

Computer Simulator: An Educational Tool for Computer Architecture

Mihyar Hesson

Al Ain University of Science and Technology, Al-Ain, P.O. Box: 64141, U.A.E.

Abstract: The great advancement in computer architecture and cache memory design and technology had a considerable influence on the way computer architecture was taught in universities. This requires students to be able to visualize the detailed activities that take place within a computer processor and its interaction with memory system. Computer simulators could effectively be used to enhance the understanding and comprehension of cache memory operation. The main objective of this project was to design and implement a computer simulator that was used as an educational tool. This paper presents design specifications, implementation and the functional and structural components of this simulator. This allows students understand the concepts and theory of the computer hardware topics by constructing and verifying knowledge, testing and comparing several different configurations and memory access. Although there was a large number of computer simulators in the market, this simulator differs in the way it contains a specially designed assembler that feeds the simulator with the binary code. In this context it was a tool that provides a high educational value that, on one hand, helps students learn to write an error-free assembly code and on the other hand comprehend the activities that take place during the execution of the program under different settings. At the front-end of the system there are two parts; the editor and the simulator while at the back-end there are the system specially developed assembler and database.

Key words: Computer architecture, simulation, assembly language, cache memory

INTRODUCTION

Computer simulators are used in education to help students understand and comprehend specific topics in computer architecture. Some of these simulators concentrate on helping students understand assembly language programming, while others concentrate on general operation of computer and data transfer between computer main components. More advanced simulators are dedicated to specific more complicated issues like cache memory operations. Large number of computer simulators of different degrees of complexity is available in the market, some of them are offered for free while the use of others requires special licenses. Examples of available simple to moderate types of simulators include Historic Machine Simulators, Digital Logic Simulators, Theoretical Machine Simulators, Novice Hypothetical Machine Simulators and Intermediate Instruction Set Simulators. Other more sophisticated simulators include Advanced Microarchitecture Simulators, Multi-Processor Simulators (including Multi-Processor Interconnection Network Simulators), Memory and Operating System Simulators, Embedded Processor Simulators and Quantum Computer Simulators.

With the advent of recent computer architecture issues such as burst-mode cache, victim cache and more complicated cache coherency issues, multiprocessors, parallel processing and complicated processors'

architectures, the need for matching sophisticated simulators becomes more demanding. This is not for education issues only but for development and improvement reasons too. In the recent years, large-scale distributed shared-memory (DSM) multiprocessors have emerged as a promising architecture to deliver high-performance computing power. However, to fully realize the potential performance of these systems, designers must solve two important and challenging problems. First, it is imperative that the cache coherence scheme for such systems be efficient, inexpensive and scalable. Second, it is necessary to develop efficient techniques to hide the large remote memory access latencies in such systems^[1]. The cache coherence techniques used in existing commercially available multiprocessors are mainly hardware-based, such as a snoopy cache protocol^[2] or a hardware directory-based scheme^[3,4].

However, in order to reduce the hardware complexity and/or increase the flexibility, many researchers have considered migrating the coherence protocol, or parts of it, to software^[5-8]. Shared virtual memory (SVM) systems go even further by completely supporting the protocol mechanisms at the operating system or application level using the virtual memory system^[9].

Trace-driven simulation is often a cost-effective way to estimate the performance of computer system designs. Above all when designing caches, Translation-

Lookaside Buffer (TLBs), or paging systems, trace-driven simulation is a very popular way to study and evaluate computer architectures, obtaining an acceptable estimation of performance before a system is built^[10].

From the author experience teaching computer architecture courses for a period of more than fifteen years in different universities in the region he identified two main problems that students were finding difficult to comprehend more than others. This seemed to have been consistent over the years. The first problem is related to programming in assembly language and the efficient use of processor's registers. The second problem is related to cache memory issues and operation such as mapping functions, write policies, replacement algorithms and cache coherence. This work is an attempt to develop a simulator that helps the students overcome these problems. This simulator is different from other available simulators in the market. It does not use memory traces but rather allows the user to enter his assembly program in an easy manner. The user then selects size of cache, mapping functions, write policies and replacement algorithm. The user can also select the speed at which the program runs so he/she can properly see how data are moved between different computer parts during execution and how the chosen settings affect execution efficiency.

THEORY

The theory of caching: The concept of the memory cache is simple. Data often moves from a slower medium, such as the file system, to a faster medium, such as main memory. Caching relies on the principle of locality of reference, that is, reading a datum predicts another read for the same datum in the near future. For example, if a browser requests an image on a Web page once, there will be a good chance that it will be requested again soon^[11]. A cache does speed up the process of fetching a datum on subsequent requests by providing a faster medium.

As the processor usually runs very fast and is constantly reading information from the memory, it has to wait for the information to arrive because the memory access times are slower. A cache memory therefore is like a small temporary fast memory that the processor uses for information it is likely to need again in the very near future. All the 5th generation processors now have cache memory that is actually built into the processor itself.

The purpose of cache memory: The processor is so much faster than other devices in the computer system and it has to spend a great deal of its time in waiting and this is a very inefficient use of the processor. This is mainly because of the slow memory access and therefore, in order to reduce the wait time of the

processor, memory access should be sped up. One way to achieve this is to reduce processor visits to main memory. This may be achieved by bringing the required data closer to the processor and this in turn is achieved by using the cache memory.

Looking at the relative speed-ups of CPUs and memories since 1980 we see that memory has increased in speed by a factor of about 4 while the CPUs by has increased by a factor of about 20000. The big mismatch, which continues to get even worse, means that CPUs are continually held back by slow memory. One obvious strategy that can help: if memory is too slow, don't use it too much. This strategy was used extensively in the early RISC processors which they often had hundreds of registers. The trouble with this technology is that it does nothing for instruction access. At least one memory access is required per instruction, in a pipelined processor, this means one per cycle.

A longstanding solution to this problem is the use of cache. Cache is a small block of high-speed memory (small enough to be affordable). The vast majority of references to instructions and data are local, that is, the next data item/instruction is usually close to the previous one. When a memory word is accessed, it will be searched first in the cache. If it is not there, a cache miss occurs and it has to read the word from the main memory through the cache.

Some methods, which are used to reduce the miss rate, increase complexity and consequently reduce speed. Therefore, in the tradeoffs between complexity and effectiveness, simplicity is much more favored by cache designers than virtual memory designers.

Figure 1 shows the relationship between main memory and a cache. Each slot in the cache can hold one block of contiguous memory words; a block is usually a specific number of main memory words (in Fig. 1 it is 3).

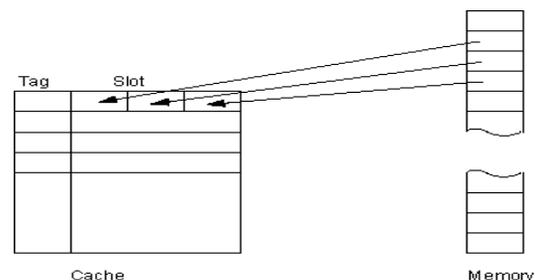


Fig. 1: Main memory and cache

The cache memory is accessed not via a memory address as such, but by pattern matching on a tag stored in the cache. The tag is constructed from the main memory address and means that a block may be stored in a vacant slot in the cache.

Mapping functions: Because there are fewer cache lines than main memory blocks, an algorithm is needed

for mapping main memory blocks into cache lines. Further, a means is needed to determine which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative and set associative^[12].

* **Direct-mapped caches:** This is the simplest mapping strategy. Each block of main memory can only map to a single slot in the cache. A simple example is shown in Fig. 2. The memory address is divided into three fields; the slot field, which is used to look up a particular slot in the cache, the tag field which is used to check if a particular block is in the cache and the word field which is used to identify the required word in the main memory^[13].

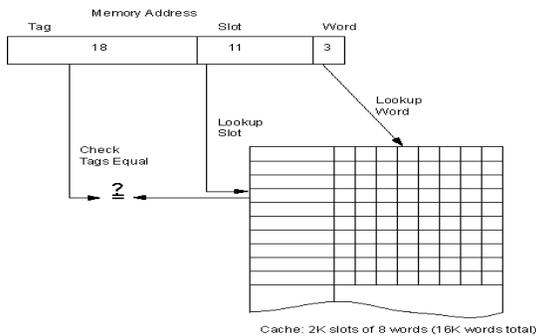


Fig. 2: Direct mapped cache

* **Associative cache:** The advantage of direct mapping is simplicity. However, it is inflexible and if two commonly-used blocks clash, it makes it very slow. This is because the need to keep swapping the two blocks in and out of cache. Associative mapping is the opposite extreme, i.e. any block of memory can map to any slot in the cache. This is illustrated in Fig. 3. It is obvious that apart from the word field, the remaining part of the address is used as a tag. There is no slot field because we do not actually lookup a particular slot. Associative cache is flexible, but expensive and/or slow since we need to simultaneously search all cache slots.

* **Set-Associative cache:** This mapping function represents a compromise between the previous two. It combines the simplicity of the first and the flexibility of the second and is meant to allow each block of memory to occupy one of a small set of slots of cache (typically 2 or 4). Figure 4 illustrates the basic idea with two-way mapping. Effectively, the main cache is divided into a number of smaller caches, each of which may contain a word or more properly a block from the main memory.

Cache line replacement algorithms: When a new line is loaded into the cache, one of the existing lines may need to be replaced. In a direct mapped cache, the requested block can go in exactly one position and the

block occupying that position must be replaced. In an associative cache the requested block can sit in any available cache slot^[14].

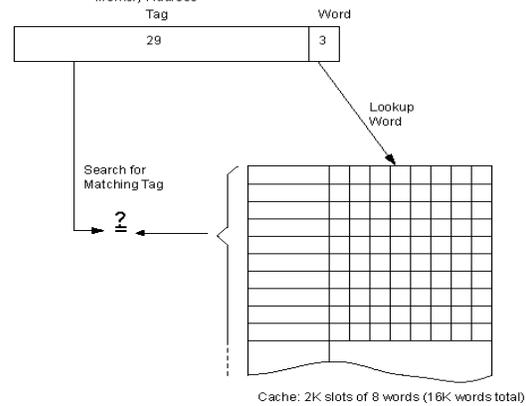


Fig. 3: Fully-associative cache

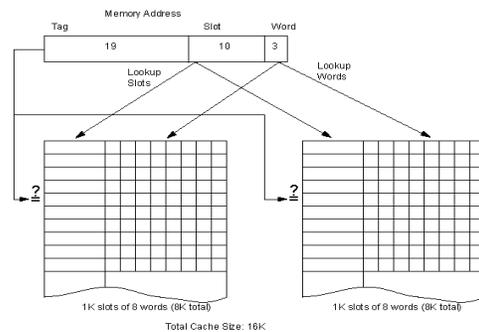


Fig. 4: Two-way set-associative cache

This means that all blocks are candidates for replacement. In a set associative cache, we must choose among the blocks in the selected set. Therefore a line replacement algorithm is needed which sets up well defined criteria upon which the replacement is made. A large number of algorithms are possible and many have been implemented. Four of the most common cache line replacement algorithms are: Least Recently Used (LRU), First-In First-Out (FIFO), Least Frequently Used (LFU) and Random

Cache write policies: Before a cache line can be replaced, it is necessary to determine if the line has been modified. The contents of the main memory block and the cache line that corresponds to that block are essentially copies of each other and should therefore hold the same data. If cache line X has not been modified since its arrival in the cache, updating the corresponding main memory block is not required. On the other hand, if the cache line has been modified, the corresponding main memory block must be updated. Basically there are two different policies that can be employed to ensure that the cache and main memory contents remain identical. These are: write-through and write-back.

* **Write-through:** Assuming a cache hit (a write hit); the information is written immediately to both the line in the cache and the block in the lower-level memory (with its normal wait-state delays). The advantages of this technique are that the contents of the main memory and the cache are always consistent. It is easy to implement and any read miss will never result in a write operation to main memory. On the other hand, the write through policy has a significant drawback. For every change in a cache line a main memory access is required and hence significantly degrades performance. In spite of this, most Intel microprocessors use a write-through cache policy.

* **Write-back** (sometimes called a posted write or copy back cache): On a cache hit, the information is written only to the line in the cache. This allows the processor to immediately resume processing. The modified cache line is written to main memory only when it is replaced. To reduce the frequency of writing back blocks on replacement, a dirty bit is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean the block is not written on a miss. The advantages of the write-back policy are that writes occur at the speed of the cache memory, multiple writes within a block require only one write to main memory, which results in less memory bandwidth usage. Write-back is a faster alternative to the write-through policy but it has one major disadvantage that comes from the possibility of the contents of the cache and the main memory may not be consistent^[15].

SYSTEM SPECIFICATIONS

This simulator is used for educational purposes and therefore is not meant to be used as performance measure for design purposes. The main intention behind this work was to help students understand the main concepts of computer architecture. This includes an easy way to code an error-proof assembly language program, to see how the program runs and how data moves between different parts of the computer including the CPU registers, main memory and cache memory. It also includes a means to easily configure the cache settings, control the speed at which the program runs, pause and resume execution at any time and to run program at a step-by-step mode. The simulator would provide the results of execution and also log all the events taken place during execution and store that in a file. The user could select different settings for the same program and see the effect of selecting different combinations of cache size, mapping functions, write policies and replacement algorithms on the results in term of hit/miss ratio. The system specifications may be summarized as:

- * An easy to use assembly language editor. The user is not required to type-in instructions. The user is using the mouse to select an instruction group such as data movements, arithmetic and logic etc. The user then selects an instruction from the group. Depending on the instruction type, the valid set of operands that go with the instruction will be displayed and the user is prompted to select the required operands. If the instruction takes only one operand, the list of operands will be deactivated right after the selection of the first operand. If the instruction takes no operands at all, no list will be displayed and the user is prompted for the next instruction. This insures that only the right instruction syntax will be allowed. This is of an additional educational value too as it teaches the users of common mistakes they usually commit.
- * The user is not to worry about address allocation, main memory and stack allocations.
- * Once a program is complete and assembled, the user is prompted for more choices. These include the size of main memory block (or cache line size), the mapping function, replacement algorithm (if applicable) and write policy. The user is also to select the speed of running to suite his/her capability of following up data movement. If the execution is too slow or too fast to follow, the user can stop the execution, adjust the speed and run again. This may be repeated a few times until the user convenient speed is reached. The user can also select a step-by-step execution instead of continuous running.
- * The user is able to easily watch the cache operation and how and when cache hits and misses occur.
- * The system logs all the events taken place during the execution of a program and stores that in a file that is available to the user to examine at any later time.
- * The result of execution in terms of hits per misses ratio is provided.

SYSTEM DESIGN AND IMPLEMENTATION

The system is architected to have four functionally-related and loosely coupled units. These units are the keypad editor, the assembler, the database and the simulator. The state chart of the system is shown in Fig. 5.

The keypad editor is used to enter assembly language programs. The mnemonics of instructions are immediately stored in the database line-by-line and at the same time it is displayed on the right-side of the editor. When the program is completely keyed-in, it is assembled by the assembler and stored in binary (executable) form in the database. The simulator loads the assembled program in its executable form and runs

it according to the configuration set by the user. After execution, the simulator stores a log file in the database. This file contains details of all events took place during the execution. In the following subsections, we will be describing these units in a bit more details.

The keypad editor: The user does not need to remember instructions nor has to remember the right format of writing instructions. Instructions are supplied and the user needs only to select the required instruction. If the user enters a wrong instruction format, the system will reject it. Only legal instruction format is accepted by the system. For example, if the user enters only one operand for an instruction that takes two operands, the system will not allow him/her to continue with the next instruction and so forth. As the number of available instructions is too large to fit in the limited space of the editor, these instructions are divided into five groups:

- * Data transfer group.
- * Arithmetic and logical group.
- * String manipulation group.
- * Control transfer.

Processor control group.

The operation of the keypad editor is shown in the state chart of Fig. 6.

Once the system is started, the user is prompted to select an instruction set group, one from the groups list stated above. All instructions from the selected group will be displayed. When selects an instruction, the user will be prompted for the exact number of operands that are required by the instruction. Every instruction displayed in will be automatically displayed on the main display. The system determines the length of the current instruction and hence calculates the start of the next instruction. Figure 7 shows a screen shot of the keypad editor when the control transfer instruction group is selected. It is useful to mention here that if other group is selected only the instructions within that group will be displayed. Figure 8 is another screen shot of the main display where the mnemonics of instructions are displayed. When the program is completely entered, it will be moved to the assembler where it is converted to binary, all links are resolved and then is converted to an executable form. In the next subsection we will be briefly talking about the assembler.

The assembler: There are a large number of commercially available assemblers, many of them are offered for free. The problem with these assemblers is the platform dependency, that is, they generate code plus extra information that is suitable to the underlying software and hardware. For the purpose of this simulator, this extra information will create problems.

In order to provide the pure binary code for the simulator we had to choose between two solutions. The first is to use one of the available assemblers and perform clean up operation on the generated code. The second is to develop our own assembler. Although the

second choice needed hard work to accomplish but nevertheless it always generates the clean code that the first choice would not guarantee. This assembler, which is regarded as one major part of this work is based on Intel x86 processors. The Intel processors are used in the majority of PCs and are also the subject in a great number of computer architecture textbooks.

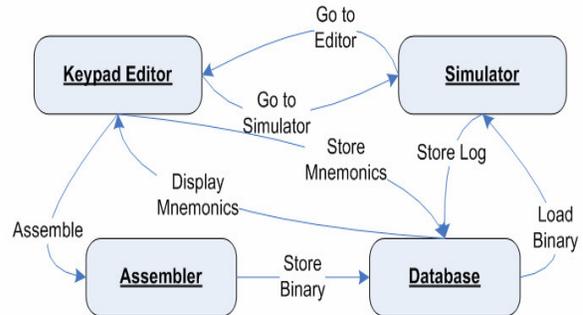


Fig. 5: The system state chart

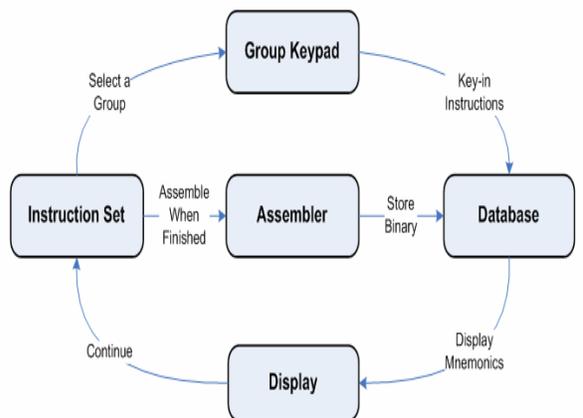


Fig. 6: The keypad state chart

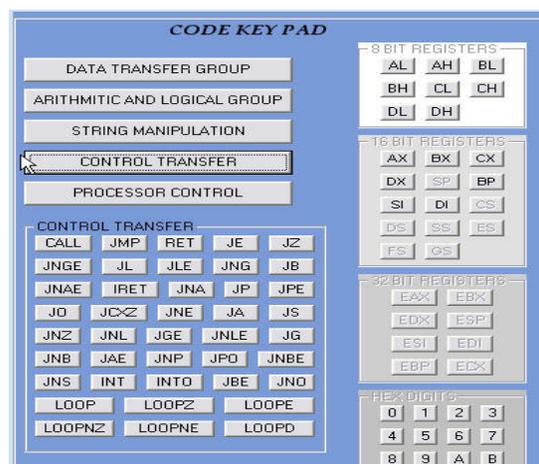


Fig. 7: Control transfer instruction group

This assembler is good for programs written for all x86 processors from 80286 to Pentium 4. The user can code programs using 8-bit, 16-bit and/or 32-bit registers.

The Database: As described in the previous subsection regarding the specially developed assembler, a database is required for assembling purposes, that is, converting assembly language programs to binary executable forms. Programs and log files are also stored. The user can at any time delete these files. However, the database conversion tables are inaccessible by the users and can not be altered or modified by them.

The simulator: The simulator contains a variety of displays, functionalities and controls. Figure 9 shows a block diagram of the simulator part. The main memory displays the binary program in hexadecimal values. At the start of execution, the user is asked to select the size of the main memory block which is the same size of the cache line. This is used in matching calculations between the main memory and the cache. Before running the program, the user has to select the mapping algorithm and the write policy. The program may be run in normal mode or in a step-by-step fashion. In normal mode, the user may pause and resume execution at any time. The user may also select the speed of running so he/she can comfortably see the transfer of data between processor, cache and main memory. This is displayed as a dynamic flowchart as seen in Fig. 10. The current action is displayed in red so the user can easily follow the program execution. On the other hand, the user may prefer to run the program in single steps. This gives more time to the user to see how the program runs and how data is moved. The execution controls are shown in Fig. 11. A log window that displays the actions taken to run the program is also given. This gives the user a detailed textual description of what has happened during the execution of the program. This is shown in Fig. 12.

The principle operation of this tool is as follows. The CPU fetches the first instruction from the main memory and put it the cache. Normally, the CPU brings more than one instruction so next time it does not need to go to the main memory. Next, the CPU reads from the cache and if the instruction or data requested is available in the cache it will take it from there. This represents a hit operation. On the other hand if the requested instruction or data is not in the cache, the CPU will bring another block from the main memory and put it in the cache. This represents a miss operation.

The cache replacement is dependant on the selected mapping function. If the Direct mapping is selected, there will be no choice as this is a one-to-one mapping technique. This means that each word in the main memory should sit in a specific place in the cache and if this place is occupied it will be overwritten. If the 2-Way Associative mapping is used, for instance, there will be two possible places in the cache for each word in the main memory. If one of them is occupied, the second one is used. If both of them are occupied, one of these positions will be overwritten. The replacement will be carried out according to the selected

replacement algorithm. The available replacement algorithms are stated in a previous section of this article.

The CPU normally operates on the data in the cache and may or may not change it. If no change takes place, this will cause no problem when the data in the cache is replaced. However, if the data in the cache is changed, this will result in two different values for the same variable, one in the main memory and the other in



Fig. 8: The program editor

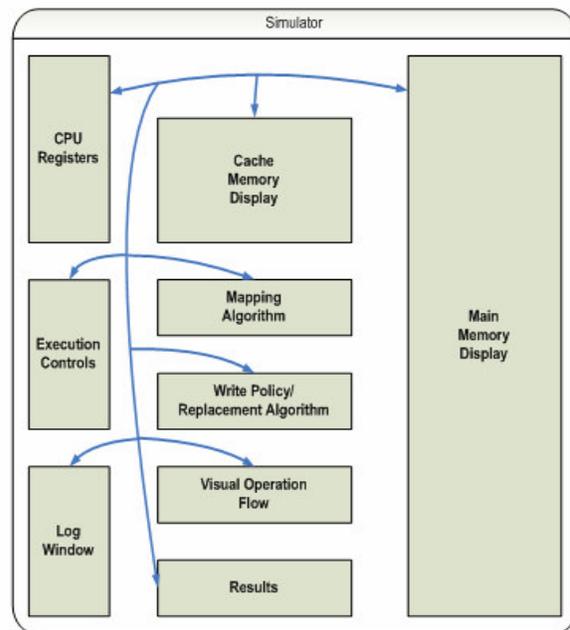


Fig. 9: The simulator block diagram

the cache. This discrepancy in values must be resolved in order to avoid potential problems.

The way the CPU resolves this problem depends on the user selection of write policy. If write through

was selected, any change in data in the cache is reflected in the main memory immediately. This might create traffic overhead on the system busses but is necessary if the main memory is used by more than one processor or another input/output device such as the DMA. However, if the write back policy was selected, the change in the memory is only done when the changed data in the cache is replaced. This reduces the traffic overhead but may create the risk of data inconsistency.

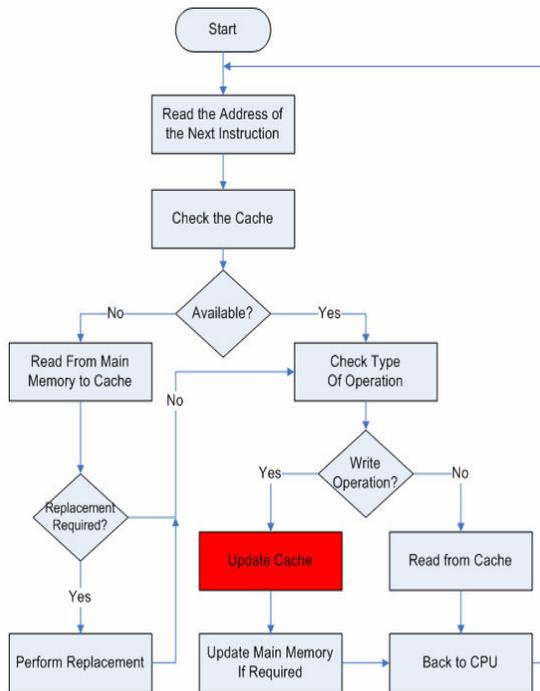


Fig. 10: The dynamic flowchart of the simulator

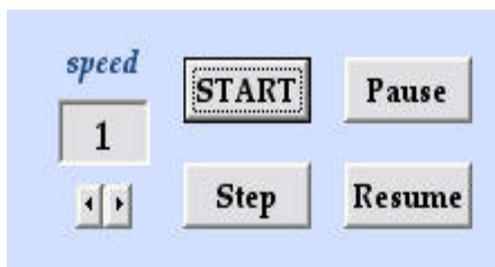


Fig. 11: Program execution controls

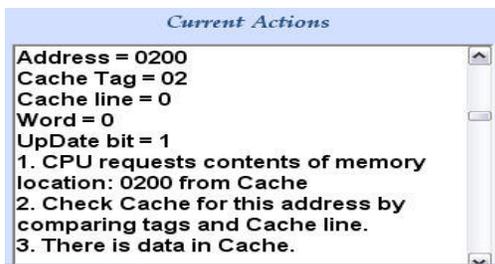


Fig. 12: Action logging window

CONCLUSION

The purpose of this work was to develop an educational tool to help describing the main architectural concepts of a computer system. This tool is of a great help to computer engineering and computer science students who study the computing systems or computer architecture and organization courses.

A large number of computer simulators of different degrees of complexity is available. Examples of available simulators include Digital Logic Simulators, Theoretical Machine Simulators, Intermediate Instruction Set Simulators, Advanced Microarchitecture Simulators and Multi-Processor Simulators (including Multi-Processor Interconnection Network Simulators). Trace-driven simulation is often a cost-effective way to estimate the performance of computer system designs. It is a very popular way to study and evaluate computer architectures, obtaining an acceptable estimation of performance before a system is built. Simulators of this type usually require memory traces in special text format to operate.

From the author experience teaching computer architecture courses for many years, two main problems were identified where students find difficult to comprehend more than others. These problems seemed to have been consistent over the years. The first problem is related to the difficulties of programming in assembly language and the inefficient use of processor's registers. The second problem is related to cache memory issues and operation such as mapping functions, write policies, replacement algorithms and cache coherence.

A computer simulator was developed to address these two issues. The simulator is equipped with easy-to-use graphical user interface. It contains an Intel-based assembly language programming editor that allows the user to select instructions rather than typing them. The total number of instructions is rather big and therefore any attempt to display all the available instructions on one screen makes it very difficult for the user to use. Instead, instructions are divided into functionally related groups. Once a group is selected, all the instructions within that group will be displayed. The editor automatically determines the number of operands needed for each particular instruction and will not allow an illegal instruction format to be entered. This, in fact, serves as a programming educator and it is, in this context, representing a valuable educational tool. Memory allocation and addressing is automatically dealt with by the system. Programs' files are stored in the database in two formats, the source (mnemonics) and in binary executable forms too. A specially designed assembler is used. The reason for that is no commercially available assembler would generate the pure binary code that is understood by the simulator.

Once a program is coded and assembled, a second user interface will be displayed. This screen contains all the controls that allow the user to define the simulator settings. This includes memory block size, cache size, mapping function, write policy and replacement algorithm. It also includes the execution modes. Two modes are available the first mode is the normal mode where the user can pause and resume at any time as well as controlling the speed of execution. A flowchart that dynamically shows the movement of data between processor, cache and main memory is provided. The screen also shows displays of the current contents of the main memory, cache memory and CPU registers. A window that logs all events during program execution is also provided.

Although this tool represents a great help to users, it still has a room for more changes and improvements. This simulator does not provide the full set of instructions and does not allow entering more complicated programs. Future work can improve the simulator to cover the full set of instructions and more complicated combination of these. Also, the simulator can be improved to cover more complex issues of cache memory such as burst-mode cache, victim cache and more complicated cache coherency issues.

ACKNOWLEDGEMENT

The author would like to express his sincere appreciation to the Research Affairs at the United Arab Emirates University for the financial support of this project. The author would also like to express his gratitude to Moza Mohammed, Hamda Al-Awar, Maryam Al-Aryani, Asma Mohd Al-Nayadi and Mariam Mohammed Al-Raisi for the valuable help.

REFERENCES

1. Lim, H.-B. and P.-C. Yew, 2001. Efficient integration of compiler-directed cache coherence and data perfecting. *J. Parallel and Distributed Computing*, 61: 1775-1802.
2. Papamarcos, M. and J. Patel, 1984. A low-overhead coherence solution for multiprocessors with private cache memories. *Proc. 11th Intl. Symp. Computer Architecture*, pp: 348-354.
3. Laudon, J. and D. Lenoski, 1997. The SGI origin: A CCNUMA highly scalable server. *Proc. 24th Intl. Symp. Computer Architecture*, pp: 241-251.
4. Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, 1990. The directory-based cache coherence protocol for the DASH multiprocessor. *Proc. 17th Intl. Symp. Computer Architecture*, pp: 148-159.
5. Agarwal, A., R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiawicz, B.-H. Lim, K. Mackenzie and D. Yeung, 1995. The MIT Alewife machine: Architecture and performance. *Proc. 22nd Intl. Symp. Computer Architecture*, pp: 2-13.
6. Chaiken, D. and A. Agarwal, 1994. Software-extended coherent shared memory: Performance and Cost. *Proc. 21st Intl. Symp. Computer Architecture*, pp: 314-324
7. Chiou, D., B.S. Ang, R. Greiner, Arvind, J.C. Hoe, M.J. Beckerle, J.C. Hoe, M.J. Beckerle, J.E. Hicks and A. Boughton, 1995. StarT-ng: Delivering seamless parallel computing. *Proc. EURO-PAR'95. Lecture Notes in Computer Science*, No. 966, pp. 101-116, Springer-Verlag, Berlin.
8. Grahm, H. and P. Stenstro, 1995. Efficient strategies for software-only directory protocols in shared-memory multiprocessors. *Proc. 22nd Intl. Symp. Computer Architecture*, pp: 38-47.
9. Grahm, H. and P. Stenstroem, 2000. Comparative evaluation of latency tolerating and reducing techniques for hardware-only and software-only directory protocols. *J. Parallel and Distributed Computing*, 60: 807-834.
10. Angel, M., J. Manuel and J. Antonio, 2001. An educational tool for testing caches on symmetric multiprocessors. *Microprocessors and Microsystems*, 25: 187-194.
11. Leon Atkinson, Caching, Zend, the php company, <http://www.zend.com/zend/trick/tricks-dec-2001.php>
12. Stalling, W., 2003. *Computer Organization and Architecture, Designing for Performance*. 6th Edn. Prentice Hall.
13. University of Swansea, Department of Computer Science, Staff and Student Server, Cache Systems, <http://www.cs.swan.ac.uk/~csneal/HPM/cache.htm>
14. Popov, D., E. Foutekova, I. Delchev, I. Krivulev and Z. Kochovski, 2003. Cache Memory Implementation And Design Techniques, <http://www.faculty.iu-bremen.de/birk/lectures/PC101-2003/07cache/cache%20memory.htm>
15. Phil Storrs PC Hardware book, Cache Memory Systems, Phil Storrs, December 1998.