COMPUTER ARCHITECTURE

Prabhat Mishra
University of Florida, Gainesville, Florida, U.S.A.

Keywords: computer organization, datapath and control, cache coherence, instruction-set, pipelining, hazards, exceptions, instruction scheduling, speculation, memory hierarchy, caches, virtual memory, hard disk, multicore, multiprocessor, processor architecture.

Contents

1. Introduction
2. Instruction-Set Design
3. Processor Design
4. Improving Processor Performance
5. Memory Hierarchy
6. Multiprocessor Architectures
7. Input/Output and Storage Systems
8. Conclusions
Acknowledgments
Glossary
Bibliography
Biographical Sketch

UNESCO-EOLSS
SAMPLE CHAPTERS

1. Introduction

Our daily life is immersed in the vast ocean of computing. In some cases, computing is obvious – when we use computers (laptop, desktop, workstation or servers) to perform personal or official activities. However, in many scenarios computing is hidden inside systems, popularly known as embedded systems. The examples of such systems include cell phones, cameras and electronic appliances. Embedded systems are also present in automobiles, airplanes, satellites, military and biomedical equipments. Embedded systems are designed for specific applications and have stringent requirements in terms of cost, energy requirements (battery life), real-time constraints, security, reliability, and so on. Although, computers and embedded systems have certain differences, they have one commonality – both uses efficient design to deliver fast computation. In fact, a $300 personal computer in 2012 can produce more performance than a 1 million dollar supercomputer in 1985. This tremendous growth in performance improvement is possible because of significant advances in *computer architecture*.

This chapter introduces history and basic principles of computer architecture. It describes two important aspects of computer architecture: computer organization and instruction-set architecture. Next, it describes various ways of improving processor performance including pipelining, dynamic scheduling, branch prediction and speculation. It also describes how caches are used in memory hierarchy to improve memory performance. It describes the role and importance of both input/output devices

and reliable storage systems. Finally, it describes the latest technology trend in multiprocessor/multicore architectures, and concludes the chapter.

1.1. Brief History

There has been debate on when and how computers or automated computing started, whether it is mechanical multiplier in the 16th century, analog calculators in the 17th century, navigation instruments in the 18th century, or electronic computers in the 19th century. In the early 1940s, John Atanasoff built a small-scale special-purpose computer, called ABC (Atanasoff Berry Computer). Eckert and Mauchly built the world's first fully operational electronic general-purpose computer, called ENIAC (Electronic Numerical Integrator and Calculator). It was used during World War II for computing artillery firing tables, but it was not publicly disclosed until 1946 [Hennessy and Patterson, 2007]. It is widely acknowledged that the development of Turing machine [Alan Turing] played a significant role in the creation of the modern computers.

While working in the ENIAC project, John von Neumann wrote a memo proposing a stored-program computer called EDVAC (Electronic Discrete Variable Automatic Computer), which has served as the basis for the commonly used term von Neumann computer. There have been many significant developments that led to the development of the first IBM computer, the 701. In those days, many people used to believe that the market for computers was very limited ("I think there is a world market for maybe five computers." Thomas Watson, 1943). IBM quickly became the most successful computer company. In 1971, the first processor on a chip (Intel 4004) was developed, which led to more powerful 8/16/32-bit high performance microprocessors and the rise of the RISC processor in the 1980s.

UNESCO-EOLSS SAMPLE CHAPTERS

1.2. Trends in Cost, Power and Performance

Cost is not important for high-end servers where companies are willing to pay a heavy price for servers that can provide required level of throughput (e.g., number of emails scanned per second), scalability and dependability. However, processor/memory cost is a major concern for personal computers as well as for many commodity embedded systems. A low-end embedded processor costs $1-$10, a low-end processor for personal computer costs $50-$100, whereas a processor for server costs $200-$10000. Similarly, cost for the memory is determined by the size and performance. When large number of items are manufactured, the price goes down over a period of time due to improved yield (manufacturing cost goes down).

Although, performance was the most important aspect of designing computer architecture until 2000, power and energy (energy = power x time) are the prime considerations these days. Energy requirements directly translate to battery life. Therefore to improve battery life, it is important to design an architecture that delivers the required level of performance while minimizes energy requirements. Even if a personal computer or server is connected to wall power, it is still important to reduce energy requirements since higher power dissipation causes the system temperature to rise, which in turn significantly affects the reliability and lifetime of the system.

Performance is a key design consideration. Prior to the mid-1980s, processor performance growth was about 25% per year and largely driven by technological advances. In 1965, Gordon Moore predicted that the number of transistors that can be integrated on a die would double every 18 to 24 months i.e., grow exponentially with time. Exponential growth in transistors leads to dramatic performance improvement. From 1986 to 2002, the increase in performance growth is about 52% primarily due to advanced architectural designs [Hennessy and Patterson, 2007]. Since 2002, focus has shifted to reducing power dissipation while maintaining 20% performance growth. Use of multiple cores enabled designers to improve performance while maintaining power/energy constraints.

1.3. Amdahl's Law

Amdahl's law is an important mechanism to measure performance gain by improving certain portions of a computer. The law states that the impact of improving the performance of a specific component on overall computer depends on the fraction of the time the faster component can be used. In other words, if a component is rarely used, even if we improve it significantly, it will not significantly improve the overall performance. Amdahl's law can be described using the following equation, where I is the enhancement (improvement) of a component that is used f (fraction) percentage of time, and *Speedup* implies the overall speedup (ratio between overall execution time before the improvement over the overall execution time after the improvement).

$$Speedup = \frac{1}{(1-f) + \frac{f}{I}}$$

Consider a processor that can perform scientific computation as well as image processing. The designers have improved the image processing part by 10 times (i.e., $I = 10$). In the given application, image processing is performed 20% of the time ($f = 0.2$). Note that the performance of scientific computation is not affected due to this enhancement. We can compute the overall speedup = $1/\left((1-0.2)+0.2/10\right) = 1/.82 = 1.22$. Although, the image processing part is improved 10 times, the overall speedup is only 22%.

1.4. The Big Picture

Figure 1 shows a broad overview of computer architecture where a processor is connected to memory using a bus. The processor also has a cache for performance improvement. In general, many input/output (I/O) devices are connected to bus using I/O controllers. In this figure, external secondary storage (hard disk) and computer display is connected to processor via bus.

It is important to understand how this computer architecture relates to program execution? A compiler translates a high-level description of a program (e.g., written in C, Java or C++) into machine representation (binary code). This binary code is stored in hard disk or loaded using a USB drive.

UNESCO-EOLSS SAMPLE CHAPTERS

During execution processor fetches the suitable binary code (instructions and data), performs the required computation and stores the results back in memory. At the end of computation, these results will be transferred to I/O devices such as stored back in hard disk and/or displayed in the computer display.

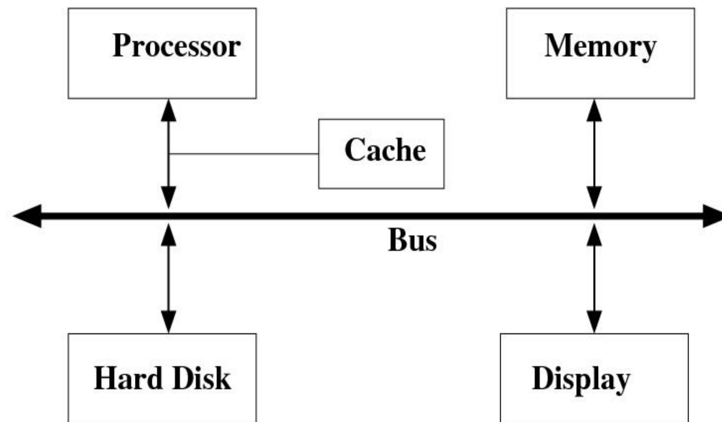

Figure 1. An overview of computer architecture

2. Instruction-Set Design

We communicate with others using some natural language such as Bengali, Chinese, English, French, German, Spanish, and so on. However, to communicate with processor (computer's hardware) we need to use computer's language, called *instructions*. The vocabulary of commands (instructions) is known as *instruction-set*.

2.1. Instruction Types

Each processor supports different types of instructions. For example, the MIPS processor [Patterson and Hennessy, 2008] supports four major types of instructions: arithmetic, logical, data transfer, and decision making. Arithmetic instructions correspond to arithmetic computations such as addition, subtraction, multiplication and division. Similarly, logical instructions are designed to support logical computations such as shift, and, or, nor etc.

The data transfer instructions enable transfer of data between processor and memory using operations such as load and store. Finally, the decision making instructions allow conditional branches and unconditional jumps. Although different processor supports different types of instructions, all of them cover the similar set of computations to ensure that any high-level program can be mapped by a compiler using the instructions supported by that processor. For example, the ARM [ARM] processor supports different instruction types such as data processing and flow control instructions.

2.2. Operations and Operands

Consider a simple MIPS-like arithmetic instruction "*add a, b, c*". This instructs the processor to add two variables *b* and *c* and put their sum in *a*. This instruction has four parts. The first part ("add") is known as *operation* and the next three parts are termed as

UNESCO-EOLSS
SAMPLE CHAPTERS

operands. In this case, the first operand ("a") is a *destination* operand whereas the last two operands ("b" and "c") are *source* operands. Source operands provide the input values whereas the destination operand stores the result of computation performed by the respective operation.

A program, written in high-level language, may use many variables to describe the computation. However, in computer hardware the number of operand locations is limited – these hardware locations are called *registers*. For example, MIPS arithmetic instructions can choose operands from 32 32-bit registers including \$s0–\$s7, \$t0–\$t9, \$zero, \$a0–\$a3, \$v0–\$v1, \$gp, \$fp, \$sp, \$ra and \$at. Figure 2a shows the general instruction format (called *assembly*) for arithmetic instructions. Figure 2c shows a simple addition statement in C language (a popular high level programming language) and its corresponding MIPS assembly in Figure 2d.

The assembly assumes that the variable A, B and C are mapped to registers \$s1, \$s2 and \$t0, respectively. Interestingly, the computer hardware does not even understand assembly instruction. It only understands binary ('0' and '1') representation of this assembly. As a result, each processor describes the binary format for each type of instruction. For example, the binary format for arithmetic instruction is shown in Figure 2b where the first and last field is 6 bits and the remaining fields are 5 bits. Figure 2e shows the binary representation of the assembly shown in Figure 2d.

<operation> <dest> <src1> <src2>

oper: operation to be performed
dest: destination operand
src1: first source operand
src2: second source operand

a) Syntax of Arithmetic Instructions

Special	Src1	Src2	Dest	Shamt	Function

Special and Function fields together determine the operation
Shamt: Shift Amount

b) Binary Format for Arithmetic Instructions

C = A + B

c) Example: C Statement

add \$t0, \$s1, \$s2

d) Example: MIPS Assembly

0	\$s1	\$s2	\$t0	0	add

= 0 17 18 8 0 32
= 000000 10001 10010 01000 00000 100000

e) Example: MIPS binary

Figure 2. An example statement and corresponding MIPS assembly and binary representation

The assembly as well as binary formats are different for different types of instructions. For example, to load (transfer) data from memory location a sample assembly would be: *lw \$s1, 20(\$s2)*. The operand "lw" implies that it is loading a word from memory. The destination operand is register \$s1, and the source memory location (address) is "\$s2+20". Note that data transfer instructions uses register locations as well as memory locations, whereas arithmetic instructions uses only register locations. The logical instructions are similar to arithmetic instructions. For example, *and \$s1, \$s2, \$s3*

performs a bit-by-bit logical *and* operation of $s2 and $s3 and places the result in $s1. There are two types of decision making instructions. The conditional branch instructions (such as "*beq $s1, $s2, 25"*) checks whether a condition is true (whether $s1 is equal to $s2 in this case) and if yes it jumps to the branch target. The unconditional branches (such as "*j 2500"*) directly jump to the branch target.

2.3. Representation of Numbers

The computer hardware understands only binary values. Therefore, when any value is provided in the assembly, it needs to be translated to binary representation by assembler (last stage of compiler). High-level languages differentiate between positive and negative numbers by using a negation operator ('–'). However, the computer hardware follows different representations for signed and unsigned numbers. Given a 32-bit location, it can represent unsigned values between 0 and $2^{32} - 1$. However, the signed numbers can be represented by sign-magnitude, 1's complement or 2's complement. Typically, signed binary numbers use 2's complement representation. In 2's complement representation, positive numbers are same as it's unsigned binary counterpart whereas the negative numbers requires 2's complement computation. For example, in a 4-bit representation, the unsigned binary for 3 is 0010. In signed 4-bit representation, +3 is also 0011, where –4 is 1101.

The representation of floating-point numbers follows IEEE 754 floating-point standard [IEEE 754]. It allows 32-bit single precision (excess-127) or 64-bit double precision (excess-1023) representations. In both forms the binary is divided in three parts. The first bit is to indicate the sign ('0' for positive and '1' for negative). The next 8 bits (11 bits for double precision) represent exponent whereas the last 23 bits (52 bits for double precision) are used to represent the mantissa. For example, $+2^6$ x 1.01000111001 has a positive sign (i.e., '0'). In excess-127 representation the exponent would be 133 i.e., "10000101". Extra 0s can be added at the right to create 23-bit mantissa of "01000111001000000000000". Therefore, 32-bit representation of the number $+2^6$ x 1.01000111001 is "0 10000101 01000111001000000000000".

UNESCO-EOLSS
SAMPLE CHAPTERS

2.4. Computer Arithmetic

Once a processor receives an arithmetic instruction, it needs to read the source registers and perform the required computation. For example, if it receives "addi $s1, $s2, -6" it needs to perform addition of the content of $s2 with -6. Note that "addi" is an add operation same as "add" except that it directly uses the value of one of the source operands (known as *immediate* value). Let us assume that register $s2 has a value of 7. Therefore, we need to computer 7 + (-6). Figure 3 shows the addition where the numbers are first converted into its 2's complement 8-bit representation, and then addition is performed in a bit-by-bit fashion from right to left. Adding '0' with '1' produces sum as '1' and carry as '0', adding '1' with '1' produces sum as '1' and carry as '1'. The carry is passed to the next digit to the left, exactly the way we add decimal numbers. The result is +1. In general, the computation needs to keep track of overflow when adding two positive or two negative numbers. If the carry output at the last bit (leftmost bit) is '1', it implies an overflow in case of 2'complement addition. Certain instructions ignore overflow, whereas other instructions uses it as an exception and

performs appropriate corrective action. Subtraction is a special case of addition since $A - B = A + (-B)$. In other words, addition is performed by treating the second one as a negative number i.e., computing the 2's complement of the second number. Floating-point addition is performed in a similar manner except that first decimal points are aligned (exponents are adjusted) followed by addition and normalization of result and rounding, if necessary.

```
Carries (1) (1) (1) (1) (1) (1) (0) (0)
    +7 = 0   0   0   0   0   1   1   1
    -6 = 1   1   1   1   1   0   1   0
    ---------------------------------------
         0   0   0   0   0   0   0   1  = +1
```

Figure 3. Addition using 2's complement arithmetic

Other arithmetic operations are also performed in a similar fashion using binary values. For example, Figure 4a shows multiplication of two 4-bit binary numbers (8 x 3 = 24). The 8-bit result is produced by padding '0's on the leftmost side. Similarly, Figure 4b shows how 8 is divided by 3 to produce 2 as quotient and 2 as remainder.

```
+8 = 1 0 0 0                      1 1 | 1 0 0 0 | 1 0
+3 = 0 0 1 1                          |   1 1   |
     -------                            -------
     1 0 0 0                              1 0
   1 0 0 0
 0 0 0 0
0 0 0 0
-----------
0 0 1 1 0 0 0 = 24

a) Compute 8 x 3                  b) Compute 8 / 3
```

Figure 4. Multiplication and division of binary numbers

3. Processor Design

As mentioned earlier, the memory holds the program (sequence of instructions) and these instructions are executed by the processor. Figure 5 shows a simplified overview of the five major activities in the processor when executing an arithmetic instruction. The program counter (PC) contains the address of the instruction that needs to be fetched from the memory. Next, the instruction (32-bit binary) is decoded to understand its different parts, operations and operands. The source operands of the instruction are read from registers. Next, the intended computation (execution) is performed. Finally,

SAMPLE CHAPTERS – UNESCO – EOLSS

the result is stored back in the destination register. When load (store) instruction is executed, the data is read from (written back to) memory.

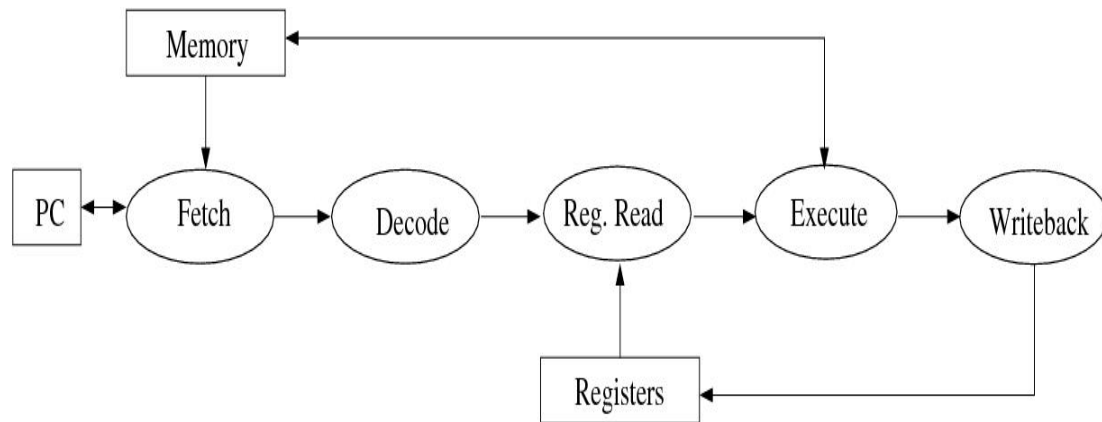

Figure 5. Five major activities during instruction execution – i) fetch an instruction from memory pointed by PC, ii) decode to figure out the operation and operands, iii) read thesource operands, iv) perform the required computation, and v) write back the result.

UNESCO-EOLSS SAMPLE CHAPTERS

3.1. Design of Datapath and Control

To build the processor hardware, the designers need to build two types of components (units): *datapath* units and *control* units. The datapath units are designed to operate on data or hold data. Instruction/data memories, registers and Arithmetic Logic Unit (ALU) are examples of datapath units. The control units, also known as controllers, are responsible for generating required control signals. Based on the current instruction the controller needs to activate the intended computation in the ALU. Assume that an ALU supports 16 types of arithmetic and logical operations. Therefore, the ALU controller needs to generate 4-bit ($2^4 = 16$) control to activate the required operation. For example, ‘0000’ is generated for ‘addition’, ‘0001’ for subtraction etc. We also need a control signal to identify whether a register is being read or written (e.g., ‘0’ means read and ‘1’ means write). If there are 32 registers, the register address will be 5-bits ($2^5 = 32$) to indicate which specific register needs to be read or written. For example, ‘00000’ implies the first register ($s0), ‘00001’ implies next register ($s1), and so on. Table 1 shows a sample set of data and control signals for an example instruction during instruction fetch, decode, operand read, ALU operation, and writing result. Assume that $s2 and $s3 have values 3 and 4, respectively, represented in four bit representation. The “Function” field represents the values of the last six bits of the 32-bit binary that represents the opcode for the MIPS arithmetic instruction. Note that x’s indicates that the values are either not known or not required at that stage. For example, during register write only three sets of signals are important: register read/write signal should be 1 (to indicate write), register address bus should be loaded with register address for $s1 (i.e., 00001), and the result should be ready. In this example, operation and operand addresses become available after instruction decode, the source operand values becomes ready after operand ready and result is available after the execution. Similarly, control signals are required to activate read and write from memory.

	Function (opcode)	Data Values						Control Signals	
		Source 1		Source 2		Destination		ALU operation	Register Read/Write
		Address	Value	Address	Value	Address	Value		
Inst. Fetch	xxxxxx	xxxxx	xxxx	xxxxx	xxxx	xxxxx	xxxx	xxxx	x
Inst. Decode	100000	00010	xxxx	00011	xxxx	00001	xxxx	xxxx	x
Operand Read	100000	00010	0011	00011	0100	00001	xxxx	xxxx	0
Execution	100000	00010	0011	00011	0100	00001	0111	0000	x
Write Result	100000	000010	0011	00011	0100	00001	0111	xxxx	1

'x' indicates unknown/undefined/don't care values

Table 1. Sample data and control signals for an example instruction "add $s1, $s2, $s3"

-

-

-

TO ACCESS ALL THE **32 PAGES** OF THIS CHAPTER,
Visit: http://www.eolss.net/Eolss-sampleAllChapter.aspx

Bibliography

Hennessy J.L., Patterson D.A. (2012). *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, California. [This is a popular textbook on computer architecture. It covers a wide variety of topics related to design of high-performance and low-power architecture.]

Turing A. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, series 2, volume 42, pages 230-265, 1937. Errata appeared in Series 2, volume 43, pages 544-546, 1937. [This is a famous paper on defining computability and development of Turing machines.]

ARM. *ARM Instruction Set*. http://infocenter.arm.com [ARM processors are widely used in embedded and mobile systems]

Patterson D.A., Hennessy J.L. (2009). *Computer Organization and Design – The Hardware / Software Interface*. Morgan Kaufmann Publishers, San Francisco, California. [This is a popular undergraduate-level textbook on computer organization. It covers fundamentals of processor and memory design including design of instruction-set (assembly language), datapath and controller, pipeline, memory hierarchy, and so on.]

IEEE 754 (2008). *IEEE 754: Standard for Binary Floating-Point Arithmetic*. http://grouper.ieee.org/groups/754/ [This standard specifies binary formats, basic operations, conversions, and exceptional conditions for floating-point numbers.]

SPEC Benchmarks. http://www.spec.org/. [These are widely used benchmarks for performance evaluation of computer architecture. These benchmarks are developed and maintained by Standard Performance Evaluation Corporation (SPEC)]

Flynn M. (1972). *Some Computer Organization and Their Effectiveness*, IEEE Transactions on Computers, volume C-21, issue 9, pages 948-960, September 1972. [The paper classifies different types of computer architectures into four major categories.]

UNESCO-EOLSS SAMPLE CHAPTERS

Stallings W. (2009). Computer Organization and Architecture, Pearson Prentice Hall, New Jersey. [This is a popular textbook that covers various aspects of both computer organization and computer architecture.]

Tanenbaum A. S. (2006). Structured Computer Organization, Pearson Prentice Hall, New Jersey. [This is a popular textbook on computer organization.]

Hamacher C., Vranesic Z., Zaky S. (2002). Computer Organization, McGraw-Hill. [This a computer organization textbook]

Pasricha S., Dutt N. (2008), *On-Chip Communication Architectures – System on Chip Interconnect*. Morgan Kaufmann Publishers, San Francisco, California. [This book describes concepts, standards and design principles of a wide variety of on-chip communication architectures.]

Mishra P., Dutt N. (2008), *Processor Description Languages – Applications and Methodologies*. Morgan Kaufmann Publishers, San Francisco, California. [This book describes a wide variety of architecture description languages and associated specification, exploration and rapid prototyping methodologies.]

Biographical Sketch

Prabhat Mishra is an Associate Professor in the Department of Computer and Information Science and Engineering (CISE) at the University of Florida where he leads the CISE Embedded Systems Group. His research interests include design automation of embedded systems, energy-aware computing, hardware/software verification, and design of trustworthy systems. He received his Ph.D. in computer science from the University of California, Irvine. Prior to joining University of Florida, he spent several years in various semiconductor and design automation companies including Intel, Motorola, Synopsys and Texas Instruments. He has published four books, ten book chapters and more than 100 research articles in premier journals and conferences. His research has been recognized by several awards including the National Science Foundation CAREER Award, two best paper awards (VLSI Design 2011 and CODES+ISSS 2003), and 2004 EDAA Outstanding Dissertation Award from the European Design Automation Association. Prof. Mishra currently serves as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, IEEE Design & Test of Computers, and Journal of Electronic Testing, Guest Editor of IEEE Transactions on Computers, and as a program/organizing committee member of several ACM and IEEE conferences. He is a senior member of Association for Computing Machinery (ACM), and a senior member of Institute of Electrical and Electronics Engineers (IEEE).

UNESCO – EOLSS
SAMPLE CHAPTERS