**COMPUTER ORGANISATION**  
University at Albany  
Department of Computer Science  
ICSI 504

**PROJECT-3 REPORT**

**On**

Simulator for stack-based Duo core processor including cache MEMORY to each core

*Submitted By*

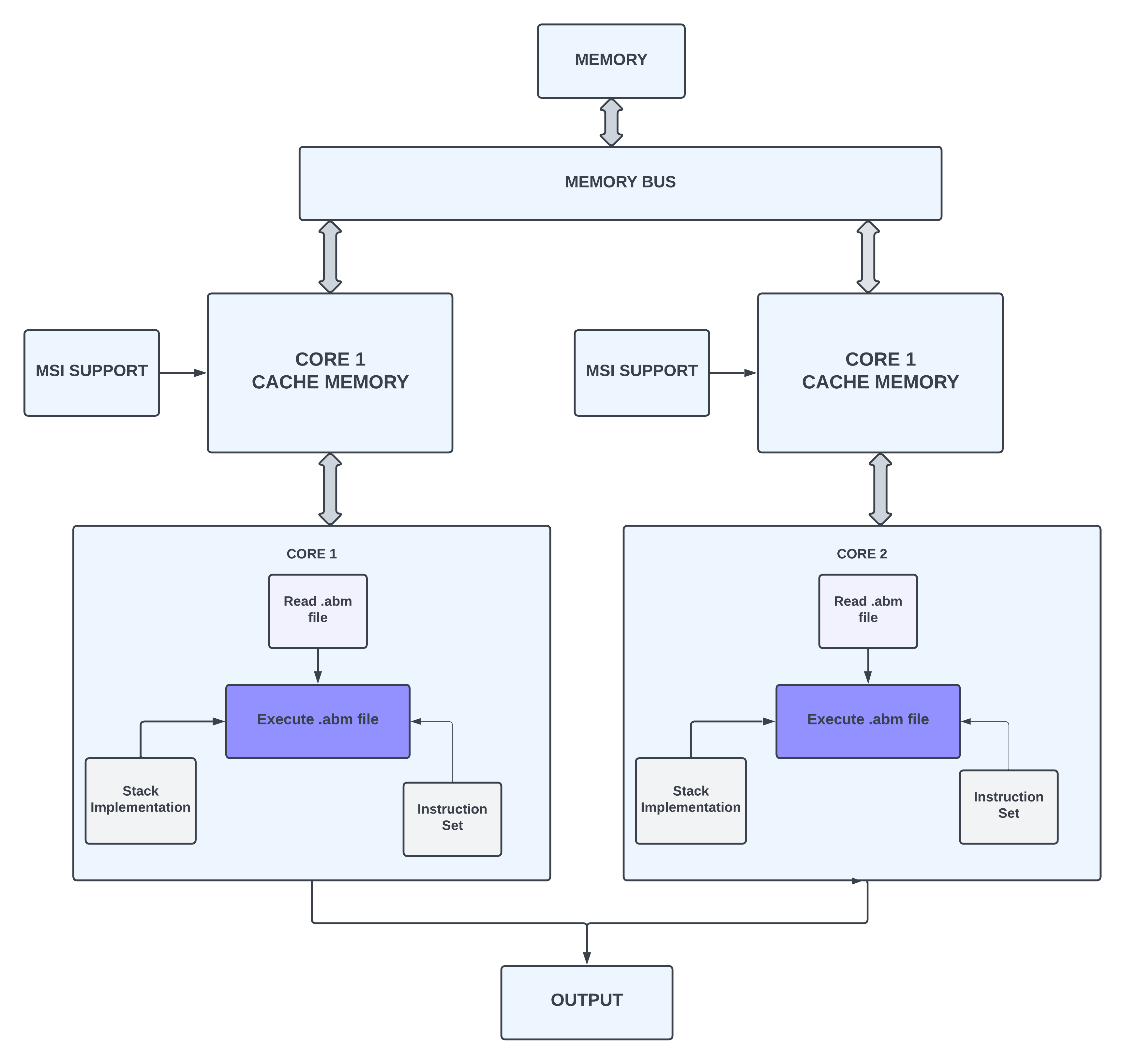
VIVEK GADDAM

UAlbanyID: 001605656

NetID: VG947964

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **TABLE OF CONTENTS** |  |
| **S.No.** | **TITLE** | | **PAGE No.** |
|  |  | |  |
| **1.** | **System Documentation** | | **3** |
|  | 1.1 | High-Level Data Flow Diagram | 3 |
|  | 1.2 | List of Routines | 4 |
|  | 1.3 | Implementation | 6 |
| **2.** | **Test Documentation** | | **8** |
|  | 2.1 | Testing the Program | 8 |
|  | 2.2 | Test Cases | 8 |
|  | 2.3 | Program Bugs and Resolution | 14 |
|  | 2.4 | Executable Version | 14 |
|  | 2.5 | Difficulties and Solutions | 14 |
| **3.** | **Algorithms and Data Structures** | | **15** |
|  | 3.1 | Algorithms Used | 15 |
|  | 3.2 | Data Structures Used | 15 |
| **4.** | **User Documentation** | | **17** |
|  | 4.1 | Operating System Used | 17 |
|  | 4.2 | How to Compile the Program | 17 |
|  | 4.3 | Other Applications Required | 17 |
|  | 4.4 | How to run the Program | 17 |
|  | 4.5 | Parameters | 17 |

1. **SYSTEM DOCUMENTATION**

****

**Figure 1: High Level Data Flow Diagram**

**1.1 High-Level Data Flow Diagram**

The High-Level Data Flow diagram in figure 1 above, depict the flow of the simulator for stack-based duo core processor. The stack is implemented for the functioning of the simulator. Stack functions are defined for its functioning. In a whole, the .abm file is read and the .abm file is executed using the stack, hash table and defined instruction set, that has the set of instructions defined to run and get the output of simulator. Later, the desired output is given for each .abm file on the terminal. The cache memory is included limited to store only 3 variables. MSI(Modified, Shared, Invalid) states have been shown along with cache memory.

**1.2 List of Routines**

**a. List of Routines for Stack implementation**

* Struct Stack

Created a user defined data type structure to define a stack. It holds the data size, initialised top in this.

* void startStack(struct Stack \*stack)

Used to create/start a stack when needed. A stack is created when this function is used in the program. Furtheroperations take place inthis stack.

* int stackFull(struct Stack \*stack)

Used to check if the stack is full or no. If stack is full, throws Stack Over follow.

* int stackEmpty(struct Stack \*stack)

Used to check if the stack is empty. If stack is empty, throws Sack Under Flow.

* pushStack(struct Stack \*stack, int val)

Used to push a value into the stack.

* int popStack(struct Stack \*stack)

Used to pop the top of stack. i.e. element in the top of stack.

* int peekStack(struct Stack \*stack)

Used to peek the topmost element in the stack.

**b. For StackSimulator\_2 Execution**

* int lineRead(char \*filename, char LineWord[MAX\_SIZE][100], char NextWord(MAX\_SIZE])
* fgets()

A line is read from the given stream and stored in the string pointed to by str. When (n-1) characters, the newline character, or the end of the file are read, or whichever occurs first, the program ends.

* strcmp()

Used to compare two null-terminated strings and returns an integer value indicating their relationship. Is 0 if they are equal.

* sscan()

Used for parsing formatted input from a character string, similar to scanf which reads from standard input, but sscanf reads from a given string.

**c. For Hash Table implementation**

* typedef struct AddressValue

This is structure for hash table pair.

* typedef struct Table

A structure table.

* unsigned int hash(int addr)

Function for hashing.

* Table \*create()

Function for creating the hash table.

* void insert(Table \*htable, int address, int value)

Function to insert value at a particular address.

* int retrieve(Table \*htable, int address)

Function to retrieve the value from a particular address.

* void changeValue(Table \*htable, int addr, int newVal)

Function to change the value at a particular address.

**d. For Queue Implementation**

* typedef struct CacheNode

This struct is to create node for the operation on queue like data to be put in the queue.

* typedef struct CacheQueue

This struct is for the creation of queue with its entities like front, rear, count.

* CacheNode\* newCacheNode(char\* dataEntered);

For creation of new Node when a new variable in to be stored.

* CacheQueue\* createCacheQueue(int VarCount);

Function for creation of queue.

* int CacheMemoryFull(CacheQueue\* CacheQueue);

Function to check if the queue as any element in it or not.

* int isCacheQueueEmpty(CacheQueue\* CacheQueue);

Function to check if cache is empty or not.

* void DeCacheQueue(CacheQueue\* CacheQueue);

Function to remove a variable from the queue.

* int VariableThere(CacheQueue\* CacheQueue, char\* dataEntered);

Function to check if particular variable is there in the queue.

* void EnCacheQueue(CacheQueue\* CacheQueue, char\* dataEntered);

Function to push the variable into queue.

* void CacheMemory(CacheQueue\* CacheQueue, char\* dataEntered);

Function to perform LRU and to store variable in the queue. That acts as a cache memory.

* void printCacheQueue(CacheQueue\* CacheQueue);

Function to print the variables that are in the cache memory.

* typedef enum

For creating struct for MSI state.

**1.3 Implementation**

To create a simulator for stack-based duo core processor with inclusion of cache memory in the respective cores, I used the structure data type. By using this I implemented a stack that is used for implementation and functioning of the duo core simulator. Also, implemented hash table, for optimised and effective functioning of the cores with respect to the data addresses. For the stack to function, I have written the functions such as push, pop, peek, isEmpty, isFull, as mentioned in the section above. Similarly, for hash table I have written functions such as insert, retrieve, changeValue, hash, as mentioned above.

For the functionality of cache memory, queue is implemented. All functions needed for operations are mentioned in the list of routines.For the functioning of cache memory, I have implemented queue, that acts as a cache memory. LRU (Least Recently Used) Algorithm has been implemented. Also, MSI (Modified, Shared, Invalid) cache coherence write back protocol has also been shown according to the operation in the cache memory. In the implementation, what ever variable is being used by lvalue and rvalue, that is, on lvalue or rvalue a variable is triggered and being used. Therefore, on these commands, the variables are stored in the cache memory. The memory limit is of 3 variables, therefore, when limit is reached, for storing the next variables, the last recently used variable is removed from the queue and then the new one is stored. If the same variable is being triggerd or used, then the memory order is changed according to LRU algorithm. The MSI state support is given by, when a variable is being removed from cache then, cache is in invalid state. When variable is being entered into the cache then its is modified. If variable is being used it is in shared state. These prompts are given in the terminal after and before each time cache memory is printed.

By help of this, a stack and hashtable is created and the operations that are needed for implementing and working of simulator this stack is used. I have built my simulator in Linux Operating System, using C programming Language. I have used VS Code for coding. Using my duo core simulator, the ABM instructions can be performed. In my project I have successfully executed the .abm file provided, that is test1forP2.abm.

Firstly, the cores, read the given .abm files. The abm filename is given to the program while compiling as the second argument. The program includes the simulator library, created for the efficient working of the simulator. It is named as Simulatordata.h. After the file is given, programs concurrently read the file in the respective cores and firstly, stores the file content into two different 2D arrays, by dividing them based on the instruction set’s commands and respective arguments. The functions of the simulator are defined in the Simulatordata.c.

Then the simulator performs the operations with help of stack and hashtable. The program holds all the definitions of each instruction set of abstact machine. Process is done in each core by reading each line after opening the arrays that have commands and arguments. In each line, the first word of the line is checked, it is compared to the words of instruction set. Therefore, for every match of first word in line to the words of instruction set, their respective operations are performed.

The memory bus is like a bridge in between the memory and the cores. The programs in cores access the memory via the memory bus. Each address, and the data given into the simulator or is being processed is stored in the memory. The data is then accessed by the processes in core, when needed via memory bus. Therefore, the program execution happens concurrently, here for the project, since there is one .abm file, both the cores process same file and give same output that is printed in the terminal. The Simulator has a header file that contains all the declarations of functions needed for the effective functioning of the simulator with duo core processor.

The operation and functionality, of each instruction is defined. The duo core simulator has the access to this instruction set and the program execution happens concurrently. The instructions are such as, for instruction “show” prints adjacent words in the line till the end of line. “push” pushes the adjacent value into the stack and so on.

All the operations have their respective functionality. For the instruction set, lvalue, the address of the target is sent into the stack, for “:=”, the particular value is assigned to a particular address in the hash table. And for rvalue, the data at the target location (address) in the hash table is retrieved and is pushed back into the stack. Other instruction sets such call, goto, return, label etc. are operating in the way required for the project. In the project, the address of one variable is manipulated with the help of another variable’s address. For address of one variable is updated or decremented by performing ‘+’ operation or ‘-‘operation. Then the value at that address is changed or modified.

To run two processes concurrently in two cores. I have implemented pipes to achieve that. I have created a fork ID. By forking the processes run concurrently, the core 1 waits after its process are done and then the second core output is seen. The cache memory works independently for both the cores. The output is shown for both the cores separately.

By this, the cores, are reading the same file at same time, and then the process is happening concurrently at same time. This increases the efficiency of the simulator. The same memory bus is being shared i.e. the hash table. And the data is being taken from same memory. To show the process are working, I have verified it by showing the outputs of both processes in the terminal.

The output is being printed in terminal. The output contains the expected output along with the cache memory and MSI state when a data variable is triggered. Further, using these instructions, queue, stack and hash table, the duo core with cache memory in each core, the simulator works. Finally, the output of the simulator is shown in terminal of respective .abm file with the cache memory and MSI state.

1. **TEST DOCUMENTATION**

**2.1 Testing the Program**

To test the program, the .abm files are given as input and output seen in terminal. Output of dual core processor along with cache memory content is shown.

* test1forP2.abm

**2.2 Test Cases**

a. Test Case 1: Tested test1forP2.abm file. Gave it as input and received output in terminal.

A screenshot of a computer

Description automatically generated(i) Below is the snapshot for the input file.

Figure 2: Input file: test1forP2.abm

(ii) Below is the output snapshot of the duo core, The output of two cores is shown along with cache memory after the program execution.

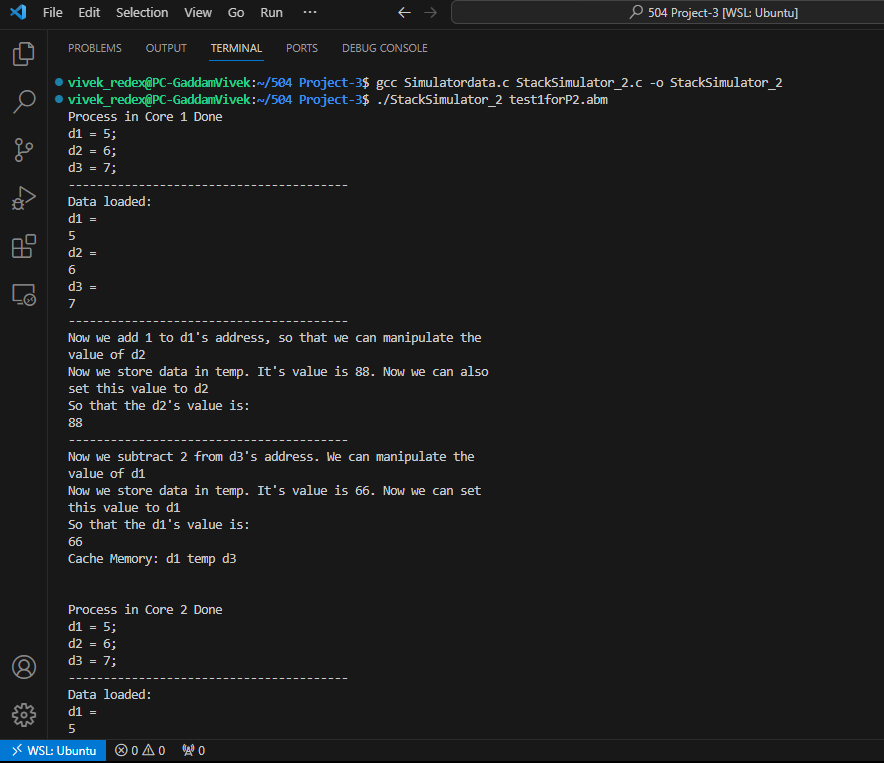


Figure 3: Output of test1forP2.abm in terminal (Core 1 Process)

(iii) Below is the snapshot of core 2 along with cache memory at the end of the program.

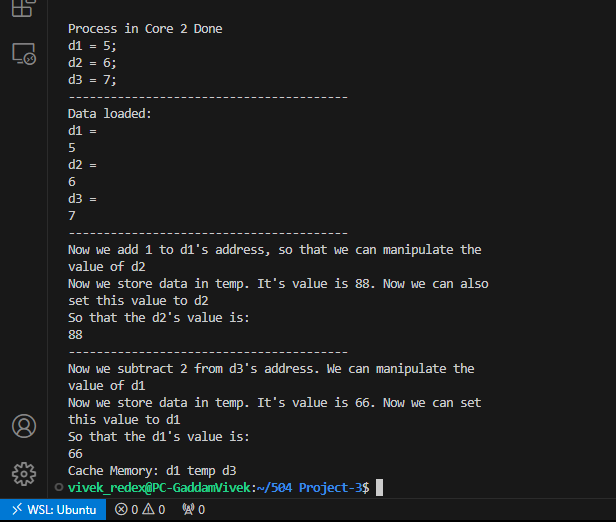


Figure 4: Output of input file (Core 2 Process)

(iv) Below is the snapshot of core 1 process along with cache memory each time variable is accessed or used.

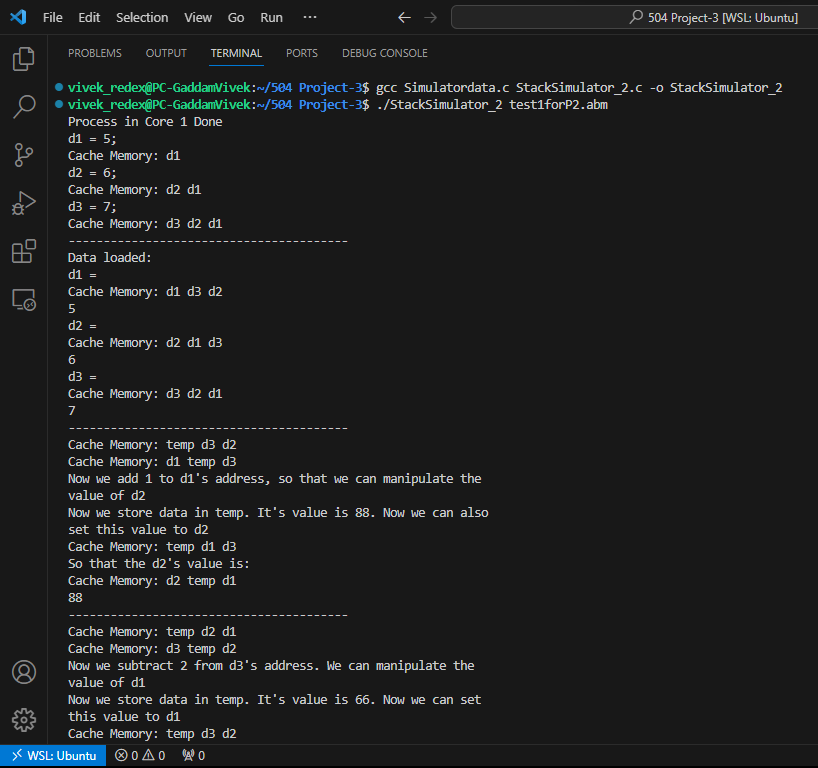


Figure 5: Output of core 1 process with cache memory

(v) Below is the snapshot of core 2 process along with cache memory each time variable is accessed or used.

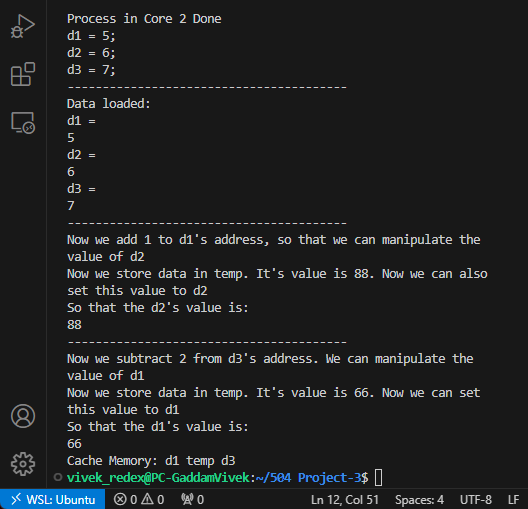


Figure 6: Output of core 2 process with cache memory

(vi) Below is the snapshot of program execution with MSI state support it each stage.

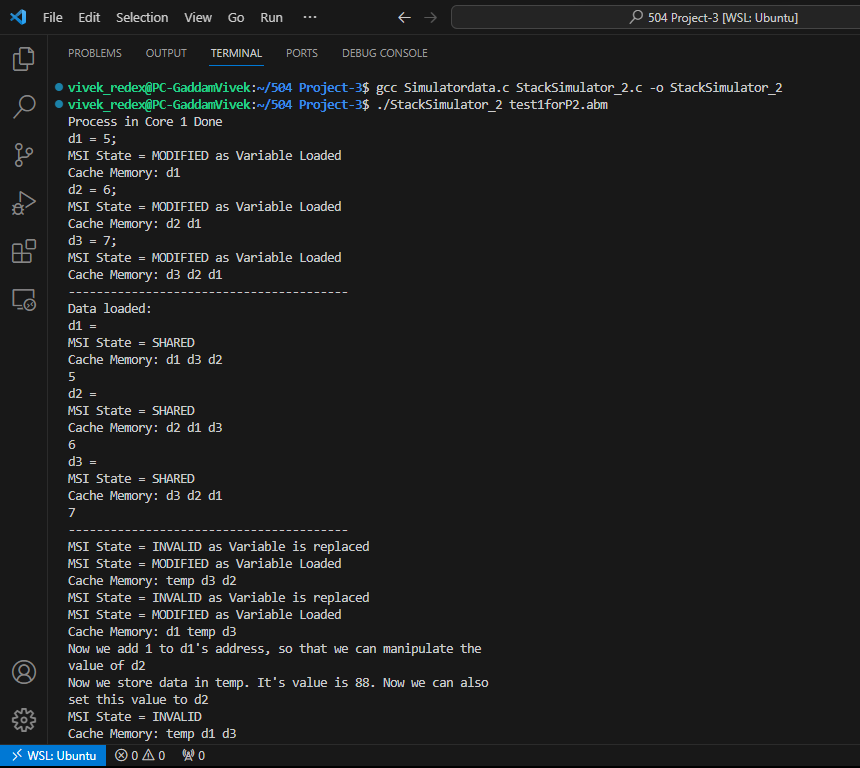


Figure 7: Output of program with cache memory and MSI State Support

**2.3 Program Bugs and Resolution**

Optimization is needed to improve the performance of the simulator is one of the important and crucial goals of this whole project. This section basically clarifies checking the code properly, again and again, to better find the places where the productivity of the whole code can be increased, including stack operations as well as memory management. They can be resolved by Handling errors, Testing of Instruction sets, and stack can optimise the system.

## 2.4 Executable Version

The particular users easily and efficiently can now run ABM codes and see executed outputs at a glance and this is only because of the development of an executable version of the ABM simulator. This simulator effectively and efficiently accepts ABM codes as input in the form of text files and displays the output results depending on the commands that are entered.

## 2.5 Difficulties and Solutions

There are some difficulties and issues in the time of creating the simulator to simulate the ABM codes effectively such as

* **Instruction Parsing Method**: Accurately and effectively parsing and understanding the various ABM codes and programs can be challenging sometimes during the usage of the ABM simulator. The answer includes a careful creation of a parser that could handle various operand and instruction forms.
* **Memory Management Process**: The stack and variable memory management basically require important and crucial thinking. In order to solve the problem of memory-related issues, some proper data structures and efficient error-handling methods are designed.
* **Testing Complicatency:** It is very difficult to make sure of the simulator's proper accuracy when using various input programs. The output entailed methodically developing many test scenarios and contrasting simulator outputs with anticipated outcomes.
* **Handling Errors:** It was very crucial and important to effectively handle the mistakes. The approach involves identifying errors in the code at important and required places and adding detailed error messages to help with debugging.

1. **ALGORITHM AND DATA STRUCTURES**

Algorithms and data structures play a role, in organizing computers determining the effectiveness and efficiency of processing information. These fundamental components serve as the building blocks for software and hardware interactions within a computer system.

**3.1 Algorithms**

Algorithms that are used to create the ABM simulator have detailed instructions and tasks to know how some particular activities and calculations are carried out by computer systems both with the hardware and software part. Algorithms that are used here are essential for maximizing system capabilities effectively, and resource management, including data processing in computer organization.

**a. LRU (Least Recently Used) Algorithm**

A caching algorithm called LRU, or least recently used, is frequently used. It works on the basis of removing the objects that haven't been accessed in the longest. This technique maximizes efficiency by keeping frequently used data in the cache by guaranteeing that the most recently accessed items stay there. To improve system performance, LRU is commonly used in many different applications, such as web browsers, databases, and operating systems. In order to effectively manage the limited cache capacity, the algorithm depends on monitoring the items' access history and making judgments based on the items' most recent access.

Particularly, I have basic implementation related to pipes, queue, stack, arrays, and hash table.

**3.2 Data Structures**

Data structures are tools to efficiently manage and store important and crucial data in a system’s memory of storage media. It offers a methodical, organized, and effective manner to collect and work with data.

**a.** **Stack**

In this case, for my project the stack is implemented using the struct data structure. The Last-In, First-Out (LIFO) principle, which states that the last element inserted is the first one withdrawn, is followed by a stack, a linear data structure. It is frequently employed in software programs to manage function calls, monitor program performance, and add undo/redo capabilities.

The struct data structure was chosen since the stack implementation required a few parameters. A structured data type, often known as a "struct" in the C programming language, enables programmers to combine many variables of various data types into a single user-defined data structure. This makes code more streamlined and organized by allowing the construction of intricate data structures that represent actual entities. The functions for stack implementation are written and stack can be used to operate the simulator.

**b. Hash Table**

For the project I have used hash table, A hash table, also known as a hash map, is a fundamental data structure designed for efficient data storage and retrieval. It consists of a collection of key-value pairs and relies on a hash function to map keys to specific locations within an array. Hash Tables are data structures that efficiently store and retrieve data by mapping keys to array indices using a hash function. They are an essential component of data storage and access in computer programming because of their versatility and widespread use in several applications.

**c. Queue**

A basic data structure in computer science that adheres to the First-In-First-Out (FIFO) concept is a queue. Elements in a queue are enqueued (added at the back) and dequeued (added at the front). This guarantees that the element that has been waiting in line the longest will be handled first. Sequential task management is a typical use case for queues, including print job scheduling, operating system process management, and breadth-first search methods. They are essential to the smooth and methodical completion of tasks by guaranteeing the efficient and well-organized processing of parts in a variety of applications.

1. **USER DOCUMENTATION**

**4.1 Operating System Used**

I have used Linus Operating System for my project. Further I have used VS Code for coding, therefore the output snapshots are of VS Code.

**4.2 How to Compile the Program**

Need to compile the program using gcc command in terminal. The GCC is basically a compiler that compiles the programs effectively.

**4.3 Other Applications**

If want to execute similarly in VS Code, need to install the VS Code software.

**4.4 How to run the program**

* Open Terminal
* Open the required directory where all files are stored.
* The executables have already been created.
* Compile the two program files.
* The programs can be compiled by the command.

-gcc Simulatordata.c StackSimulator\_2.c -o StackSimulator\_2

* Then run the program using below command.

./StackSimulator\_2 filename.abm

* Replace ‘filename’ with the name of the respective .abm file.
* Example, for test1forP2.abm : ./StackSimulator\_2 test1forP2.abm
* The output is seen on the terminal of both the processes(core 1 and core 2) along with cache memory and MSI state.

**4.5 Parameters**

No parameters used for the project.