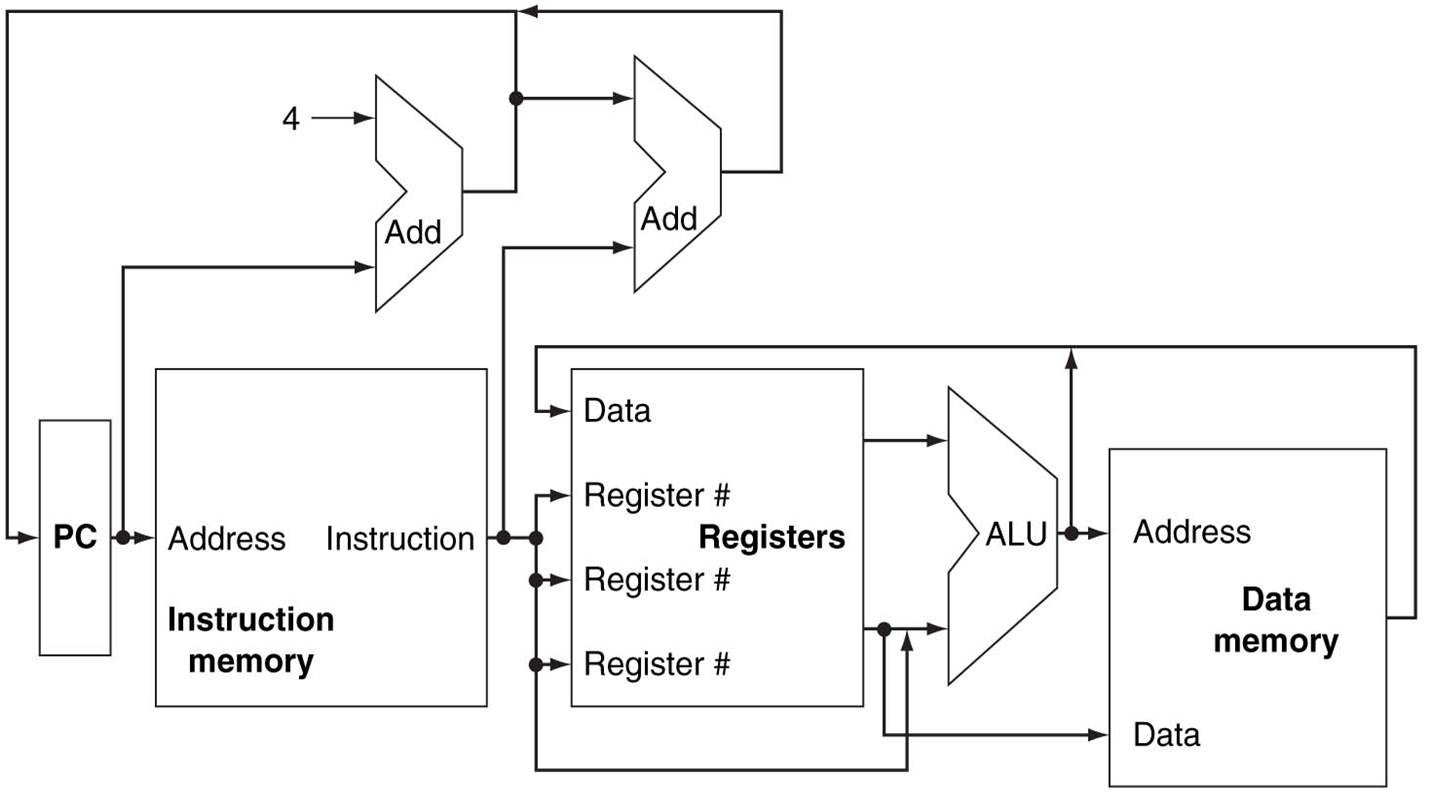
Name : Ahmed Abdel Tawab Abdel Aal

# B.N : 1129

# Date : 11/6/2021

# Topic : computer architecture

# Application Brief : I chose the topic of computer architecture because of its importance in our time, and I will talk about What is computer architecture.Instruction set architecture (ISA).Microarchitecture. Systems design.



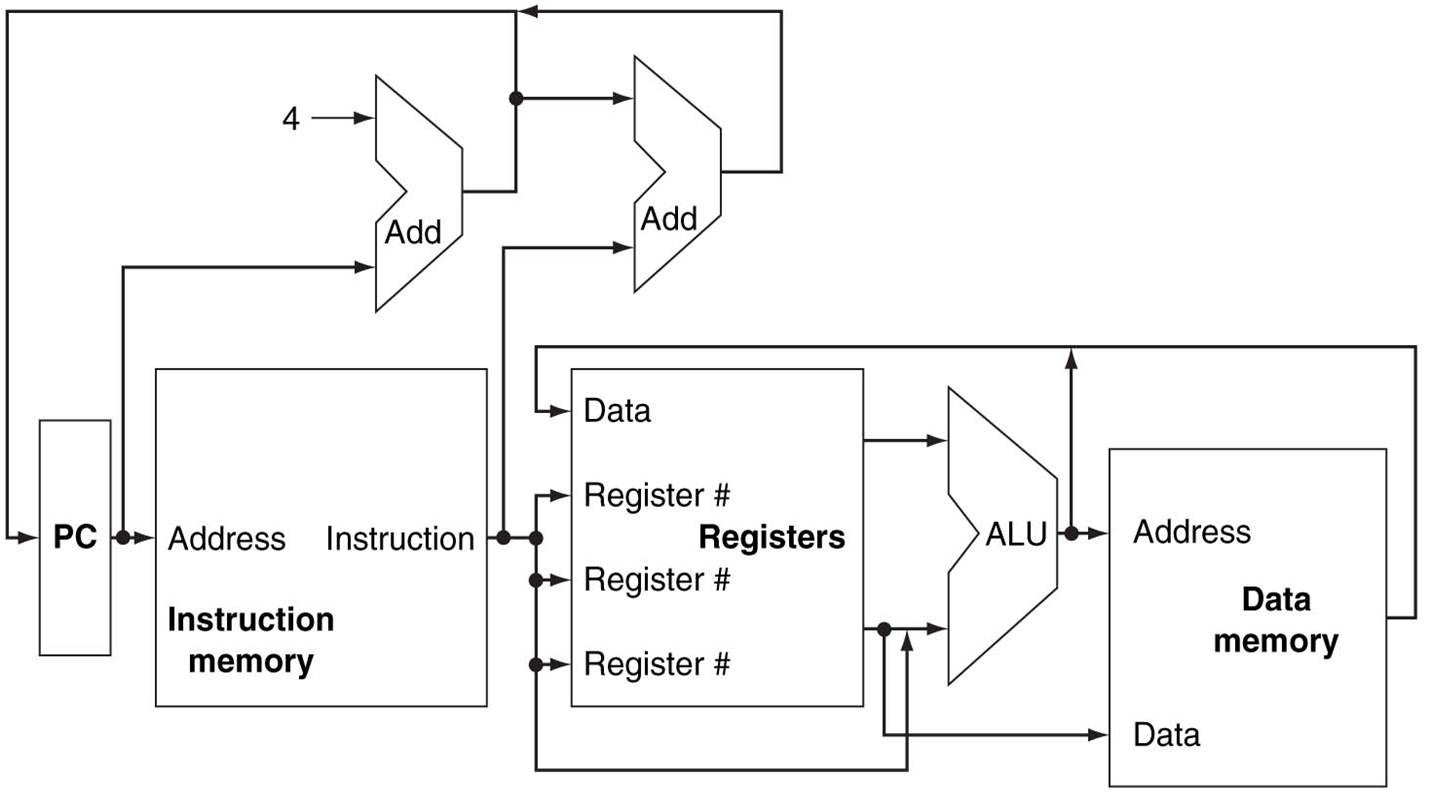
**What is computer architecture ?**

In computer engineering, **computer architecture** is a set of rules and methods that describe the functionality, organization, and implementation of computer systems.  
Some definitions of architecture define it as describing the capabilities and programming model of a computer but not a particular implementation.[1] In other definitions computer architecture involves instruction set architecture design, microarchitecture design, logic design, and implementation.[2]  
The first documented computer architecture was in the correspondence between Charles Babbage and Ada Lovelace, describing the analytical engine. When building the computer Z1 in 1936, Konrad Zuse described in two patent applications for his future projects that machine instructions could be stored in the same storage used for data, i.e., the stored-program concept.[3][4] Two other early and important examples are:  
John von Neumann's 1945 paper, First Draft of a Report on the EDVAC, which described an organization of logical elements;[5] and Alan Turing's more detailed Proposed Electronic Calculator for the Automatic Computing Engine, also 1945 and which cited John von Neumann's paper.[6] The term “architecture” in computer literature can be traced to the work of Lyle R. Johnson and Frederick P. Brooks, Jr., members of the Machine Organization department in IBM's main research center in 1959. Johnson had the opportunity to write a proprietary research communication about the Stretch, an IBM-developed supercomputer for Los Alamos National Laboratory (at the time known as Los Alamos Scientific Laboratory). To describe the level of detail for discussing the luxuriously embellished computer, he noted that his description of formats, instruction types, hardware parameters, and speed enhancements were at the level of “system architecture”, a term that seemed more useful than “machine organization”.[7]  
Subsequently, Brooks, a Stretch designer, opened Chapter 2 of a book called Planning a Computer System: Project Stretch by stating, “Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints.”[8]  
Brooks went on to help develop the IBM System/360 (now called the IBM zSeries) line of computers, in which “architecture” became a noun defining “what the user needs to know”.[9] Later, computer users came to use the term in many less explicit ways.[10]  
  
The earliest computer architectures were designed on paper and then directly built into the final hardware form.[11] Later, computer architecture prototypes were physically built in the form of a transistor–transistor logic (TTL) computer—such as the prototypes of the 6800 and the PA-RISC—tested, and tweaked, before committing to the final hardware form. As of the 1990s, new computer architectures are typically "built", tested, and tweaked—inside some other computer architecture in a computer architecture simulator; or inside a FPGA as a soft microprocessor; or both—before committing to the final hardware form.[12]

**The discipline of computer architecture has three main subcategories:[13]**

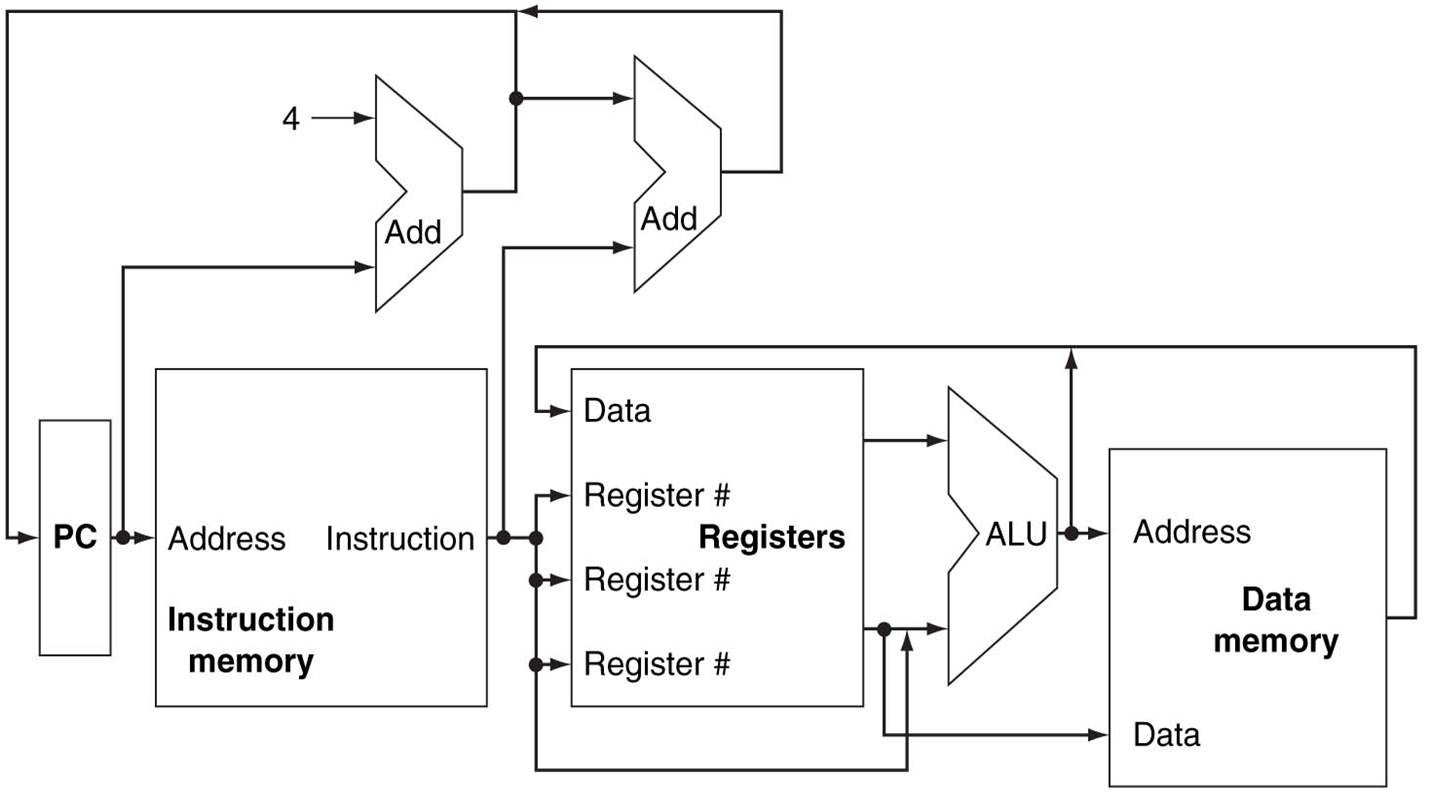
* Instruction set architecture (ISA): defines the machine code that a processor reads and acts upon as well as the word size, memory address modes, processor registers, and data type.
* Microarchitecture: also known as "computer organization", this describes how a particular processor will implement the ISA.[14] The size of a computer's CPU cache for instance, is an issue that generally has nothing to do with the ISA.
* Systems design: includes all of the other hardware components within a computing system, such as data processing other than the CPU (e.g., direct memory access), virtualization, and multiprocessing

Computer architecture is concerned with balancing the performance, efficiency, cost, and reliability of a computer system. The case of instruction set architecture can be used to illustrate the balance of these competing factors. More complex instruction sets enable programmers to write more space efficient programs, since a single instruction can encode some higher-level abstraction (such as the x86 Loop instruction).[16] However, longer and more complex instructions take longer for the processor to decode and can be more costly to implement effectively. The increased complexity from a large instruction set also creates more room for unreliability when instructions interact in unexpected ways.  
  
The implementation involves integrated circuit design, packaging, power, and cooling. Optimization of the design requires familiarity with compilers, operating systems to logic design, and packaging.[17]



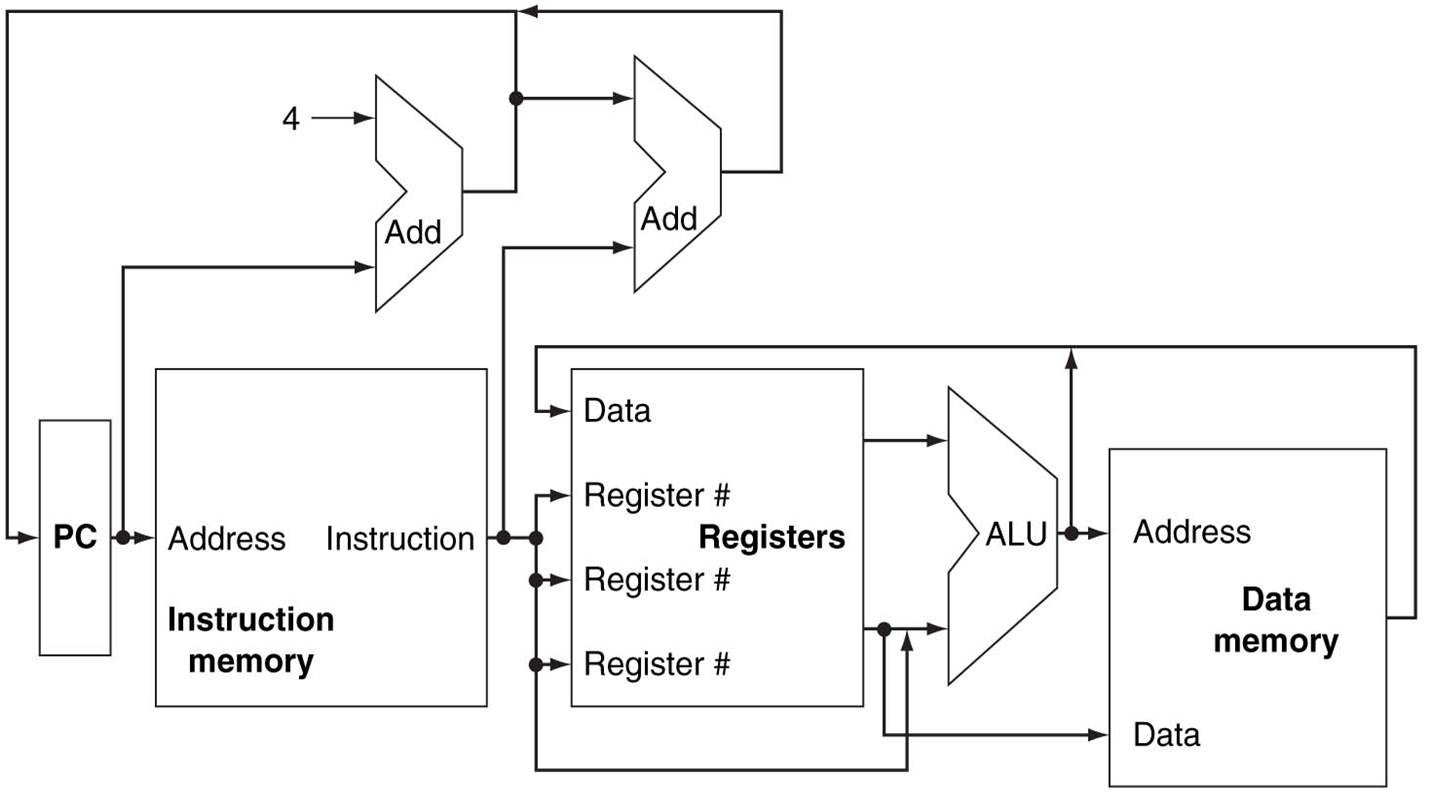
**Instruction set architecture (ISA)**

In computer science, **an instruction set architecture (ISA)** is an abstract model of a computer. It is also referred to as architecture or computer architecture. A realization of an ISA, such as a central processing unit (CPU), is called an implementation.  
In general, an ISA defines the supported data types, the registers, the hardware support for managing main memory, fundamental features (such as the memory consistency, addressing modes, virtual memory), and the input/output model of a family of implementations of the ISA.  
  
An ISA specifies the behavior of machine code running on implementations of that ISA in a fashion that does not depend on the characteristics of that implementation, providing binary compatibility between implementations. This enables multiple implementations of an ISA that differ in performance, physical size, and monetary cost (among other things), but that are capable of running the same machine code, so that a lower-performance, lower-cost machine can be replaced with a higher-cost, higher-performance machine without having to replace software. It also enables the evolution of the microarchitectures of the implementations of that ISA, so that a newer, higher-performance implementation of an ISA can run software that runs on previous generations of implementations.  
  
  
If an operating system maintains a standard and compatible application binary interface (ABI) for a particular ISA, machine code for that ISA and operating system will run on future implementations of that ISA and newer versions of that operating system. However, if an ISA supports running multiple operating systems, it does not guarantee that machine code for one operating system will run on another operating system, unless the first operating system supports running machine code built for the other operating system.  
  
An ISA can be extended by adding instructions or other capabilities, or adding support for larger addresses and data values; an implementation of the extended ISA will still be able to execute machine code for versions of the ISA without those extensions. Machine code using those extensions will only run on implementations that support those extensions.  
The binary compatibility that they provide make ISAs one of the most fundamental abstractions in computing.  
An instruction set architecture is distinguished from a microarchitecture, which is the set of processor design techniques used, in a particular processor, to implement the instruction set. Processors with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the Advanced Micro Devices Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs.  
  
The concept of an architecture, distinct from the design of a specific machine, was developed by Fred Brooks at IBM during the design phase of System/360. Prior to NPL [System/360], the company's computer designers had been free to honor cost objectives not only by selecting technologies but also by fashioning functional and architectural refinements. The SPREAD compatibility objective, in contrast, postulated a single architecture for a series of five processors spanning a wide range of cost and performance. None of the five engineering design teams could count on being able to bring about adjustments in architectural specifications as a way of easing difficulties in achieving cost and performance objectives.[1]:p.137  
  
Some virtual machines that support bytecode as their ISA such as Smalltalk, the Java virtual machine, and Microsoft's Common Language Runtime, implement this by translating the bytecode for commonly used code paths into native machine code. In addition, these virtual machines execute less frequently used code paths by interpretation (see: Just-in-time compilation). Transmeta implemented the x86 instruction set atop VLIW processors in this fashion.



**Microarchitecture**

In computer engineering, **microarchitecture**, also called **computer organization** and sometimes abbreviated as µarch or uarch, is the way a given instruction set architecture (ISA) is implemented in a particular processor.[1] A given ISA may be implemented with different microarchitectures;[2][3] implementations may vary due to different goals of a given design or due to shifts in technology.[4]  
  
Computer architecture is the combination of microarchitecture and instruction set architecture.  
The ISA is roughly the same as the programming model of a processor as seen by an assembly language programmer or compiler writer. The ISA includes the execution model, processor registers, address and data formats among other things. The microarchitecture includes the constituent parts of the processor and how these interconnect and interoperate to implement the ISA.  
  
The microarchitecture of a machine is usually represented as (more or less detailed) diagrams that describe the interconnections of the various microarchitectural elements of the machine, which may be anything from single gates and registers, to complete arithmetic logic units (ALUs) and even larger elements. These diagrams generally separate the datapath (where data is placed) and the control path (which can be said to steer the data).[5]  
  
The person designing a system usually draws the specific microarchitecture as a kind of data flow diagram. Like a block diagram, the microarchitecture diagram shows microarchitectural elements such as the arithmetic and logic unit and the register file as a single schematic symbol. Typically, the diagram connects those elements with arrows, thick lines and thin lines to distinguish between three-state buses (which require a three-state buffer for each device that drives the bus), unidirectional buses (always driven by a single source, such as the way the address bus on simpler computers is always driven by the memory address register), and individual control lines. Very simple computers have a single data bus organization – they have a single three-state bus. The diagram of more complex computers usually shows multiple three-state buses, which help the machine do more operations simultaneously.  
  
Each microarchitectural element is in turn represented by a schematic describing the interconnections of logic gates used to implement it. Each logic gate is in turn represented by a circuit diagram describing the connections of the transistors used to implement it in some particular logic family. Machines with different microarchitectures may have the same instruction set architecture, and thus be capable of executing the same programs. New microarchitectures and/or circuitry solutions, along with advances in semiconductor manufacturing, are what allows newer generations of processors to achieve higher performance while using the same ISA.  
In principle, a single microarchitecture could execute several different ISAs with only minor changes to the microcode.



**Systems design**

**Systems design** is the process of defining the architecture, product design, modules, interfaces, and data for a system to satisfy specified requirements. Systems design could be seen as the application of systems theory to product development. There is some overlap with the disciplines of systems analysis, systems architecture and systems engineering.[1][2]  
  
If the broader topic of product development "blends the perspective of marketing, design, and manufacturing into a single approach to product development,"[3] then design is the act of taking the marketing information and creating the design of the product to be manufactured. Systems design is therefore the process of defining and developing systems to satisfy specified requirements of the user.  
  
The basic study of system design is the understanding of component parts and their subsequent interaction with one another.[4]  
Until the 1990s, systems design had a crucial and respected role in the data processing industry. In the 1990s, standardization of hardware and software resulted in the ability to build modular systems. The increasing importance of software running on generic platforms has enhanced the discipline of software engineering.

**Architectural design**

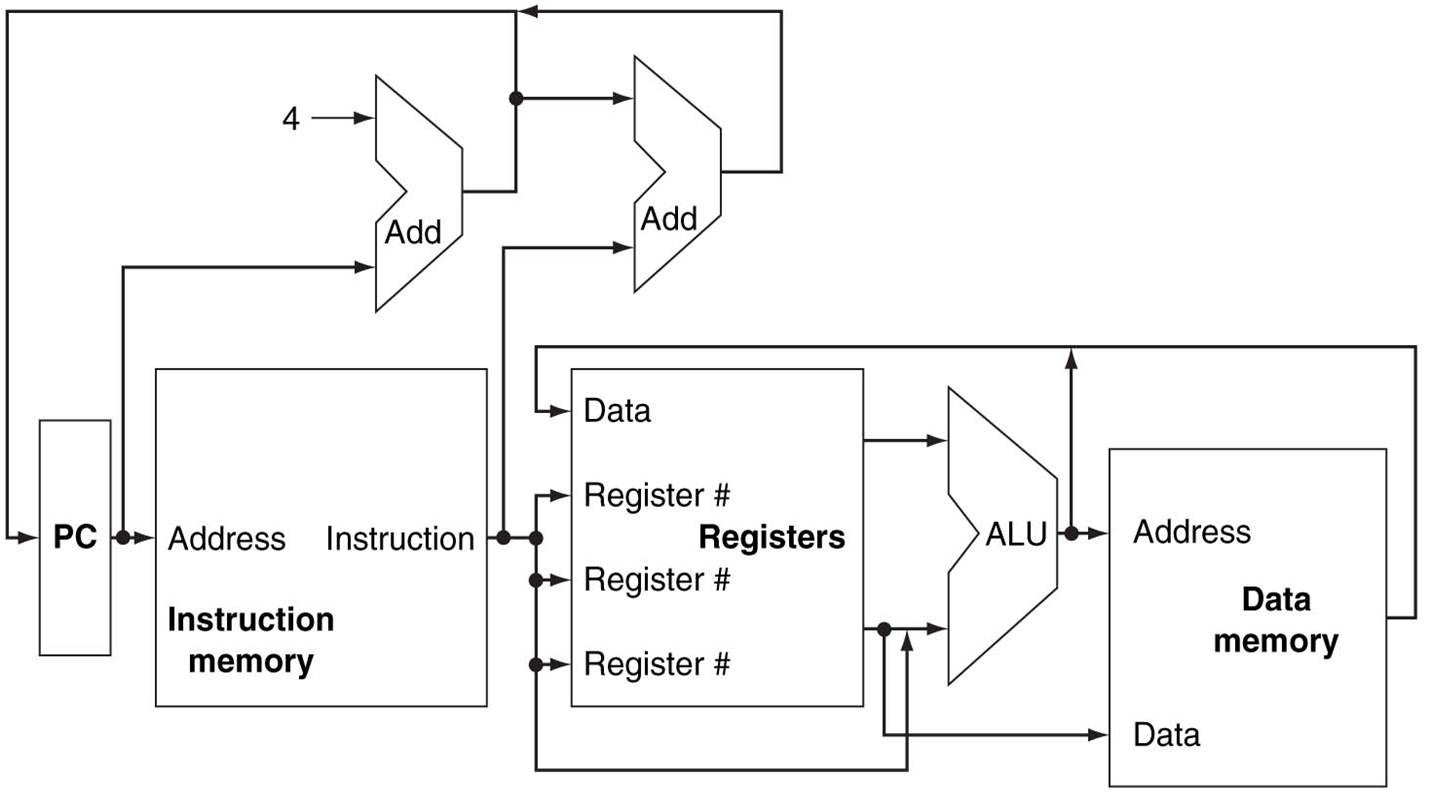
The architectural design of a system emphasizes the design of the system architecture that describes the structure, behavior and more views of that system and analysis.

**Logical design**

The logical design of a system pertains to an abstract representation of the data flows, inputs and outputs of the system. This is often conducted via modelling, using an over-abstract (and sometimes graphical) model of the actual system. In the context of systems, designs are included. Logical design includes entity-relationship diagrams (ER diagrams).

**Physical design**

The physical design relates to the actual input and output processes of the system. This is explained in terms of how data is input into a system, how it is verified/authenticated, how it is processed, and how it is displayed. In physical design, the following requirements about the system are decided. *Input requirement,  
Output requirements,  
Storage requirements,  
Processing requirements,  
System control and backup or recovery.*Put another way, the physical portion of system design can generally be broken down into three sub-tasks: User Interface Design Data Design Process Design User Interface Design is concerned with how users add information to the system and with how the system presents information back to them. Data Design is concerned with how the data is represented and stored within the system. Finally, Process Design is concerned with how data moves through the system, and with how and where it is validated, secured and/or transformed as it flows into, through and out of the system. At the end of the system design phase, documentation describing the three sub-tasks is produced and made available for use in the next phase. Physical design, in this context, does not refer to the tangible physical design of an information system. To use an analogy, a personal computer's physical design involves input via a keyboard, processing within the CPU, and output via a monitor, printer, etc. It would not concern the actual layout of the tangible hardware, which for a PC would be a monitor, CPU, motherboard, hard drive, modems, video/graphics cards, USB slots, etc. It involves a detailed design of a user and a product database structure processor and a control processor. The H/S personal specification is developed for the proposed system.



An ISA may be classified in a number of different ways. A common classification is by architectural complexity. A complex instruction set computer (CISC) has many specialized instructions, some of which may only be rarely used in practical programs. A reduced instruction set computer (RISC) simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines, having their resulting additional processor execution time offset by infrequent use.[2]  
  
Other types include very long instruction word (VLIW) architectures, and the closely related long instruction word (LIW) and explicitly parallel instruction computing (EPIC) architectures. These architectures seek to exploit instruction-level parallelism with less hardware than RISC and CISC by making the compiler responsible for instruction issue and scheduling.  
Architectures with even less complexity have been studied, such as the minimal instruction set computer (MISC) and one instruction set computer (OISC). These are theoretically important types, but have not been commercialized.  
  
Machine language is built up from discrete statements or instructions. On the processing architecture, a given instruction may specify:  
opcode (the instruction to be performed) e.g. add, copy, test any explicit operands: registers literal/constant values addressing modes used to access memory More complex operations are built up by combining these simple instructions, which are executed sequentially, or as otherwise directed by control flow instructions.  
Instruction types Examples of operations common to many instruction sets include: **Data handling and memory operations** Set a register to a fixed constant value. Copy data from a memory location or a register to a memory location or a register (a machine instruction is often called move; however, the term is misleading). Used to store the contents of a register, the result of a computation, or to retrieve stored data to perform a computation on it later. Often called load and store operations. Read and write data from hardware devices. Arithmetic and logic operations Add, subtract, multiply, or divide the values of two registers, placing the result in a register, possibly setting one or more condition codes in a status register. increment, decrement in some ISAs, saving operand fetch in trivial cases. Perform bitwise operations, e.g., taking the conjunction and disjunction of corresponding bits in a pair of registers, taking the negation of each bit in a register. Compare two values in registers (for example, to see if one is less, or if they are equal). Floating-point instructions for arithmetic on floating-point numbers. Control flow operations Branch to another location in the program and execute instructions there. Conditionally branch to another location if a certain condition holds. Indirectly branch to another location. Call another block of code, while saving the location of the next instruction as a point to return to. Coprocessor instructions Load/store data to and from a coprocessor or exchanging with CPU registers. Perform coprocessor operations. Complex instructions Processors may include "complex" instructions in their instruction set. A single "complex" instruction does something that may take many instructions on other computers.[citation needed] Such instructions are typified by instructions that take multiple steps, control multiple functional units, or otherwise appear on a larger scale than the bulk of simple instructions implemented by the given processor. Some examples of "complex" instructions include:  
transferring multiple registers to or from memory (especially the stack) at once moving large blocks of memory (e.g. string copy or DMA transfer) complicated integer and floating-point arithmetic (e.g. square root, or transcendental functions such as logarithm, sine, cosine, etc.) SIMD instructions, a single instruction performing an operation on many homogeneous values in parallel, possibly in dedicated SIMD registers performing an atomic test-and-set instruction or other read-modify-write atomic instruction instructions that perform ALU operations with an operand from memory rather than a register Complex instructions are more common in CISC instruction sets than in RISC instruction sets, but RISC instruction sets may include them as well. RISC instruction sets generally do not include ALU operations with memory operands, or instructions to move large blocks of memory, but most RISC instruction sets include SIMD or vector instructions that perform the same arithmetic operation on multiple pieces of data at the same time. SIMD instructions have the ability of manipulating large vectors and matrices in minimal time. SIMD instructions allow easy parallelization of algorithms commonly involved in sound, image, and video processing. Various SIMD implementations have been brought to market under trade names such as MMX, 3DNow!, and AltiVec.  
  
Instruction encoding One instruction may have several fields, which identify the logical operation, and may also include source and destination addresses and constant values. This is the MIPS "Add Immediate" instruction, which allows selection of source and destination registers and inclusion of a small constant. On traditional architectures, an instruction includes an opcode that specifies the operation to perform, such as add contents of memory to register—and zero or more operand specifiers, which may specify registers, memory locations, or literal data. The operand specifiers may have addressing modes determining their meaning or may be in fixed fields. In very long instruction word (VLIW) architectures, which include many microcode architectures, multiple simultaneous opcodes and operands are specified in a single instruction.  
Some exotic instruction sets do not have an opcode field, such as transport triggered architectures (TTA), only operand(s).  
The Forth virtual machine and other "0-operand" instruction sets lack any operand specifier fields, such as some stack machines including NOSC.[3][better source needed]  
Conditional instructions often have a predicate field—a few bits that encode the specific condition to cause an operation to be performed rather than not performed. For example, a conditional branch instruction will transfer control if the condition is true, so that execution proceeds to a different part of the program, and not transfer control if the condition is false, so that execution continues sequentially. Some instruction sets also have conditional moves, so that the move will be executed, and the data stored in the target location, if the condition is true, and not executed, and the target location not modified, if the condition is false. Similarly, IBM z/Architecture has a conditional store instruction. A few instruction sets include a predicate field in every instruction; this is called branch predication.  
Number of operands Instruction sets may be categorized by the maximum number of operands explicitly specified in instructions.  
(In the examples that follow, a, b, and c are (direct or calculated) addresses referring to memory cells, while reg1 and so on refer to machine registers.)  
==> C = A+B ⋛ ==  
0-operand (zero-address machines), so called stack machines: All arithmetic operations take place using the top one or two positions on the stack: push a, push b, add, pop c. C = A+B needs four instructions. For stack machines, the terms "0-operand" and "zero-address" apply to arithmetic instructions, but not to all instructions, as 1-operand push and pop instructions are used to access memory. 1-operand (one-address machines), so called accumulator machines, include early computers and many small microcontrollers: most instructions specify a single right operand (that is, constant, a register, or a memory location), with the implicit accumulator as the left operand (and the destination if there is one): load a, add b, store c. C = A+B needs three instructions. 2-operand — many CISC and RISC machines fall under this category: CISC — move A to C; then add B to C. C = A+B needs two instructions. This effectively 'stores' the result without an explicit store instruction. CISC — Often machines are limited to one memory operand per instruction: load a,reg1; add b,reg1; store reg1,c; This requires a load/store pair for any memory movement regardless of whether the add result is an augmentation stored to a different place, as in C = A+B, or the same memory location: A = A+B. C = A+B needs three instructions. RISC — Requiring explicit memory loads, the instructions would be: load a,reg1; load b,reg2; add reg1,reg2; store reg2,c. C = A+B needs four instructions. 3-operand, allowing better reuse of data:[4] CISC — It becomes either a single instruction: add a,b,c C = A+B needs one instruction. CISC — Or, on machines limited to two memory operands per instruction, move a,reg1; add reg1,b,c; C = A+B needs two instructions. RISC — arithmetic instructions use registers only, so explicit 2-operand load/store instructions are needed: load a,reg1; load b,reg2; add reg1+reg2->reg3; store reg3,c; C = A+B needs four instructions. Unlike 2-operand or 1-operand, this leaves all three values a, b, and c in registers available for further reuse.[4] more operands—some CISC machines permit a variety of addressing modes that allow more than 3 operands (registers or memory accesses), such as the VAX "POLY" polynomial evaluation instruction. Due to the large number of bits needed to encode the three registers of a 3-operand instruction, RISC architectures that have 16-bit instructions are invariably 2-operand designs, such as the Atmel AVR, TI MSP430, and some versions of ARM Thumb. RISC architectures that have 32-bit instructions are usually 3-operand designs, such as the ARM, AVR32, MIPS, Power ISA, and SPARC architectures. Each instruction specifies some number of operands (registers, memory locations, or immediate values) explicitly. Some instructions give one or both operands implicitly, such as by being stored on top of the stack or in an implicit register. If some of the operands are given implicitly, fewer operands need be specified in the instruction. When a "destination operand" explicitly specifies the destination, an additional operand must be supplied. Consequently, the number of operands encoded in an instruction may differ from the mathematically necessary number of arguments for a logical or arithmetic operation (the arity). Operands are either encoded in the "opcode" representation of the instruction, or else are given as values or addresses following the opcode. Register pressure Register pressure measures the availability of free registers at any point in time during the program execution. Register pressure is high when a large number of the available registers are in use; thus, the higher the register pressure, the more often the register contents must be spilled into memory. Increasing the number of registers in an architecture decreases register pressure but increases the cost.[5] While embedded instruction sets such as Thumb suffer from extremely high register pressure because they have small register sets, general-purpose RISC ISAs like MIPS and Alpha enjoy low register pressure. CISC ISAs like x86-64 offer low register pressure despite having smaller register sets. This is due to the many addressing modes and optimizations (such as sub-register addressing, memory operands in ALU instructions, absolute addressing, PC-relative addressing, and register-to-register spills) that CISC ISAs offer.[6] Instruction length The size or length of an instruction varies widely, from as little as four bits in some microcontrollers to many hundreds of bits in some VLIW systems. Processors used in personal computers, mainframes, and supercomputers have instruction sizes between 8 and 64 bits. The longest possible instruction on x86 is 15 bytes (120 bits).[7] Within an instruction set, different instructions may have different lengths. In some architectures, notably most reduced instruction set computers (RISC), instructions are a fixed length, typically corresponding with that architecture's word size. In other architectures, instructions have variable length, typically integral multiples of a byte or a halfword. Some, such as the ARM with Thumb-extension have mixed variable encoding, that is two fixed, usually 32-bit and 16-bit encodings, where instructions cannot be mixed freely but must be switched between on a branch (or exception boundary in ARMv8). A RISC instruction set normally has a fixed instruction length (often 4 bytes = 32 bits), whereas a typical CISC instruction set may have instructions of widely varying length (1 to 15 bytes for x86). Fixed-length instructions are less complicated to handle than variable-length instructions for several reasons (not having to check whether an instruction straddles a cache line or virtual memory page boundary,[4] for instance), and are therefore somewhat easier to optimize for speed. Code density In early computers, memory was expensive, so minimizing the size of a program to make sure it would fit in the limited memory was often central. Thus the combined size of all the instructions needed to perform a particular task, the code density, was an important characteristic of any instruction set. Computers with high code density often have complex instructions for procedure entry, parameterized returns, loops, etc. (therefore retroactively named Complex Instruction Set Computers, CISC). However, more typical, or frequent, "CISC" instructions merely combine a basic ALU operation, such as "add", with the access of one or more operands in memory (using addressing modes such as direct, indirect, indexed, etc.). Certain architectures may allow two or three operands (including the result) directly in memory or may be able to perform functions such as automatic pointer increment, etc. Software-implemented instruction sets may have even more complex and powerful instructions. Reduced instruction-set computers, RISC, were first widely implemented during a period of rapidly growing memory subsystems. They sacrifice code density to simplify implementation circuitry, and try to increase performance via higher clock frequencies and more registers. A single RISC instruction typically performs only a single operation, such as an "add" of registers or a "load" from a memory location into a register. A RISC instruction set normally has a fixed instruction length, whereas a typical CISC instruction set has instructions of widely varying length. However, as RISC computers normally require more and often longer instructions to implement a given task, they inherently make less optimal use of bus bandwidth and cache memories. Certain embedded RISC ISAs like Thumb and AVR32 typically exhibit very high density owing to a technique called code compression. This technique packs two 16-bit instructions into one 32-bit word, which is then unpacked at the decode stage and executed as two instructions.[8] Minimal instruction set computers (MISC) are a form of stack machine, where there are few separate instructions (16-64), so that multiple instructions can be fit into a single machine word. These types of cores often take little silicon to implement, so they can be easily realized in an FPGA or in a multi-core form. The code density of MISC is similar to the code density of RISC; the increased instruction density is offset by requiring more of the primitive instructions to do a task.[citation needed] There has been research into executable compression as a mechanism for improving code density. The mathematics of Kolmogorov complexity describes the challenges and limits of this. Representation The instructions constituting a program are rarely specified using their internal, numeric form (machine code); they may be specified by programmers using an assembly language or, more commonly, may be generated from programming languages by compilers.

Source Code :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Home</title>

</head>

<body>

    <table style="width: 100%;">

        <tr>

            <td><a href="index.html"> Home </a></td>

            <td><a href="page1.html"> page-1 </a></td>

            <td><a href="page2.html"> page-2 </a></td>

            <td><a href="page3.html"> page-3 </a></td>

            <td><a href="page4.html"> page-4 </a></td>

        </tr>

    </table>

    <br>

    <center><img src="main\_image.JPG" alt="main\_image" width="68%"></center>

    <hr>

    <h1>What is computer architecture ?</h1>

    <big>

    </big>

</body>

</html>