various techniques.

In the superscalar approach without multithreading support, the number of instructions issued per clock cycle is dependent on the ILP available. Additionally, a major stall, such as an

instruction cache miss, can leave the entire processor idle. In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle, however, ILP limitations still lead to a significant number of idle slots within individual clock cycles. In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, within each clock cycle, the ILP limitations still lead to idle cycles.

Furthermore, in a coarse-grained multithreaded processor, since thread switching only occurs when there is a stall and the new thread has a start-up period, there are likely to be some fully idle cycles. In the SMT case, TLP and ILP are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors—including how many active threads are considered, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads—can also restrict how many slots are used.



The other option that we need to discuss to exploit TLP and ILP is Chip Multi Processors (CMPs). Instead of looking at a powerful processor that might be a dynamically scheduled superscalar with support for speculation and also SMT, can we look at a simpler processor, but multiples of them? That is what CMPs stands for – several processors on a single chip. Each processor canindividually support a thread of execution. Thus, with multiple processors, we have several threads of execution.

These processors can have both shared and distributed memory architectures and they may be made up of both homogenous and heterogeneous processor types. Having several processors on the same chip reduces the wire delays. Since the processors are just replicated in most of the cases (homogenous), the very long design and verification times needed for modern complicated processors is avoided. The difference between an SMT processor and a CMP can be summarized as follows:

SMT:

- Pool of execution units (wide machine)
- Several Logical processors

  – Copy of state for each of these logical processors

  – Multiple threads run concurrently

  – Better utilization and latency tolerance

CMP:

- Simple Cores

  – Moderate amount of parallelism

  – Threads are running concurrently on different cores

- Chip Multiprocessors integrate multiple processor cores on a single chip
- Eases the physical challenges of packing and interconnecting multiple processors
- This kind of tight integration reduces off-chip signaling and results in reduced latencies for processor-to-processor communication and synchronization.
- CMPs use relatively simple single-thread processor cores to exploit thread-level parallelism with one application by executing multiple threads in parallel across multiple processor cores.
- Allows a fairly short cycle time.
- Reduces the hardware overhead.
- Reduces power consumption.
- CMP is an ideal platform to run multi-programmed workloads or multithreaded applications. However, CMP architecture may lead to resource waste if an application cannot be effectively decomposed intothreads or there is not enough TLP.

RESULT:

Thus the process of above simulation is understood.

DATE: 19.02.2021	SIMULATION OF DATA LEVEL PARALLELISM
EX.NO: 08	

AIM:

To understand the Simulation of Data Level Parallelism.

THEORY:

Data parallelism (also known as **loop-level parallelism**) is a form of parallel computing for multiple processors using a technique for distributing the data across different parallel processor nodes. It contrasts to task parallelism as another form of parallelism.

In a multiprocessor system where each one is executing a single set of instructions, data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

For example, if we are running code on a 2-processor system (CPUs A and B) in a parallel computing environment, and we want to do a task on some data D, it is possible to tell CPU A to do that task on one part of D and CPU B on another part of D simultaneously (at the same time), in order to reduce the runtime of the execution.

Data parallelism is used by many applications especially data processing applications; one of the examples is database applications. Most real programs use a combination of data parallelism and task parallelism.

Data parallelismis parallelization across multiple processors in parallel computing environments. It focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel. It contrasts to task parallelism as another form of parallelism.

A data parallel job on an array of *n* elements can be divided equally among all the processors. Let us assume we want to sum all the elements of the given array and the time for a single addition operation is Ta time units. In the case of sequential execution, the time taken by

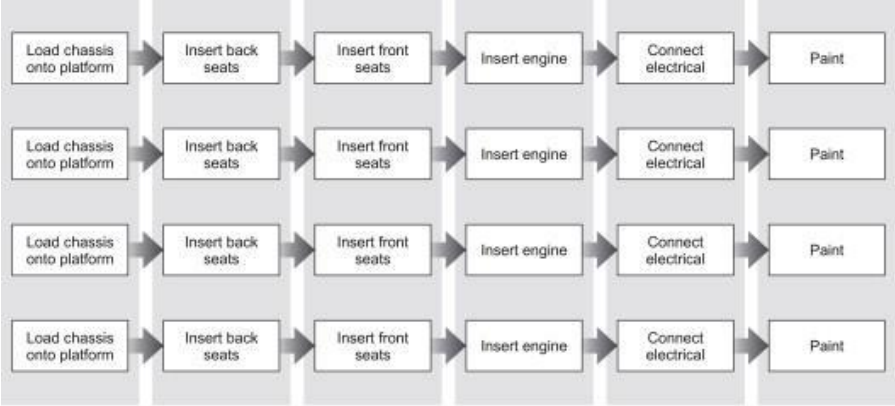
the process will be *n**Ta time units as it sums up all the elements of an array. On the other hand, if we execute this job as a data parallel job on 4 processors the time taken would reduce to (*n*/4)*Ta + merging overhead time units. Parallel execution results in a speedup of 4 over sequential execution. One important thing to note is that the locality of data references plays an important part in evaluating the performance of a data parallel programming model. Locality of data depends on the memory accesses performed by the program as well as the size of the cache.

Data parallelism is a different kind of parallelism that, instead of relying on process or task concurrency, is related to both the flow and the structure of the information. An analogy might revisit the automobile factory. There we looked at how the construction of an automobile could be transformed into a pipelined process.

Here, because the construction of cars along one assembly has no relation to the construction of the same kinds of cars along any other assembly line, there is no reason why we can't duplicate the same assembly line multiple times; two assembly lines will result in twice as many cars being produced in the same amount of time as a single assembly line.

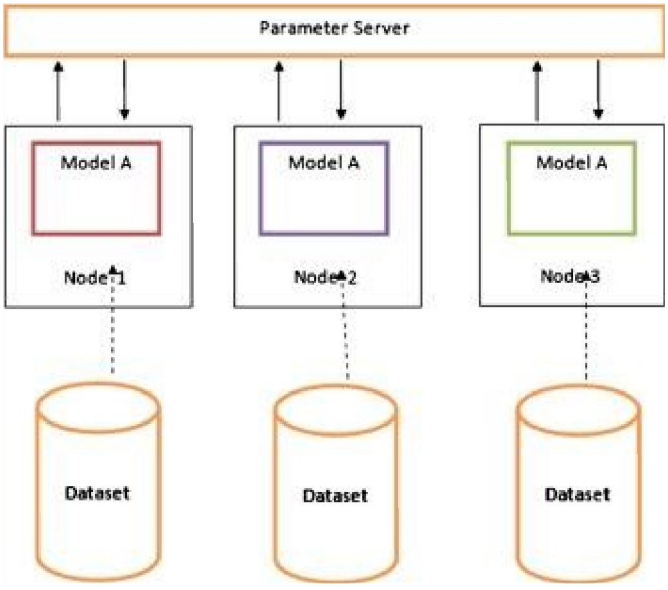
For data parallelism, the goal is to scale the throughput of processing based on the ability to decompose the data set into concurrent processing streams, all performing the same set of operations. For example, a customer address standardization process iteratively grabs an address and attempts to transform it into a standard form.

This task is adaptable to data parallelism and can be sped up by a factor of 4 by instantiating four address standardization processes and streaming one-fourth of the address records through each instantiation. Data parallelism is a more finely grained parallelism in that we achieve our performance improvement by applying the same small set of tasks iteratively over multiple streams of data.



Duplicating the assembly shows how the same operations can be executed on different cars in production, even in the same stage. If the cars correspond to partially completed data processing stages, this can be compared to data parallelism.

The idea of data parallelism was brought up by Jeff Dean style as parameter averaging. We have three copies of the same model. We deploy the same model A over three different nodes, and a subset of the data is fed over the three identical models. The values of the parameters are sent to the parameter server and, after collecting all the parameters, they are averaged. Using the parameter server, the omega is synchronized. The neural networks can be trained in parallel in two ways, i.e., synchronously (by waiting for one complete iteration and updating the value for omega) and asynchronously (by sending outdated parameters out of the network). But the amount of time taken for both methods is same, and the method choice is not a big issue here.



To overcome the problems in data parallelism, task level parallelism has been introduced. Independent computation tasks are processed in parallel by using the conditional statements in GPUs. Task level parallelism can act without the help of data parallelism only to a certain extent, beyond which the GPU needs data parallelism for better efficiency. But the task level parallelism gives more flexibility and computation acceleration to the GPUs.

Data parallel programming environments

A variety of data parallel programming environments are available today, most widely used of which are:

1. **Message Passing Interface:** It is a cross-platform message passing programming interface for parallel computers. It defines the semantics of library functions to allow users to write portable message passing programs in C, C++ and Fortran.
2. **Open Multi Processing (Open MP):** It's an Application Programming Interface (API) which supports shared memory programming models on multiple platforms of multiprocessor systems.
3. **CUDA and OpenACC:** CUDA and OpenACC (respectively) are parallel computing API platforms designed to allow a software engineer to utilize GPU's computational units for general purpose processing.
4. **Threading Building Blocks and RaftLib:** Both open source programming environments that enable mixed data/task parallelism in C/C++ environments across heterogeneous resources.

Applications

Data parallelism finds its applications in a variety of fields ranging from physics, chemistry, biology, material sciences to signal processing. Sciences imply data parallelism for simulating models like molecular dynamics, sequence analysis of genome data and other

physical phenomenon. Driving forces in signal processing for data parallelism are video encoding, image and graphics processing, wireless communications to name a few.

RESULT:

Thus the process of above simulation is understood.