GLOBAL JOURNAL OF MATHEMATICAL SCIENNCES VOL. 6, NO. 2, 2007: 97 - 103
COPYRIGHT (C) BACHUDO SCIENCE CO. LTD. PRINTED IN NIGERIA. ISSN 1596-6208

VON NEUMANN ARCHITECTURE AND MODERN COMPUTERS

I. I. ARIKPO, F. U. OGBAN and I. E. ETENG

(Received 23 July 2007, Revision Accepted 31 August 2007)

ABSTRACT

Computers with von Neumann architecture are those computers with single memory for both data and instructions, as well as sequential execution of instructions; as against the Harvard Scheme with separate memories for data and instructions, respectively. We took a critical look at the original architecture as proposed by John von Neumann. We considered an evolutionary perspective of the von Neumann architecture, with special interest in his greatest contribution – the Stored-Program Concept. The basic design of the von Neumann scheme at inception and the scheme in modern computers is also discussed. We discussed the fundamental features of the von Neumann architecture, the consequences of these characteristics, bottlenecks of this architecture, as well as ameliorative measures included in modern computers to handle these bottlenecks. We also presented different parallel architectures as enhancements to the von Neumann scheme and proposed alternative architectures that will cater for the shortcomings of the former. The von Neumann architecture remains the cornerstone of the architecture of modern computers and a complete extinction of this architecture may not take place in the near future.

KEYWORDS: von Neumann Architecture, Harvard Architecture, Evolution of Computers, Self-identifying Data Architecture.

1.0 INTRODUCTION

The term von Neumann Computer has two common meanings. Its strictest definition refers to a specific type of computer organization, or "architecture", in which instructions and data are stored together in a common memory (Lilja et al., 1998). This type of architecture is distinguished from the Harvard Architecture in which separate memories are used to store instructions and data (Lilja et al., 1998). The term von Neumann Computer is also used colloquially to refer in general, to computers that execute a single sequence of instructions, which operate on a single stream of data values. That is, von Neumann computers are the typical computers available today (Lilja et al., 1998).

Any discussion of computer architectures, that is, how computers and computer systems are organized, designed, and implemented, inevitably makes reference to the von Neumann architecture as a basis for comparison. And of course this is so, since virtually every electronic computer ever built has been rooted in this architecture (Riley, 1987).

2.0 THE VON NEUMANN COMPUTER ARCHITECTURE

The heart of the von Neumann computer architecture is the Central Processing Unit (CPU), consisting of the Control Unit and the Arithmetic and Logic Unit (ALU). The CPU interacts with a memory and an input/output (I/O) subsystem and executes a stream of instructions (the computer program) that process the data stored in memory and perform I/O operations (Lilja et al., 1998). The key concept of the von Neumann architecture is that data and instructions are stored in the memory system in exactly the same way (Lilja et al., 1998). Thus, the memory content is defined entirely by how it is interpreted. This is essential, for example, for a program compiler that translates a user-understandable program into the instruction stream understood by the machine. The output of the compiler (object program) is like ordinary data. However, these data (object program) can then be executed by the CPU as instructions.

A variety of instructions can be executed for moving and modifying data, and for controlling which instructions to execute next. This entire group of commands that the CPU understands and can react to is called the instruction set or command set (Rosch, 1999). The instruction set together with the resources needed for their execution is called the instruction set architecture (ISA). The instruction execution is driven by a periodic clock signal. Although several substeps have to be performed for the execution of each instruction, sophisticated CPU implementation technologies exist that can overlap these steps such that, ideally, one instruction can be executed per clock cycle (Rosch, 1999).

3.0 EVOLUTION OF THE VON NEUMANN COMPUTER ARCHITECTURE

3.1 Computer Technology Before The Electronic Computer

Ideas of an Analytical Machine to solve computing problems date back to Charles Babbage around 1822 with simple pegged-cylinder automata dating back even significantly further (Randell, 1994). Babbage described four logical units for his machine concept: memory, input/output, arithmetic units, and a decision mechanism based on computation results. The latter is a fundamental concept that distinguishes a computer from its simple sequencer predecessors. While Babbage's machine had to be constructed from mechanical building blocks, it took almost 100 years

I. I. Arikpo, Department of Mathematics/ Statistics and Computer Science, University of Calabar, Calabar, Nigeria
F. U. Ogban, Department of Mathematics/ Statistics and Computer Science, University of Calabar, Calabar, Nigeria
I. E. Eteng, Department of Mathematics/ Statistics and Computer Science, University of Calabar, Calabar, Nigeria

before his ideas were realized with more advanced technology such as electromechanical relays (e.g., the Bell Labs Model 1 in 1940) and vacuum tubes (e.g., ENIAC in 1946) (Randell, 1994).

3.2 The Birth of Electronic Computers

ENIAC, the Electronic Numerical Integrator And Computer, is considered to be the first true general-purpose electronic computer. It was built from 1944 through 1946 at the University of Pennsylvania's Moore School of Electrical Engineering (Stern, 1981). The leading designers were John Presper Eckert Jr. and John William Mauchly. ENIAC included some 18,000 vacuum tubes and 1,500 relays. Addition and subtraction were performed with 20 accumulators. There also was a multiplier, a divider, and square root unit. Input and output was given in the form of punch cards. An electronic memory was available for storing tabular functions and numerical constants. Temporary data produced and needed during computation could be stored in the accumulators or punched out and later reintroduced.

The designers expected that a problem would be run many times before the machine had to be reprogrammed. As a result, programs were hardwired in the form of switches located on the faces of the various units. This expectation, and the technological simplicity driven by War-time needs, kept the designers from implementing the more advanced concept of storing the instructions in memory (Stern, 1981).

3.3 Von Neumann's Contribution

John von Neumann was born in Hungary in 1903. A chemical engineer and mathematician by training, his well-respected work in the U.S.A., which was centered around physics and applied mathematics, made him an important consultant to various U.S. government agencies (Aspray, 1990). He became interested in electronic devices to speed-up the computations of problems he faced for projects in Los Alamos during World War II. Von Neumann learned about ENIAC in 1944 and became a consultant to its design team. His primary interest in this project was the logical structure and mathematical description of the new technology. This interest was in some contrast to the engineering view of Eckert and Mauchly whose goal was to establish a strong commercial base for the electronic computer (Aspray, 1990).

The Development of EDVAC, a follow-up project to ENIAC, began during the time that von Neumann, Eckert, and Mauchly were actively collaborating. At this time, substantial differences in viewpoints began to emerge. In 1945, von Neumann wrote the paper "First Draft of a Report on the EDVAC", which was the first written description of what has become to be called the von Neumann stored-program computer concept (Aspray, 1990), and (Godfrey and Hendry, 1993). The EDVAC, as designed differed substantially from this design, evidencing the diverging viewpoints. As a result, von Neumann engaged in the design of a machine of his own at the Institute for Advanced Study at Princeton University, referred to as the IAS computer (Godfrey and Hendry, 1993).

3.4 The Stored-Program Concept

Given the prior technology of the Babbage machine and ENIAC, the direct innovation of the von Neumann concept was that programs no longer needed to be encoded by setting mechanical switch arrays. Instead, instructions could be placed in memory in the same way as data (Godfrey and Hendry, 1993). It is this equivalence of data and instructions that represents the real revolution of the von Neumann idea.

One advantage of the stored program concept as envisioned by von Neumann was that instructions now could be changed quickly which enabled the computer to perform many different jobs in a short time. However, the storage equivalence between data and instructions allows an even greater advantage: programs can now be generated by other programs. Examples of such program-generating programs include compilers, linkers, and loaders, which are the common tools of a modern software environment (Godfrey and Hendry, 1993). These tools automate the tasks of software development that previously had to be performed manually. In enabling such tools, the foundation was laid for the modern programming system of today's computers.

4.0 ORGANIZATION AND OPERATION OF THE VON NEUMANN ARCHITECTURE

As shown in Figure 1, ideally, the heart of a computer system with a von Neumann architecture is the CPU. This component fetches (i.e., reads) instructions and data from the main memory and coordinates the complete execution of each instruction. It is typically organized into two separate subunits: the Arithmetic and Logic Unit (ALU), and the control unit. The ALU combines and transforms data using arithmetic operations, such as addition, subtraction, multiplication, and division, and logical operations, such as bit-wise negation, AND, and OR. The control unit interprets the instructions fetched from the memory and coordinates the operation of the entire system. It determines the order in which instructions are executed and provides all of the electrical signals necessary to control the operation of the ALU and the interfaces to the other system components.

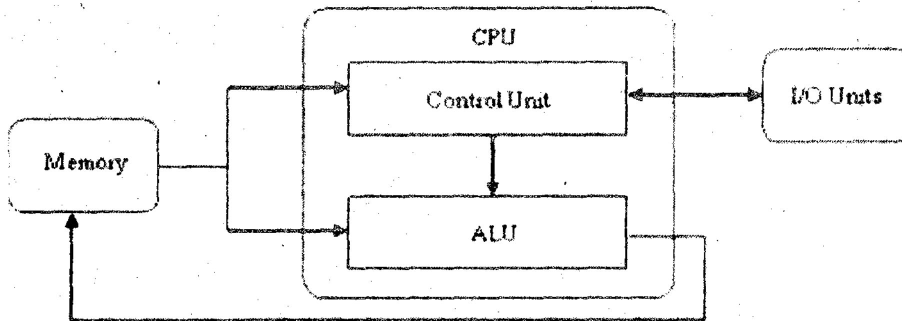

Figure 1: The original von Neumann architecture (Source: Lilja et al., 1998)

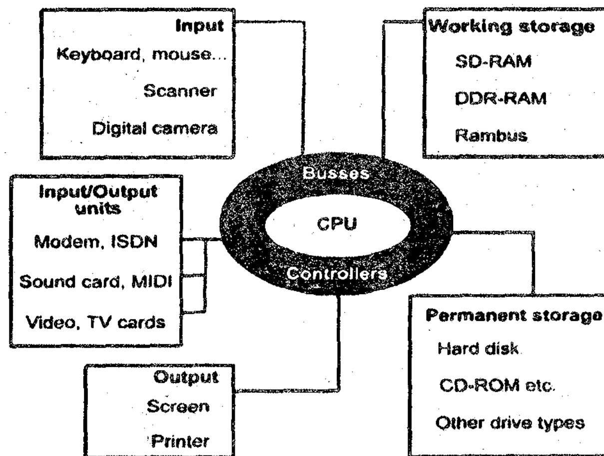

Figure 2: The von Neumann architecture in modern computers (Source: Lilja et al., 1998)

The memory is a collection of storage cells, each of which can be in one of two different states. One state represents a value of "0", and the other state represents a value of "1". By distinguishing these two different logical states, each cell is capable of storing a single binary digit, or bit, of information. These bit storage cells are logically organized into words, each of which is b bits wide. Each word is assigned a unique address in the range [0,...,N – 1], (Lilja et al., 1998).

The CPU identifies the word that it wants either to read or write by storing its unique address in a special memory address register (MAR) – A register temporarily stores a value within the CPU. The memory responds to a read request by reading the value stored at the requested address and passing it to the CPU via the CPU-memory data bus. The value then is temporarily stored in the memory buffer register (MBR) – also sometimes called the memory data register – before it is used by the control unit or ALU. For a write operation, the CPU stores the value it wishes to write into the MBR and the corresponding address in the MAR. The memory then copies the value from the MBR into the address pointed to by the MAR (Lilja et al., 1998).

Finally, the input/output (I/O) devices interface the computer system with the outside world. These devices allow programs and data to be entered into the system and provide a means for the system to control some type of output device. Each I/O port has a unique address to which the CPU can either read or write a value. From the CPU's point of view, an I/O device is accessed in a manner very similar to the way it accesses memory. In fact, in some systems the hardware makes it appear to the CPU that the I/O devices are actually memory locations. This configuration, in which the CPU sees no distinction between memory and I/O devices, is referred to as memory-mapped I/O. In this case, no separate I/O instructions are necessary (Rosch, 1999).

4.1 Key Features

Given its basic organization, computers with a von Neumann architecture generally share several key features that distinguish them from simple preprogrammed (or hardwired) controllers (Myers, 1982).

- Instructions and data are both stored in the same main memory. As a result, instructions are not distinguished from data. Similarly, different types of data, such as a floating-point value, an integer value, or a character code, are all

indistinguishable. The meaning of the data is not stored with it; rather the meaning of a particular bit pattern stored in the memory is determined entirely by how the CPU interprets it. An interesting consequence of this feature is that the same data stored at a given memory location can be interpreted at different times as either an instruction or as data. For example, when a compiler executes, it reads the source code of a program written in a high-level language, such as C++ or Pascal, and converts it to a sequence of instructions that can be executed by the CPU. The output of the compiler (object code) is stored in memory like any other type of data. However, the CPU can now execute the compiler output data (object code) simply by interpreting them as instructions. Thus, the same values stored in memory are treated as data by the compiler, but are subsequently treated as executable instructions by the CPU.

Another consequence of this feature is that each instruction must specify how it interprets the data on which it operates. Thus, for instance, a von Neumann architecture will have one set of arithmetic instructions for operating on integer values and another set for operating on floating-point values, and so on (Lilja et al., 1998).

- The second key feature is that memory is accessed by name (i.e., address) independent of the bit pattern stored at each address. Because of this feature, values stored in memory can be interpreted as addresses as well as data or instructions. Thus, programs can manipulate addresses using the same set of instructions that the CPU uses to manipulate data. This flexibility of how values in memory are interpreted allows very complex, dynamically changing patterns to be generated by the CPU to access any variety of data structure regardless of the type of value being read or written (Lilja et al., 1998).
- The third major characteristic of a computer with the von Neumann architecture is that, the order in which a program executes its instructions is sequential, unless that order is explicitly altered. A special register in the CPU called the program counter (PC) contains the address of the next instruction in memory to be executed. After each instruction is executed, the value in the PC is incremented to point to the next instruction in the sequence to be executed. This sequential execution order can be changed by the program itself using branch instructions (such as an IF statement), which store a new value into the PC register. Alternatively, special hardware can sense some external event, such as an interrupt, and load a new value into the PC to cause the CPU to begin executing a new sequence of instructions. One consequence of this feature is that, while this scheme simplifies the writing of programs and the design and implementation of the CPU, it also limits the potential performance of this architecture (Lilja et al., 1998).
- Fourthly, memory is sequentially addressed. As a result, memory in a von Neumann computer is one-dimensional. These are in conflict with our programming languages. Most of the resulting program, therefore, is generated to provide for the mapping of multidimensional data onto the one-dimensioned memory and to contend with the placement of all of the data into the same memory (Myers, 1982).

4.2 Instruction Types in a von Neumann Architecture

A processor's instruction set is the collection of all the instructions that can be executed. The individual instructions can be classified into three basic types: data movement, data transformation, and program control (Lilja et al., 1998). Data movement instructions simply move data between registers or memory locations, or between I/O devices and the CPU. Data movement instructions are actually somewhat misnamed, since most move operations are nondestructive. That is, the data are not actually moved but, instead, are copied from one location to another. Nevertheless, common usage continues to refer to these operations as data movement instructions. Data transformation instructions take one or more data values as input and perform some operation on them, such as an addition, a logical OR, or some other arithmetic or logical operation, to produce a new value. Finally, program control instructions can alter the flow of instruction execution from its normal sequential order by loading a new value into the PC. This change in the instruction execution order can be done conditionally on the results of previous instructions (Lilja et al., 1998).

In addition to these three basic instruction types, more recent processors have added instructions that can be broadly classified as system control instructions. These types of instructions generally are not necessary for the correct operation of the CPU but, instead, are used to improve its performance. For example, some CPUs have implemented prefetch instructions that can begin reading a location in memory even before it is needed. A variety of other system control instructions also can be supported by the system (VanderWiel and Lilja, 1997).

4.3 Instruction Execution

Execution of instructions is a two-step process. First, the next instruction to be executed, which is the one whose address is in the program counter (PC), is fetched from the memory and stored in the Instruction Register (IR) in the CPU. The CPU then executes the instruction to produce the desired result. This fetch-execute cycle, which is called an instruction cycle, is then repeated for each instruction in the program (Heuring and Jordan, 1997).

In fact, the execution of an instruction is slightly more complex than is indicated by this simple fetch-execute cycle. The interpretation of each instruction actually requires the execution of several smaller substeps called microoperations. The microoperations performed for a typical instruction execution cycle in a

von Neumann architecture are described in the following steps (Heuring and Jordan, 1997):

1. Fetch an instruction from memory at the address pointed to by the Program Counter (PC). Store this instruction in the IR.
2. Increment the value stored in the PC to point to the next instruction in the sequence of instructions to be executed.
3. Decode the instruction in the IR to determine the operation to be performed and the addressing modes of the operands.
4. Calculate any address values needed to determine the locations of the source operands and the address of the destination.
5. Read the values of the source operands.
6. Perform the operation specified by the op-code.
7. Store the results at the destination location.
8. Go to Step 1 to repeat this entire process for the next instruction.

We will like to point out here that, not all of these microoperations need to be performed for all types of instructions. For instance, a conditional branch instruction does not produce a value to be stored at a destination address. Instead, it will load the address of the next instruction to be executed (i.e., the branch target address) into the PC if the branch is to be taken. Otherwise, if the branch is not taken, the PC is not changed and executing this instruction has no effect. Similarly, an instruction that has all of its operands available in registers will not need to calculate the addresses of its source operands.

The time at which each microoperation can execute is coordinated by a periodic signal called the CPU's clock. Each microoperation requires one clock period to execute. The time required to execute the slowest of these microoperations determines the minimum period of this clock, which is referred to as the CPU's cycle time (Rosch, 1999). The reciprocal of this time is the CPU's clock rate. The minimum possible value of the cycle time is determined by the electronic circuit technology used to implement the CPU. Typical clock rates in today's CPUs (Pentium IV, for example) are 1.8 to 2.0 GHz, which corresponds to a cycle time of 3.0×10^7 to 3.3×10^8 ns (Heuring and Jordan, 1997).

An instruction that requires all seven of these microoperations to be executed will take seven clock cycles to complete from the time it is fetched to the time its final result is stored in the destination location. Thus, the combination of the number of microoperations to be executed for each instruction, the mix of instructions executed by a program, and the cycle time determine the overall performance of the CPU (Heuring and Jordan, 1997).

5.0 BOTTLENECKS OF THE VON NEUMANN ARCHITECTURE

5.1 Memory Access Bottleneck

While the basic computer organization proposed by von Neumann is widely used, the separation of the memory and the CPU also has led to one of its fundamental performance limitations, specifically, the delay to access memory. Due to the differences in technologies used to implement CPUs and memory devices and to the improvements in CPU architecture and organization, such as very deep pipelining, the cycle time of CPUs has reduced at a rate much faster than the time required to access memory (Feldman and Retter, 1994). As a result, a significant imbalance between the potential performance of the CPU and the memory has developed. Since the overall performance of the system is limited by its slowest component, this imbalance presents an important performance bottleneck. This limitation often has been referred to as the von Neumann bottleneck (Feldman and Retter, 1994).

However, the provision of Cache memory in the CPU has helped to reduce the effect of this imbalance; because, while an instruction is being executed the next instruction can be fetched and placed in the cache in the CPU, thus, reducing the time lag for the CPU to go far in memory to pick the requisite instruction (Smith, 1982).

5.2 Decode Bottleneck

The performance of computer systems based on the von Neumann architecture is also limited by this architecture's "one instruction at a time" execution paradigm. Executing multiple instructions simultaneously using pipelining can improve performance by exploiting parallelism among instructions. However, performance is still limited by the decode bottleneck (Flynn, 1966) since only one instruction can be decoded for execution in each cycle. To allow more parallelism to be exploited, multiple operations must be simultaneously decoded for execution.

However, while pipelining can improve the performance of the CPU, it also adds substantial complexity to its design and implementation (Hennessy and Patterson, 1995).

6.0 PARALLELISM AS ENHANCEMENT OF VON NEUMANN ARCHITECTURE

The sequence of instructions decoded and executed by the CPU is referred to as an instruction stream. Similarly, a data stream is the corresponding sequence of operands specified by those instructions (Flynn, 1966). Using these definitions, we consider the following taxonomy for parallel computing systems, taking cognizance of the fact that Single Instruction stream Single Data stream (SISD) systems are traditional von Neumann systems:

6.1 Single Instruction stream Multiple Data stream (SIMD)

In such systems, an instruction specifies a single operation that is performed on several different data values simultaneously. For example, the basic operand in an SIMD machine may be an array. In this case, an element-by-element addition of one array to another would require a single addition instruction whose operands are two complete arrays of the same size. If the arrays consist of n rows and m columns, nm total additions would be performed simultaneously. Because of their ability to efficiently operate on large arrays, SIMD processors often are referred to as array processors and are frequently used in image-processing types of applications (Flynn, 1966).

6.2 Multiple Instruction stream Single Data stream (MISD)

In an MISD processor, each individual element in the data stream passes through multiple instruction execution units (Flynn, 1966). These execution units may combine several data streams into a single stream (by adding them together, for instance), or an execution unit may transform a single stream of data (performing a square root operation on each element, for instance). The operations performed and the flow of the data streams are often fixed, however, limiting the range of applications for which this type of system would be useful. MISD processors often are referred to as *systolic arrays* and typically are used to execute a fixed algorithm, such as a digital filter, on a continuous stream of input data (Flynn, 1966).

6.3 Multiple Instruction stream Multiple Data stream (MIMD)

MIMD systems often are considered to be the "true" parallel computer systems (Flynn, 1966). Message-passing parallel computer systems are essentially independent SISD processors that can communicate with each other by sending messages over a specialized communication network. Each processor maintains its own independent address space so any sharing of data must be explicitly specified by the application programmer.

We will like to point out here that, while these parallel architectures have shown excellent potential for improving the performance of computer systems, they are still limited by their requirement that only independent instructions can be executed concurrently. For example, if a programmer or a compiler is unable to verify that two instructions or two tasks are never dependent upon one another, they must conservatively be assumed to be dependent. This assumption then forces the parallel computer system to execute them sequentially.

6.4 Speculative Parallel Architectures

Consequent upon the limitations of conventional parallel systems described above, there are several recently proposed speculative parallel architectures (Sohi et al., 1995) which would, in this case, aggressively assume that the instructions or tasks are not dependent and would begin executing them in parallel. Simultaneous with this execution, the processors would check predetermined conditions to ensure that the independence assumption was correct when the tasks are actually executed. If the speculation was wrong, the processors must rollback their processing to a non-speculative point in the instruction execution stream. The tasks then must be re-executed sequentially. A considerable performance enhancement is possible, however, when the speculation is determined to be correct. Obviously, there must be a careful trade-off between the cost of rolling-back the computation and the probability of being wrong.

7.0 PROPOSED ALTERNATIVE ARCHITECTURES

What we have been discussing so far has been pure von Neumann machines with different levels of enhancements for the resulting bottlenecks. However, there are proposals for alternative architectures that will not directly be based on the von Neumann scheme.

- **Data Flow Approach:** This proposal aims at replacing the notion of defining computation in terms of a sequence of discrete operations (Sharp, 1985). This model, deeply rooted in the von Neumann tradition, sees a program in terms of an orderly execution of instructions as set forth by the program. The programmer defines the order in which operations will take place, and the program counter follows this order as the CPU executes the instructions. This control flow approach would be replaced by a data flow model in which the operations are executed in an order resulting only from the interdependencies of the data.

- Another proposal aims at avoiding the memory access bottleneck by the use of programs that operate on structures or conceptual units rather than on words (which normally travel between memory and CPU, one word at a time). Functions are defined without naming any data, and then these functions are combined to produce a program. Such a functional approach began with LISP, but had to be forced into a conventional hardware-software environment. New functional programming architectures may be developed from the ground up (Eisenbach, 1987).

- **Harvard Architecture:** It was developed by a research group at Harvard University at roughly the same time as von Neumann's group developed the von Neumann architecture (Lilja et al., 1998). The primary advantage of the Harvard architecture is that it provides two separate paths between the processor and the memory. This separation allows both an instruction and a data value to be transferred simultaneously from the memory to the processor. The ability to access both instructions and data simultaneously is especially important to achieving high performance in pipelined CPUs because one instruction can be fetching its operands from memory at the same time a new instruction is being fetched from memory; more so when the Harvard architecture provides separate memories for data and instructions, respectively (Lilja et al., 1998).

While we agree with the Data Flow, structures or conceptual units and the Harvard architecture respectively, as alternatives to the traditional von Neumann scheme, we offer what we call **Self-identifying Data Architecture** as an alternative.

With this proposed architecture, each operand in memory will carry with it some bits to identify its type. The computer only needs to perform one operation, such as ADD. This is all we see in a high-level language. Since the data identifies its type, it is the responsibility of the hardware to determine whether to

perform an Integer Add, Floating-point Add, Double-precision, Complex, or whatever it might be. This is unlike the von Neumann concept, in which the instructions themselves must determine whether a set of bits is operated upon as an integer, floating-pint, character, or other data type.

The result of our proposed alternative architecture will be, more complex and expensive hardware, but greatly simplified and shorter programs, since the burden of type-identification will now lie with the hardware and not program instructions.

8.0 CONCLUSION

The fundamental ideas embodied in the traditional von Neumann architecture have proven to be amazingly robust. Enhancements and extensions to these ideas have led to tremendous improvements in the performance of computer systems over the past 60 years. Today, however, many computer researchers feel that future improvements in computer system performance will require the extensive use of new, innovative techniques, such as parallel (Hwang, 1993) and speculative execution, as well as our proposed Self-identifying Data Architecture. Besides, complementing software technology needs to be developed that can lower the development costs of an ever-increasing range of potential applications.

Nevertheless, we will like to point out that, no matter the innovations in the near future, it is likely that the underlying organizations of future computer systems will continue to be based on the architecture and concepts proposed by von Neumann and his contemporaries.

REFERENCES

Aspray, W., 1990. John von Neumann and the Origins of Modern Computing. The MIT Press, Cambridge, Mass.

Eisenbach, S., 1987. Functional Programming: Languages, Tools, and Architectures. Ellis Horwood Limited, Chichester, England.

Feldman, James M. and Retter, Charles T., 1994. Computer Architecture: A Designer's Text Based on a Generic RISC. McGraw-Hill, Inc., New York.

Flynn, Michael J., 1966. Very high-speed computing systems. Proceedings of the IEEE, 54(12):1901-1909.

Godfrey, M. D. and Hendry, D. F., 1993. The computer as von Neumann planned it. IEEE Annals of the History of Computing, 15(1):11-21.

Hennessy, J. L. and Patterson, D. A., 1995. Computer Architecture: A Quantitative Approach, Second Edition. Morgan Kaufmann, San Mateo, CA.

Heuring, V. P. and Jordan, H. F., 1997. Computer Systems Design and Architecture. Addison Wesley Longman, Menlo Park, CA.

Hwang, Kai., 1993. Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw-Hill, Inc., New York, NY.

Lilja, David J. and Eigenmann, R., 1998. Von Neumann Computers. University of Minnesota and Purdue University.

Myers, G. J., 1982. Advances in Computer Architecture. John Wiley & Sons, New York.

Randell, Brian., 1994. The origins of computer programming. IEEE Annals of the History of Computing, 16(4):6-15.

Riley, H. Norton., 1987. The von Neumann Architecture of Computer Systems. California State Polytechnic University, Pomona, California.

Rosch, Winn L., 1999. Hardware Bible. Que, Indiana, USA.

Sharp, J. A., 1985. Data Flow Computing. Ellis Horwood Limited, Chichester, England.

Smith, Alan J., 1982. Cache Memories. ACM Computing Surveys, 14(3): 473 – 530.

Sohi, G. S., Breach, S. E. and Vijaykumar, T. N., 1995. Multiscalar processors. International Symposium on Computer Architecture, Pages 414 – 425.

Stern, Nancy., 1981. From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers. Digital Press, Bedford, Mass.

VanderWiel, Steven and Lilja, David J., 1997. When caches are not enough: Data prefetching techniques. IEEE Computer, 30(7):23-30.