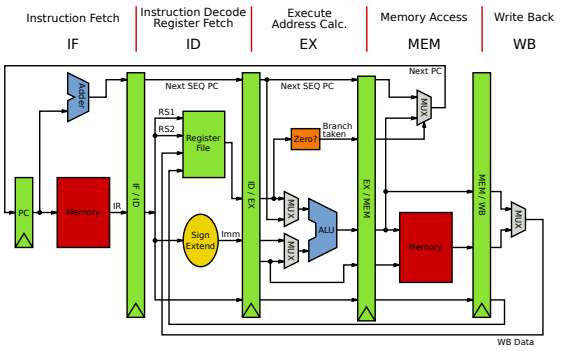
Architecture 1

CISC, RISC, VLIW, Dataflow

Contents

Chapter 1

Computer architecture



Pipelined implementation of MIPS architecture. Pipelining is a key concept in 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 descriptions computer architecture involves instruction set architecture design, microarchitecture design, logic design, and implementation.[2]

1.1 History

The first documented computer architecture was in the correspondence between Charles Babbage and Ada Lovelace, describing the analytical engine. Two other early and important examples were:

- John von Neumann's 1945 paper, First Draft of a Report on the EDVAC, which described an organization of logical elements; and
- Alan Turing's more detailed *Proposed Electronic Calculator* for the Automatic Computing Engine, also 1945 and which cited von Neumann's paper.[3]

The term "architecture" in computer literature can be traced to the work of Lyle R. Johnson, Mohammad Usman Khan and Frederick P. Brooks, Jr., members in 1959 of the Machine Organization department in IBM's main research center. 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."

Subsequently, Brooks, a Stretch designer, started Chapter 2 of a book (Planning a Computer System: Project Stretch, ed. W. Buchholz, 1962) by writing,

> 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.

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". Later, computer users came to use the term in many less-explicit ways.

The earliest computer architectures were designed on paper and then directly built into the final hardware form.[4] 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.

1.2 Subcategories

The discipline of computer architecture has three main subcategories:[5]

1. *Instruction Set Architecture*, or ISA. The 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.

2. *Microarchitecture*, or *computer organization* describes how a particular processor will implement the ISA.[6] The size of a computer's CPU cache for instance, is an issue that generally has nothing to do with the ISA.

3. *System Design* includes all of the other hardware components within a computing system. These include:

   (a) Data processing other than the CPU, such as direct memory access (DMA)

   (b) Other issues such as virtualization, multiprocessing, and software features.

There are other types of computer architecture. The following types are used in bigger companies like Intel, and count for 1% of all of computer architecture

- Macroarchitecture: architectural layers more abstract than microarchitecture

- Assembly Instruction Set Architecture (ISA): A smart assembler may convert an abstract assembly language common to a group of machines into slightly different machine language for different implementations

- Programmer Visible Macroarchitecture: higher level language tools such as compilers may define a consistent interface or contract to programmers using them, abstracting differences between underlying ISA, UISA, and microarchitectures. E.g. the C, C++, or Java standards define different Programmer Visible Macroarchitecture.

- UISA (Microcode Instruction Set Architecture)—a group of machines with different hardware level microarchitectures may share a common microcode architecture, and hence a UISA.

- Pin Architecture: The hardware functions that a microprocessor should provide to a hardware platform, e.g., the x86 pins A20M, FERR/IGNNE or FLUSH. Also, messages that the processor should emit so that external caches can be invalidated (emptied). Pin architecture functions are more flexible than ISA functions because external hardware can adapt to new encodings, or change from a pin to a message. The term "architecture" fits, because the functions must be provided for compatible systems, even if the detailed method changes.

1.3 Roles

1.3.1 Definition

The purpose is to design a computer that maximizes performance while keeping power consumption in check, costs low relative to the amount of expected performance, and is also very reliable. For this, many aspects are to be considered which includes Instruction Set Design, Functional Organization, Logic Design, and Implementation. 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.

1.3.2 Instruction set architecture

Main article: Instruction set architecture

An instruction set architecture (ISA) is the interface between the computer's software and hardware and also can be viewed as the programmer's view of the machine. Computers do not understand high level languages such as Java, C++, or most programming languages used. A processor only understands instructions encoded in some numerical fashion, usually as binary numbers. Software tools, such as compilers, translate those high level languages into instructions that the processor can understand.

Besides instructions, the ISA defines items in the computer that are available to a program—e.g. data types, registers, addressing modes, and memory. Instructions locate these available items with register indexes (or names) and memory addressing modes.

The ISA of a computer is usually described in a small instruction manual, which describes how the instructions are encoded. Also, it may define short (vaguely) mnemonic names for the instructions. The names can be recognized by a software development tool called an assembler. An assembler is a computer program that translates a human-readable form of the ISA into a computer-readable form. Disassemblers are also widely available, usually in debuggers and software programs to isolate and correct malfunctions in binary computer programs.

ISAs vary in quality and completeness. A good ISA compromises between programmer convenience (how easy the code is to understand), size of the code (how much code is required to do a specific action), cost of the computer to interpret the instructions (more complexity means more space needed to disassemble the instructions), and speed of the computer (with larger disassemblers comes longer disassemble time). For example, single-instruction ISAs like an ISA that subtracts one from a value and if the value is zero then the value returns

to a higher value are both inexpensive, and fast, however ISAs like that are not convenient or helpful when looking at the size of the ISA. Memory organization defines how instructions interact with the memory, and how memory interacts with itself.

During design emulation software (emulators) can run programs written in a proposed instruction set. Modern emulators can measure size, cost, and speed to determine if a particular ISA is meeting its goals.

1.3.3 Computer organization

Main article: Microarchitecture

Computer organization helps optimize performance-based products. For example, software engineers need to know the processing power of processors. They may need to optimize software in order to gain the most performance for the lowest price. This can require quite detailed analysis of the computer's organization. For example, in a SD card, the designers might need to arrange the card so that the most data can be processed in the fastest possible way.

Computer organization also helps plan the selection of a processor for a particular project. Multimedia projects may need very rapid data access, while virtual machines may need fast interrupts. Sometimes certain tasks need additional components as well. For example, a computer capable of running a virtual machine needs virtual memory hardware so that the memory of different virtual computers can be kept separated. Computer organization and features also affect power consumption and processor cost.

1.3.4 Implementation

Once an instruction set and micro-architecture are designed, a practical machine must be developed. This design process is called the *implementation*. Implementation is usually not considered architectural design, but rather hardware design engineering. Implementation can be further broken down into several steps:

- **Logic Implementation** designs the circuits required at a logic gate level
- **Circuit Implementation** does transistor-level designs of basic elements (gates, multiplexers, latches etc.) as well as of some larger blocks (ALUs, caches etc.) that may be implemented at the log gate level, or even at the physical level if the design calls for it.
- **Physical Implementation** draws physical circuits. The different circuit components are placed in a chip floorplan or on a board and the wires connecting them are created.
- **Design Validation** tests the computer as a whole to see if it works in all situations and all timings. Once the design validation process starts, the design at the logic level are tested using logic emulators. However, this is usually too slow to run realistic test. So, after making corrections based on the first test, prototypes are constructed using Field-Programmable Gate-Arrays (FPGAs). Most hobby projects stop at this stage. The final step is to test prototype integrated circuits. Integrated circuits may require several redesigns to fix problems.

For CPUs, the entire implementation process is organized differently and is often referred too as CPU design.

1.4 Design goals

The exact form of a computer system depends on the constraints and goals. Computer architectures usually trade off standards, power versus performance, cost, memory capacity, latency (latency is the amount of time that it takes for information from one node to travel to the source) and throughput. Sometimes other considerations, such as features, size, weight, reliability, and expandability are also factors.

The most common scheme does an in depth power analysis and figures out how to keep power consumption low, while maintaining adequate performance.

1.4.1 Performance

Modern computer performance is often described in IPC (instructions per cycle). This measures the efficiency of the architecture at any refresh rate. Since a faster rate can make a faster computer, this is a useful measurement. Older computers had IPC counts as low as 0.1 instructions per cycle. Simple modern processors easily reach near 1. Superscalar processors may reach three to five IPC by executing several instructions per refresh. Multicore and vector processing CPUs can multiply this further by acting on a lot of data per instruction, which have several CPU cores executing in parallel.

Counting machine language instructions would be misleading because they can do varying amounts of work in different ISAs. The "instruction" in the standard measurements is not a count of the ISA's actual machine language instructions, but a unit of measurement, usually based on the speed of the VAX computer architecture.

Many people used to measure a computer's speed by the clock rate (usually in MHz or GHz). This refers to the cycles per second of the main clock of the CPU. However, this metric is somewhat misleading, as a machine with a higher clock rate may not necessarily have greater performance. As a result, manufacturers have moved away from clock speed as a measure of performance.

Other factors influence speed, such as the mix of functional units, bus speeds, available memory, and the type and order of instructions in the programs.

In a typical home computer, the simplest, most reliable way to speed performance is usually to add random access memory (RAM). More RAM increases the likelihood that needed data or a program is stored in the RAM—so the system is less likely to need to move memory data from the disk. The reason why RAM is important is because in a HDD (Hard disk drive) you have physical moving parts that you would need to move to acsess certain parts of a memory. SSD (Solid state drive) are faster than HDD but they still are much slower than the read/write speed of RAM.

There are two main types of speed: latency and throughput. Latency is the time between the start of a process and its completion. Throughput is the amount of work done per unit time. Interrupt latency is the guaranteed maximum response time of the system to an electronic event (like when the disk drive finishes moving some data).

Performance is affected by a very wide range of design choices — for example, pipelining a processor usually makes latency worse, but makes throughput better. Computers that control machinery usually need low interrupt latencies. These computers operate in a real-time environment and fail if an operation is not completed in a specified amount of time. For example, computer-controlled anti-lock brakes must begin braking within a predictable, short time after the brake pedal is sensed or else failure of the brake will occur.

Benchmarking takes all these factors into account by measuring the time a computer takes to run through a series of test programs. Although benchmarking shows strengths, it shouldn't be how you choose a computer. Often the measured machines split on different measures. For example, one system might handle scientific applications quickly, while another might render video games more smoothly. Furthermore, designers may target and add special features to their products, through hardware or software, that permit a specific benchmark to execute quickly but don't offer similar advantages to general tasks.

1.4.2 Power consumption

Main article: Low-power electronics

Power consumption is another measurement that is important in modern computers. Power efficiency can often be traded for speed or lower cost. The typical measurement when referring to power consumption in Computer Architecture is MIPS/W (millions of instructions per second per watt).

Modern circuits have less power required per transistor as the number of transistors per chip grows. This is because each transistor that is in a new chip requires its own power supply. Therefore, power efficiency has increased in importance over time. Recent processor designs such as Intel's Haswell (microarchitecture), put more emphasis on increasing power efficiency. Also, in the world of embedded computing, power efficiency has long been and remains an important goal next to throughput and latency.

1.4.3 Shifts in market demand

Increases in publicly released refresh rates have grown slowly over the past few years, with respect to vast leaps in power consumption reduction and miniaturization demand. Compared to the processing speed increase of 3 GHz to 4 GHz (2006 too 2014), a new demand for more battery life and reductions in size is the current focus because of the mobile technology being produced. This change in focus can be shown by the significant reductions in power consumption, as much as 50%, that were reported by Intel in their release of the Haswell (microarchitecture); where they dropped their benchmark down to 10-20 watts vs. 30-40 watts in the previous model.

1.5 See also

- Comparison of CPU architectures
- Computer hardware
- CPU design
- Floating point
- Harvard architecture
- Influence of the IBM PC on the personal computer market
- Orthogonal instruction set
- Software architecture
- von Neumann architecture

1.6 Notes

- John L. Hennessy and David Patterson (2006). *Computer Architecture: A Quantitative Approach* (Fourth ed.). Morgan Kaufmann. ISBN 978-0-12-370490-0.
- Barton, Robert S., "Functional Design of Computers", *Communications of the ACM* 4(9): 405 (1961).
- Barton, Robert S., "A New Approach to the Functional Design of a Digital Computer", *Proceedings of the Western Joint Computer Conference*, May 1961, pp. 393–396. About the design of the Burroughs B5000 computer.

- Bell, C. Gordon; and Newell, Allen (1971). "Computer Structures: Readings and Examples", McGraw-Hill.
- Blaauw, G.A., and Brooks, F.P., Jr., "The Structure of System/360, Part I-Outline of the Logical Structure", *IBM Systems Journal*, vol. 3, no. 2, pp. 119–135, 1964.
- Tanenbaum, Andrew S. (1979). *Structured Computer Organization*. Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-148521-0.

1.7 References

[1] Clements, Alan. *Principles of Computer Hardware* (Fourth ed.). p. 1. Architecture describes the internal organization of a computer in an abstract way; that is, it defines the capabilities of the computer and its programming model. You can have two computers that have been constructed in different ways with different technologies but with the same architecture.

[2] Hennessy, John; Patterson, David. *Computer Architecture: A Quantitative Approach* (Fifth ed.). p. 11. This task has many aspects, including instruction set design, functional organization, logic design, and implementation.

[3] Reproduced in B. J. Copeland (Ed.), "Alan Turing's Automatic Computing Engine", OUP, 2005, pp. 369-454.

[4] ACE underwent seven paper designs in one year, before a prototype was initiated in 1948. [B. J. Copeland (Ed.), "Alan Turing's Automatic Computing Engine", OUP, 2005, p. 57]

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach* (Third ed.). Morgan Kaufmann Publishers.

[6] Laplante, Phillip A. (2001). *Dictionary of Computer Science, Engineering, and Technology*. CRC Press. pp. 94–95. ISBN 0-8493-2691-5.

1.8 External links

- ISCA: Proceedings of the International Symposium on Computer Architecture
- Micro: IEEE/ACM International Symposium on Microarchitecture
- HPCA: International Symposium on High Performance Computer Architecture
- ASPLOS: International Conference on Architectural Support for Programming Languages and Operating Systems
- ACM Transactions on Computer Systems
- ACM Transactions on Architecture and Code Optimization
- IEEE Transactions on Computers
- The von Neumann Architecture of Computer Systems

Chapter 2

Complex instruction set computing

Complex instruction set computing (**CISC** /ˈsɪsk/) is a processor design where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions. The term was retroactively coined in contrast to reduced instruction set computer (RISC)[1][2] and has therefore become something of an umbrella term for everything that is not RISC, i.e. everything from large and complex mainframes to simplistic microcontrollers where memory load and store operations are not separated from arithmetic instructions.

A modern RISC processor can therefore be much more complex than, say, a modern microcontroller using a CISC-labeled instruction set, especially in terms of implementation (electronic circuit complexity), but also in terms of the number of instructions or the complexity of their encoding patterns. The only differentiating characteristic (nearly) "guaranteed" is the fact that most RISC designs use uniform instruction length for (almost) all instructions and employ strictly separate load/store-instructions.

Examples of instruction set architectures that have been retroactively labeled CISC are System/360 through z/Architecture, the PDP-11 and VAX architectures, Data General Nova and many others. Well known microprocessors and microcontrollers that have also been labeled CISC in many academic publications include the Motorola 6800, 6809 and 68000-families; the Intel 8080, iAPX432 and x86-family; the Zilog Z80, Z8 and Z8000-families; the National Semiconductor 32016 and NS320xx-line; the MOS Technology 6502-family; the Intel 8051-family; and others.

Some designs have been regarded as borderline cases by some writers. For instance, the Microchip Technology PIC has been labeled RISC in some circles and CISC in others and the 6502 and 6809 have both been described as "RISC-like", although they have complex addressing modes as well as arithmetic instructions that access memory, contrary to the RISC-principles.

2.1 Historical design context

2.1.1 Incitements and benefits

Before the RISC philosophy became prominent, many computer architects tried to bridge the so-called semantic gap, i.e. to design instruction sets that directly supported high-level programming constructs such as procedure calls, loop control, and complex addressing modes, allowing data structure and array accesses to be combined into single instructions. Instructions are also typically highly encoded in order to further enhance the code density. The compact nature of such instruction sets results in smaller program sizes and fewer (slow) main memory accesses, which at the time (early 1960s and onwards) resulted in a tremendous savings on the cost of computer memory and disc storage, as well as faster execution. It also meant good programming productivity even in assembly language, as high level languages such as Fortran or Algol were not always available or appropriate (microprocessors in this category are sometimes still programmed in assembly language for certain types of critical applications).

New instructions

In the 1970s, analysis of high level languages indicated some complex machine language implementations and it was determined that new instructions could improve performance. Some instructions were added that were never intended to be used in assembly language but fit well with compiled high-level languages. Compilers were updated to take advantage of these instructions. The benefits of semantically rich instructions with compact encodings can be seen in modern processors as well, particularly in the high-performance segment where caches are a central component (as opposed to most embedded systems). This is because these fast, but complex and expensive, memories are inherently limited in size, making compact code beneficial. Of course, the fundamental reason they are needed is that main memories (i.e. dynamic RAM today) remain slow compared to a (high performance) CPU core.

2.1.2 Design issues

While many designs achieved the aim of higher throughput at lower cost and also allowed high-level language constructs to be expressed by fewer instructions, it was observed that this was not *always* the case. For instance, low-end versions of complex architectures (i.e. using less hardware) could lead to situations where it was possible to improve performance by *not* using a complex instruction (such as a procedure call or enter instruction), but instead using a sequence of simpler instructions.

One reason for this was that architects (microcode writers) sometimes "over-designed" assembly language instructions, i.e. including features which were not possible to implement efficiently on the basic hardware available. This could, for instance, be "side effects" (above conventional flags), such as the setting of a register or memory location that was perhaps seldom used; if this was done via ordinary (non duplicated) internal buses, or even the *external* bus, it would demand extra cycles every time, and thus be quite inefficient.

Even in balanced high-performance designs, highly encoded and (relatively) high-level instructions could be complicated to decode and execute efficiently within a limited transistor budget. Such architectures therefore required a great deal of work on the part of the processor designer in cases where a simpler, but (typically) slower, solution based on decode tables and/or microcode sequencing is not appropriate. At a time when transistors and other components were a limited resource, this also left fewer components and less opportunity for other types of performance optimizations.

The RISC idea

The circuitry that performs the actions defined by the microcode in many (but not all) CISC processors is, in itself, a processor which in many ways is reminiscent in structure to very early CPU designs. In the early 1970s, this gave rise to ideas to return to simpler processor designs in order to make it more feasible to cope without (*then* relatively large and expensive) ROM tables and/or PLA structures for sequencing and/or decoding. The first (retroactively) RISC-*labeled* processor (IBM 801 – IBM's Watson Research Center, mid-1970s) was a tightly pipelined simple machine originally intended to be used as an internal microcode kernel, or engine, in CISC designs, but also became the processor that introduced the RISC idea to a somewhat larger public. Simplicity and regularity also in the visible instruction set would make it easier to implement overlapping processor stages (pipelining) at the machine code level (i.e. the level seen by compilers). However, pipelining at that level was already used in some high performance CISC "supercomputers" in order to reduce the instruction cycle time (despite the complications of implementing within the limited component count and wiring complexity feasible at the time). Internal microcode execution in CISC processors, on the other hand, could be more or less pipelined depending on the particular design, and therefore more or less akin to the basic structure of RISC processors.

Superscalar

In a more modern context, the complex variable-length encoding used by some of the typical CISC architectures makes it complicated, but still feasible, to build a superscalar implementation of a CISC programming model *directly*; the in-order superscalar original Pentium and the out-of-order superscalar Cyrix 6x86 are well known examples of this. The frequent memory accesses for operands of a typical CISC machine may limit the instruction level parallelism that can be extracted from the code, although this is strongly mediated by the fast cache structures used in modern designs, as well as by other measures. Due to inherently compact and semantically rich instructions, the average amount of work performed per machine code unit (i.e. per byte or bit) is higher for a CISC than a RISC processor, which may give it a significant advantage in a modern cache based implementation.

Transistors for logic, PLAs, and microcode are no longer scarce resources; only large high-speed cache memories are limited by the maximum number of transistors today. Although complex, the transistor count of CISC decoders do not grow exponentially like the total number of transistors per processor (the majority typically used for caches). Together with better tools and enhanced technologies, this has led to new implementations of highly encoded and variable length designs without load-store limitations (i.e. non-RISC). This governs re-implementations of older architectures such as the ubiquitous x86 (see below) as well as new designs for microcontrollers for embedded systems, and similar uses. The superscalar complexity in the case of modern x86 was solved by converting instructions into one or more micro-operations and dynamically issuing those micro-operations, i.e. indirect and dynamic superscalar execution; the Pentium Pro and AMD K5 are early examples of this. It allows a fairly simple superscalar design to be located after the (fairly complex) decoders (and buffers), giving, so to speak, the best of both worlds in many respects.

CISC and RISC terms

The terms CISC and RISC have become less meaningful with the continued evolution of both CISC and RISC designs and implementations. The first highly (or tightly) pipelined x86 implementations, the 486 designs from Intel, AMD, Cyrix, and IBM, supported every instruction that their predecessors did, but achieved *maximum efficiency* only on a fairly simple x86 subset that was only a little more than a typical RISC instruction set (i.e.

without typical RISC *load-store* limitations). The Intel P5 Pentium generation was a superscalar version of these principles. However, modern x86 processors also (typically) decode and split instructions into dynamic sequences of internally buffered micro-operations, which not only helps execute a larger subset of instructions in a pipelined (overlapping) fashion, but also facilitates more advanced extraction of parallelism out of the code stream, for even higher performance.

Contrary to popular simplifications (present also in some academic texts), not all CISCs are microcoded or have "complex" instructions. As CISC became a catch-all term meaning anything that's not a load-store (RISC) architecture, it's not the number of instructions, nor the complexity of the implementation or of the instructions themselves, that define CISC, but the fact that arithmetic instructions also perform memory accesses. Compared to a small 8-bit CISC processor, a RISC floating-point instruction is complex. CISC does not even need to have complex addressing modes; 32 or 64-bit RISC processors may well have more complex addressing modes than small 8-bit CISC processors.

A PDP-10, a PDP-8, an Intel 386, an Intel 4004, a Motorola 68000, a System z mainframe, a Burroughs B5000, a VAX, a Zilog Z80000, and a MOS Technology 6502 all vary wildly in the number, sizes, and formats of instructions, the number, types, and sizes of registers, and the available data types. Some have hardware support for operations like scanning for a substring, arbitrary-precision BCD arithmetic, or transcendental functions, while others have only 8-bit addition and subtraction. But they are all in the CISC category because they have "load-operate" instructions that load and/or store memory contents within the same instructions that perform the actual calculations. For instance, the PDP-8, having only 8 fixed-length instructions and no microcode at all, is a CISC because of *how* the instructions work, PowerPC, which has over 230 instructions (more than some VAXes) and complex internals like register renaming and a reorder buffer is a RISC, while Minimal CISC has 8 instructions, but is clearly a CISC because it combines memory access and computation in the same instructions.

Some of the problems and contradictions in this terminology will perhaps disappear as more systematic terms, such as (***non***) ***load/store***, become more popular and eventually replace the imprecise and slightly counter-intuitive RISC/CISC terms.

2.2 See also

- CPU design
- Computer architecture
- Computer
- CPU
- MISC
- RISC
- ZISC
- VLIW
- Microprocessor

2.3 Notes

- Tanenbaum, Andrew S. (2006) *Structured Computer Organization, Fifth Edition*, Pearson Education, Inc. Upper Saddle River, NJ.

2.4 References

[1] Patterson, D. A.; Ditzel, D. R. (October 1980). "The case for the reduced instruction set computer". *SIGARCH Computer Architecture News* (ACM) **8** (6): 25–33. doi:10.1145/641914.641917.

[2] Lakhe, Pravin R. (June 2013). "A Technology in Most Recent Processor is Complex Reduced Instruction Set Computers (CRISC): A Survey" (PDF). *International Journal of Innovation Research and Studies*. pp. 711–715.

2.5 Further reading

- Mano, M. Morris. *Computer System Architecture (3rd Edition)*. ISBN 978-0131755635.

2.6 External links

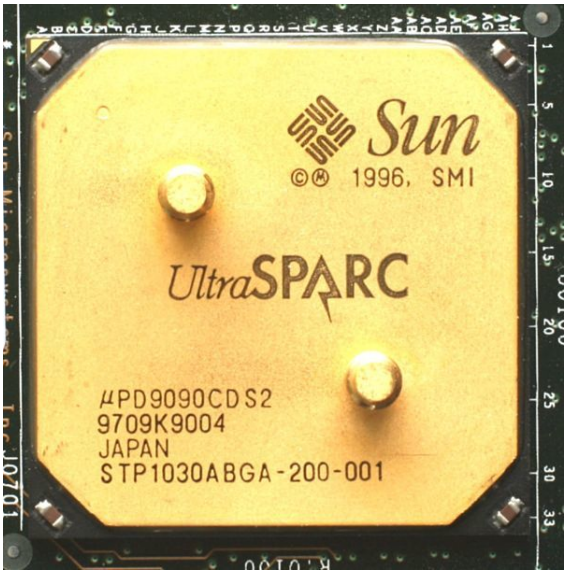
- COSC 243_Computer Architecture 2

Chapter 3

Reduced instruction set computing

"RISC" redirects here. For other uses, see RISC (disambiguation).

Reduced instruction set computing, or **RISC** (pronounced 'risk'), is a CPU design strategy based on the insight that a simplified instruction set (as opposed to a complex set) provides higher performance when combined with a microprocessor architecture capable of executing those instructions using fewer microprocessor cycles per instruction.[1] A computer based on this strategy is a *reduced instruction set computer*, also called *RISC*. The opposing architecture is called complex instruction set computing, i.e. CISC.



A Sun UltraSPARC, a RISC microprocessor

Various suggestions have been made regarding a precise definition of RISC, but the general concept is that of a system that uses a small, highly optimized set of instructions, rather than a more versatile set of instructions often found in other types of architectures. Another common trait is that RISC systems use the load/store architecture,[2] where memory is normally accessed only through specific instructions, rather than accessed as part of other instructions like add.

Although a number of systems from the 1960s and 70s have been identified as being forerunners of RISC, the modern version of the design dates to the 1980s. In particular, two projects at Stanford University and University of California, Berkeley are most associated with the popularization of this concept. Stanford's design would go on to be commercialized as the successful MIPS architecture, while Berkeley's RISC gave its name to the entire concept, commercialized as the SPARC. Another success from this era were IBM's efforts that eventually led to the Power Architecture. As these projects matured, a wide variety of similar designs flourished in the late 1980s and especially the early 1990s, representing a major force in the Unix workstation market as well as embedded processors in laser printers, routers and similar products.

Well-known RISC families include DEC Alpha, AMD Am29000, ARC, ARM, Atmel AVR, Blackfin, Intel i860 and i960, MIPS, Motorola 88000, PA-RISC, Power (including PowerPC), RISC-V, SuperH, and SPARC. In the 21st century, the use of ARM architecture processors in smart phones and tablet computers such as the iPad and Android devices provided a wide user base for RISC-based systems. RISC processors are also used in supercomputers such as the K computer, the fastest on the TOP500 list in 2011, second at the 2012 list, and fourth at the 2013 list,[3][4] and Sequoia, the fastest in 2012 and third in the 2013 list.

3.1 History and development

A number of systems, going back to the 1970s (and even 1960s) have been credited as the first RISC architecture, partly based on their use of load/store approach.[5] The term RISC was coined by David Patterson of the Berkeley RISC project, although somewhat similar concepts had appeared before.[6]

The CDC 6600 designed by Seymour Cray in 1964 used a load/store architecture with only two addressing modes (register+register, and register+immediate constant) and 74 opcodes, with the basic clock cycle/instruction issue rate being 10 times faster than the memory access time.[7] Partly due to the optimized load/store architecture of the CDC 6600 Jack Dongarra states that it can be considered as a forerunner of modern RISC systems, although a num-

ber of other technical barriers needed to be overcome for the development of a modern RISC system.[8]

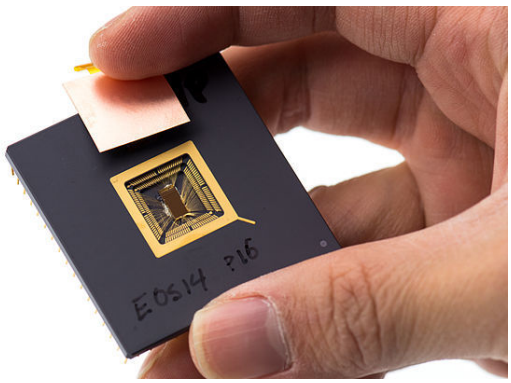
An IBM PowerPC 601 RISC microprocessor.

Michael J. Flynn views the first RISC system as the IBM 801 design which began in 1975 by John Cocke, and completed in 1980.[2] The 801 was eventually produced in a single-chip form as the ROMP in 1981, which stood for 'Research OPD [Office Products Division] Micro Processor'.[9] As the name implies, this CPU was designed for "mini" tasks, and was also used in the IBM RT-PC in 1986, which turned out to be a commercial failure.[10] But the 801 inspired several research projects, including new ones at IBM that would eventually lead to the IBM POWER instruction set architecture.[11][12]

The most public RISC designs, however, were the results of university research programs run with funding from the DARPA VLSI Program. The VLSI Program, practically unknown today, led to a huge number of advances in chip design, fabrication, and even computer graphics. The Berkeley RISC project started in 1980 under the direction of David Patterson and Carlo H. Sequin.[6] [13][14]

Berkeley RISC was based on gaining performance through the use of pipelining and an aggressive use of a technique known as register windowing.[13][14] In a traditional CPU, one has a small number of registers, and a program can use any register at any time. In a CPU with register windows, there are a huge number of registers, e.g. 128, but programs can only use a small number of them, e.g. eight, at any one time. A program that limits itself to eight registers per procedure can make very fast procedure calls: The call simply moves the window "down" by eight, to the set of eight registers used by that procedure, and the return moves the window back.[15] The Berkeley RISC project delivered the RISC-I processor in 1982. Consisting of only 44,420 transistors (compared with averages of about 100,000 in newer CISC designs of the era) RISC-I had only 32 instructions, and yet completely outperformed any other single-chip design. They followed this up with the 40,760 transistor, 39 instruction RISC-II in 1983, which ran over three times as fast as RISC-I.[14]

The MIPS architecture grew out of a graduate course by John L. Hennessy at Stanford University in 1981, resulted in a functioning system in 1983, and could run simple programs by 1984.[16] The MIPS approach emphasized an aggressive clock cycle and the use of the pipeline, making sure it could be run as "full" as possible.[16] The MIPS system was followed by the MIPS-X and in 1984 Hennessy and his colleagues formed MIPS Computer Systems.[16][17] The commercial venture resulted in the R2000 microprocessor in 1985, and was followed by the R3000 in 1988.[17]



Co-designer Yunsup Lee holding RISC-V prototype chip in 2013.

In the early 1980s, significant uncertainties surrounded the RISC concept, and it was uncertain if it could have a commercial future, but by the mid-1980s the concepts had matured enough to be seen as commercially viable.[10][16] In 1986 Hewlett Packard started using an early implementation of their PA-RISC in some of their computers.[10] In the meantime, the Berkeley RISC effort had become so well known that it eventually became the name for the entire concept and in 1987 Sun Microsystems began shipping systems with the SPARC processor, directly based on the Berkeley RISC-II system.[10][19]

The US government Committee on Innovations in Computing and Communications credits the acceptance of the viability of the RISC concept to the success of the SPARC system.[10] The success of SPARC renewed interest within IBM, which released new RISC systems by 1990 and by 1995 RISC processors were the foundation of a $15 billion server industry.[10]

Since 2010 a new open source ISA, RISC-V, is under development at the University of California, Berkeley, for research purposes and as a free alternative to proprietary ISAs. As of 2014 version 2 of the userspace ISA is fixed.[20] The ISA is designed to be extensible from a barebones core sufficient for a small embedded processor to supercomputer and cloud computing use with standard and chip designer defined extensions and coprocessors. It has been tested in silicon design with the ROCKET SoC

which is also available as an open source processor generator in the CHISEL language.

3.2 Characteristics and design philosophy

For more details on this topic, see CPU design.

3.2.1 Instruction set philosophy

A common misunderstanding of the phrase "reduced instruction set computer" is the mistaken idea that instructions are simply eliminated, resulting in a smaller set of instructions.[21] In fact, over the years, RISC instruction sets have grown in size, and today many of them have a larger set of instructions than many CISC CPUs.[22][23] Some RISC processors such as the PowerPC have instruction sets as large as the CISC IBM System/370, for example; conversely, the DEC PDP-8—clearly a CISC CPU because many of its instructions involve multiple memory accesses—has only 8 basic instructions and a few extended instructions.

The term "reduced" in that phrase was intended to describe the fact that the amount of work any single instruction accomplishes is reduced—at most a single data memory cycle—compared to the "complex instructions" of CISC CPUs that may require dozens of data memory cycles in order to execute a single instruction.[24] In particular, RISC processors typically have separate instructions for I/O and data processing.

The term load/store architecture is sometimes preferred.

3.2.2 Instruction format

Most RISC machines used a fixed length instruction(e.g. 32 bits) and layout, with more predictable encodings, which simplifies fetch and interdependency logic considerably; Several, such as ARM, Power ISA, MIPS, RISC-V, and the Adapteva Epiphany, have an optional compressed instruction option to work around the problem of reduced code density. The SH5 also follows this pattern, albeit having evolved in the opposite direction, having added longer media instructions to an original 16bit encoding.

3.2.3 Hardware utilization

For any given level of general performance, a RISC chip will typically have far fewer transistors dedicated to the core logic which originally allowed designers to increase the size of the register set and increase internal parallelism.

Other features that are typically found in RISC architectures are:

- Uniform instruction format, using a single word with the opcode in the same bit positions in every instruction, demanding less decoding;
- Identical general purpose registers, allowing any register to be used in any context, simplifying compiler design (although normally there are separate floating point registers);
- Simple addressing modes, with complex addressing performed via sequences of arithmetic, load-store operations, or both;
- Few data types in hardware, some CISCs have byte string instructions, or support complex numbers; this is so far unlikely to be found on a RISC.
- Processor throughput of one instruction per cycle on average

Exceptions abound, of course, within both CISC and RISC.

RISC designs are also more likely to feature a Harvard memory model, where the instruction stream and the data stream are conceptually separated; this means that modifying the memory where code is held might not have any effect on the instructions executed by the processor (because the CPU has a separate instruction and data cache), at least until a special synchronization instruction is issued. On the upside, this allows both caches to be accessed simultaneously, which can often improve performance.

Many early RISC designs also shared the characteristic of having a branch delay slot. A branch delay slot is an instruction space immediately following a jump or branch. The instruction in this space is executed, whether or not the branch is taken (in other words the effect of the branch is delayed). This instruction keeps the ALU of the CPU busy for the extra time normally needed to perform a branch. Nowadays the branch delay slot is considered an unfortunate side effect of a particular strategy for implementing some RISC designs, and modern RISC designs generally do away with it (such as PowerPC and more recent versions of SPARC and MIPS).

Some aspects attributed to the first RISC-*labeled* designs around 1975 include the observations that the memory-restricted compilers of the time were often unable to take advantage of features intended to facilitate *manual* assembly coding, and that complex addressing modes take many cycles to perform due to the required additional memory accesses. It was argued that such functions would be better performed by sequences of simpler instructions if this could yield implementations small enough to leave room for many registers, reducing the

number of slow memory accesses. In these simple designs, most instructions are of uniform length and similar structure, arithmetic operations are restricted to CPU registers and only separate *load* and *store* instructions access memory. These properties enable a better balancing of pipeline stages than before, making RISC pipelines significantly more efficient and allowing higher clock frequencies.

In the early days of the computer industry, programming was done in assembly language or machine code, which encouraged powerful and easy-to-use instructions. CPU designers therefore tried to make instructions that would do as much work as feasible. With the advent of higher level languages, computer architects also started to create dedicated instructions to directly implement certain central mechanisms of such languages. Another general goal was to provide every possible addressing mode for every instruction, known as orthogonality, to ease compiler implementation. Arithmetic operations could therefore often have results as well as operands directly in memory (in addition to register or immediate).

The attitude at the time was that hardware design was more mature than compiler design so this was in itself also a reason to implement parts of the functionality in hardware or microcode rather than in a memory constrained compiler (or its generated code) alone. After the advent of RISC, this philosophy became retroactively known as complex instruction set computing, or CISC.

CPUs also had relatively few registers, for several reasons:

- More registers also implies more time-consuming saving and restoring of register contents on the machine stack.
- A large number of registers requires a large number of instruction bits as register specifiers, meaning less dense code (see below).
- CPU registers are more expensive than external memory locations; large register sets were cumbersome with limited circuit boards or chip integration.

An important force encouraging complexity was very limited main memories (on the order of kilobytes). It was therefore advantageous for the code density—the density of information held in computer programs—to be high, leading to features such as highly encoded, variable length instructions, doing data loading as well as calculation (as mentioned above). These issues were of higher priority than the ease of decoding such instructions.

An equally important reason was that main memories were quite slow (a common type was ferrite core memory); by using dense information packing, one could reduce the frequency with which the CPU had to access this slow resource. Modern computers face similar limiting factors: main memories are slow compared to the CPU and the fast cache memories employed to overcome this are limited in size. This may partly explain why highly encoded instruction sets have proven to be as useful as RISC designs in modern computers.

RISC was developed as an alternative to what is now known as CISC. Over the years, other strategies have been implemented as alternatives to RISC and CISC. Some examples are VLIW, MISC, OISC, massive parallel processing, systolic array, reconfigurable computing, and dataflow architecture.

In the mid-1970s, researchers (particularly John Cocke) at IBM (and similar projects elsewhere) demonstrated that the majority of combinations of these orthogonal addressing modes and instructions were not used by most programs generated by compilers available at the time. It proved difficult in many cases to write a compiler with more than limited ability to take advantage of the features provided by conventional CPUs.

It was also discovered that, on microcoded implementations of certain architectures, complex operations tended to be *slower* than a sequence of simpler operations doing the same thing. This was in part an effect of the fact that many designs were rushed, with little time to optimize or tune every instruction; only those used most often were optimized, and a sequence of those instructions could be faster than a less-tuned instruction performing an equivalent operation as that sequence. One infamous example was the VAX's INDEX instruction.[13]

As mentioned elsewhere, core memory had long since been slower than many CPU designs. The advent of semiconductor memory reduced this difference, but it was still apparent that more registers (and later caches) would allow higher CPU operating frequencies. Additional registers would require sizeable chip or board areas which, at the time (1975), could be made available if the complexity of the CPU logic was reduced.

Yet another impetus of both RISC and other designs came from practical measurements on real-world programs. Andrew Tanenbaum summed up many of these, demonstrating that processors often had oversized immediates. For instance, he showed that 98% of all the constants in a program would fit in 13 bits, yet many CPU designs dedicated 16 or 32 bits to store them. This suggests that, to reduce the number of memory accesses, a fixed length machine could store constants in unused bits of the instruction word itself, so that they would be immediately ready when the CPU needs them (much like immediate addressing in a conventional design). This required small opcodes in order to leave room for a reasonably sized constant in a 32-bit instruction word.

Since many real-world programs spend most of their time executing simple operations, some researchers decided to focus on making those operations as fast as possible. The clock rate of a CPU is limited by the time it takes to execute the slowest *sub-operation* of any instruction; decreasing that cycle-time often accelerates the execution of other instructions.[25] The focus on “reduced instruc-

tions" led to the resulting machine being called a "reduced instruction set computer" (RISC). The goal was to make instructions so simple that they could *easily* be pipelined, in order to achieve a *single clock* throughput at *high frequencies*.

Later, it was noted that one of the most significant characteristics of RISC processors was that external memory was only accessible by a *load* or *store* instruction. All other instructions were limited to internal registers. This simplified many aspects of processor design: allowing instructions to be fixed-length, simplifying pipelines, and isolating the logic for dealing with the delay in completing a memory access (cache miss, etc.) to only two instructions. This led to RISC designs being referred to as *load/store* architectures.[26]

One more issue is that some complex instructions are difficult to restart, e.g. following a page fault. In some cases, restarting from the beginning will work (although wasteful), but in many cases this would give incorrect results. Therefore, the machine needs to have some hidden state to remember which parts went through and what remains to be done. With a load/store machine, the program counter is sufficient to describe the state of the machine.

The main distinguishing feature of RISC is that the instruction set is optimized for a highly regular instruction pipeline flow.[21] All the other features associated with RISC—branch delay slots, separate instruction and data caches, load/store architecture, large register set, etc.—may seem to be a random assortment of unrelated features, but each of them is helpful in maintaining a regular pipeline flow that completes an instruction every clock cycle.

3.3 Comparison to other architectures

Some CPUs have been specifically designed to have a very small set of instructions – but these designs are very different from classic RISC designs, so they have been given other names such as minimal instruction set computer (MISC), or transport triggered architecture (TTA), etc.

Despite many successes, RISC has made few inroads into the desktop PC and commodity server markets, where Intel's x86 platform remains the dominant processor architecture. There are three main reasons for this:

1. A very large base of proprietary PC applications are written for x86 or compiled into x86 machine code, whereas no RISC platform has a similar installed base; hence PC users were locked into the x86.

2. Although RISC was indeed able to scale up in performance quite quickly and cheaply, Intel took advantage of its large market by spending vast amounts of money on processor development. Intel could spend many times as much as any RISC manufacturer on improving low level design and manufacturing. The same could not be said about smaller firms like Cyrix and NexGen, but they realized that they could apply (tightly) pipelined design practices also to the x86-architecture, just as in the 486 and Pentium. The 6x86 and MII series did exactly this, but was more advanced; it implemented superscalar speculative execution via register renaming, directly at the x86-semantic level. Others, like the Nx586 and AMD K5 did the same, but *indirectly*, via dynamic microcode buffering and semi-independent superscalar scheduling and instruction dispatch at the micro-operation level (older or simpler 'CISC' designs typically execute rigid micro-operation sequences directly). The first *available* chip deploying such dynamic buffering and scheduling techniques was the NexGen Nx586, released in 1994; the AMD K5 was severely delayed and released in 1995.

3. Later, more powerful processors, such as Intel P6, AMD K6, AMD K7, and Pentium 4, employed similar dynamic buffering and scheduling principles and implemented loosely coupled superscalar (and speculative) execution of micro-operation sequences generated from several parallel x86 decoding stages. Today, these ideas have been further refined (some x86-pairs are instead merged into a more complex micro-operation, for example) and are still used by modern x86 processors such as Intel Core 2 and AMD K8.

Outside of the desktop arena, however, the ARM architecture (RISC and born at about the same time as SPARC) has to a degree broken the Intel stranglehold with its widespread use in smartphones, tablets and many forms of embedded device. It is also the case that since the Pentium Pro (P6) Intel has been using an internal RISC processor core for its processors.[27]

While early RISC designs differed significantly from contemporary CISC designs, by 2000 the highest performing CPUs in the RISC line were almost indistinguishable from the highest performing CPUs in the CISC line.[28][29][30]

3.4 Use of RISC architectures

RISC architectures are now used across a wide range of platforms, from cellular telephones and tablet computers to some of the world's fastest supercomputers such as the K computer, the fastest on the TOP500 list in 2011.[3][4]

3.4.1 Low end and mobile systems

By the beginning of the 21st century, the majority of low end and mobile systems relied on RISC architectures.[31] Examples include:

- The ARM architecture dominates the market for low power and low cost embedded systems (typically 200–1800 MHz in 2014). It is used in a number of systems such as most Android-based systems, the Apple iPhone and iPad, Microsoft Windows Phone (former Windows Mobile), RIM devices, Nintendo Game Boy Advance and Nintendo DS, etc.
- The MIPS line, (at one point used in many SGI computers) and now in the PlayStation, PlayStation 2, Nintendo 64, PlayStation Portable game consoles, and residential gateways like Linksys WRT54G series.
- Hitachi's SuperH, originally in wide use in the Sega Super 32X, Saturn and Dreamcast, now developed and sold by Renesas as the SH4
- Atmel AVR used in a variety of products ranging from Xbox handheld controllers to BMW cars.
- RISC-V, the open source fifth Berkeley RISC ISA, with 32 bit address space a small core integer instruction set, an experimental "Compressed" ISA for code density and designed for standard and special purpose extensions.

3.4.2 High end RISC and supercomputing

- MIPS, by Silicon Graphics (ceased making MIPS-based systems in 2006).
- SPARC, by Oracle (previously Sun Microsystems), and Fujitsu.
- IBM's Power Architecture, used in many of IBM's supercomputers, midrange servers and workstations.
- Hewlett-Packard's PA-RISC, also known as HP-PA (discontinued at the end of 2008).
- Alpha, used in single-board computers, workstations, servers and supercomputers from Digital Equipment Corporation, Compaq and HP (discontinued as of 2007).
- RISC-V, the open source fifth Berkeley RISC ISA, with 64 or 128-bit address spaces, and the integer core extended with floating point, atomics and vector processing, and designed to be extended with instructions for networking, IO, data processing etc. A 64-bit superscalar design, "Rocket", is available for download.

3.5 See also

- Addressing mode
- Classic RISC pipeline
- Complex instruction set computer
- Computer architecture
- Instruction set
- Microprocessor
- Minimal instruction set computer

3.6 References

[1] Northern Illinois University, Department of Computer Science, "RISC - Reduced instruction set computer"

[2] Flynn, Michael J. (1995). *Computer architecture: pipelined and parallel processor design*. pp. 54–56. ISBN 0867202041.

[3] "Japanese 'K' Computer Is Ranked Most Powerful". *The New York Times*. 20 June 2011. Retrieved 20 June 2011.

[4] "Supercomputer "K computer" Takes First Place in World". Fujitsu. Retrieved 20 June 2011.

[5] Fisher, Joseph A.; Faraboschi, Paolo; Young, Cliff (2005). *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. p. 55. ISBN 1558607668.

[6] *Milestones in computer science and information technology* by Edwin D. Reilly 2003 ISBN 1-57356-521-0 page 50

[7] Grishman, Ralph. Assembly Language Programming for the Control Data 6000 Series. Algorithmics Press. 1974. pg 12

[8] *Numerical Linear Algebra on High-Performance Computers* by Jack J. Dongarra, et al 1987 ISBN 0-89871-428-1 page 6

[9] *Processor architecture: from dataflow to superscalar and beyond* by Jurij Šilc, Borut Robič, Theo Ungerer 1999 ISBN 3-540-64798-8 page 33

[10] *Funding a Revolution: Government Support for Computing Research* by Committee on Innovations in Computing and Communications 1999 ISBN 0-309-06278-0 page 239

[11] *Processor design: system-on-chip computing for ASICs and FPGAs* by Jari Nurmi 2007 ISBN 1-4020-5529-3 pages 40-43

[12] *Readings in computer architecture* by Mark Donald Hill, Norman Paul Jouppi, Gurindar Sohi 1999 ISBN 1-55860-539-8 pages 252-254

[13] Patterson, D. A.; Ditzel, D. R. (1980). "The case for the reduced instruction set computer". *ACM SIGARCH Computer Architecture News* **8** (6): 25–33. doi:10.1145/641914.641917. CiteSeerX: 10.1.1.68.9623.

[14] *RISC I: A Reduced Instruction Set VLSI Computer* by David A. Patterson and Carlo H. Sequin, in the Proceedings of the 8th annual symposium on Computer Architecture, 1981.

[15] *Design and Implementation of RISC I by Carlo Sequin and David Patterson, in the Proceedings of the Advanced Course on VLSI Architecture, University of Bristol, July 1982*

[16] *The MIPS-X RISC microprocessor* by Paul Chow 1989 ISBN 0-7923-9045-8 pages xix-xx

[17] *Processor design: system-on-chip computing for ASICs and FPGAs* by Jari Nurmi 2007 ISBN 1-4020-5529-3 pages 52-53

[18] "Joseph H. Condon". Princeton University History of Science.

[19] *Computer science handbook* by Allen B. Tucker 2004 ISBN 1-58488-360-X page 100-6

[20] Waterman, Andrew; Lee, Yunsup; Patterson, David A.; Asanovi, Krste. "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2 (Technical Report EECS-2014-54)". University of California, Berkeley. Retrieved 26 Dec 2014.

[21] Margarita Esponda and Ra'ul Rojas. "The RISC Concept - A Survey of Implementations". Section 2: "The confusion around the RISC concept". 1991.

[22] "RISC vs. CISC: the Post-RISC Era" by Jon "Hannibal" Stokes (Arstechnica)

[23] "RISC versus CISC" by Lloyd Borrett Australian Personal Computer, June 1991

[24] "Guide to RISC Processors for Programmers and Engineers": Chapter 3: "RISC Principles" by Sivarama P. Dandamudi, 2005, ISBN 978-0-387-21017-9. "the main goal was not to reduce the number of instructions, but the complexity"

[25] "Microprocessors From the Programmer's Perspective" by Andrew Schulman 1990

[26] Kevin Dowd. High Performance Computing. O'Reilly & Associates, Inc. 1993.

[27] "Intel x86 Processors – CISC or RISC? Or both??" by Sundar Srinivasan

[28] "Schaum's Outline of Computer Architecture" by Nicholas P. Carter 2002 p. 96 ISBN 0-07-136207-X

[29] "CISC, RISC, and DSP Microprocessors" by Douglas L. Jones 2000

[30] "A History of Apple's Operating Systems" by Amit Singh. "the line between RISC and CISC has been growing fuzzier over the years."

[31] *Guide to RISC processors: for programmers and engineers* by Sivarama P. Dandamudi - 2005 ISBN 0-387-21017-2 pages 121-123

3.7 External links

- RISC vs. CISC
- What is RISC
- The RISC-V Instruction Set Architecture
- Not Quite RISC

Chapter 4

History of general-purpose CPUs

The **history of general-purpose CPUs** is a continuation of the earlier history of computing hardware.

A Vacuum tube module from early 700 series IBM computers

4.1 1950s: early designs

Each of the computer designs of the early 1950s was a unique design; there were no upward-compatible machines or computer architectures with multiple, differing implementations. Programs written for one machine would not run on another kind, even other kinds from the same company. This was not a major drawback at the time because there was not a large body of software developed to run on computers, so starting programming from scratch was not seen as a large barrier.

The design freedom of the time was very important, for designers were very constrained by the cost of electronics, yet just beginning to explore how a computer could best be organized. Some of the basic features introduced during this period included index registers (on the Ferranti Mark 1), a return-address saving instruction (UNIVAC I), immediate operands (IBM 704), and the detection of invalid operations (IBM 650).

By the end of the 1950s commercial builders had developed factory-constructed, truck-deliverable computers. The most widely installed computer was the IBM 650, which used drum memory onto which programs were loaded using either paper tape or punched cards. Some very high-end machines also included core memory which provided higher speeds. Hard disks were also starting to become popular.

A computer is an automatic abacus. The type of number system affects the way it works. In the early 1950s most computers were built for specific numerical processing tasks, and many machines used decimal numbers as their basic number system – that is, the mathematical functions of the machines worked in base-10 instead of base-2 as is common today. These were not merely binary coded decimal. Most machines actually had ten vacuum tubes per digit in each register. Some early Soviet computer designers implemented systems based on ternary logic; that is, a bit could have three states: +1, 0, or −1, corresponding to positive, zero, or negative voltage.

An early project for the U.S. Air Force, BINAC attempted to make a lightweight, simple computer by using binary arithmetic. It deeply impressed the industry.

As late as 1970, major computer languages were unable to standardize their numeric behavior because decimal computers had groups of users too large to alienate.

Even when designers used a binary system, they still had many odd ideas. Some used sign-magnitude arithmetic (−1 = 10001), or ones' complement (−1 = 11110), rather than modern two's complement arithmetic (−1 = 11111). Most computers used six-bit character sets, because they adequately encoded Hollerith cards. It was a major revelation to designers of this period to realize that the data word should be a multiple of the character size. They began to design computers with 12, 24 and 36 bit data words (e.g. see the TX-2).

In this era, Grosch's law dominated computer design: Computer cost increased as the square of its speed.

4.2 1960s: the computer revolution and CISC

One major problem with early computers was that a program for one would not work on others. Computer companies found that their customers had little reason to remain loyal to a particular brand, as the next computer they purchased would be incompatible anyway. At that point, price and performance were usually the only concerns.

In 1962, IBM tried a new approach to designing computers. The plan was to make an entire family of computers that could all run the same software, but with different performances, and at different prices. As users' requirements grew they could move up to larger computers, and still keep all of their investment in programs, data and storage media.

In order to do this they designed a single *reference computer* called the **System/360** (or **S/360**). The System/360 was a virtual computer, a reference instruction set and capabilities that all machines in the family would support. In order to provide different classes of machines, each computer in the family would use more or less hardware emulation, and more or less microprogram emulation, to create a machine capable of running the entire System/360 instruction set.

For instance a low-end machine could include a very simple processor for low cost. However this would require the use of a larger microcode emulator to provide the rest of the instruction set, which would slow it down. A high-end machine would use a much more complex processor that could directly process more of the System/360 design, thus running a much simpler and faster emulator.

IBM chose to make the reference instruction set quite complex, and very capable. This was a conscious choice. Even though the computer was complex, its "control store" containing the microprogram would stay relatively small, and could be made with very fast memory. Another important effect was that a single instruction could describe quite a complex sequence of operations. Thus the computers would generally have to fetch fewer instructions from the main memory, which could be made slower, smaller and less expensive for a given combination of speed and price.

As the S/360 was to be a successor to both scientific machines like the 7090 and data processing machines like the 1401, it needed a design that could reasonably support all forms of processing. Hence the instruction set was designed to manipulate not just simple binary numbers, but text, scientific floating-point (similar to the numbers used in a calculator), and the binary coded decimal arithmetic needed by accounting systems.

Almost all following computers included these innovations in some form. This basic set of features is now called a "Complex Instruction Set Computer," or CISC (pronounced "sisk"), a term not invented until many years later, when RISC (Reduced Instruction Set Computer) began to get market share.

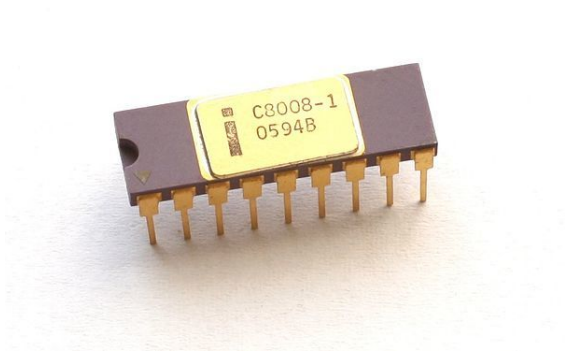
In many CISCs, an instruction could access either registers or memory, usually in several different ways. This made the CISCs easier to program, because a programmer could remember just thirty to a hundred instructions, and a set of three to ten addressing modes rather than thousands of distinct instructions. This was called an "orthogonal instruction set." The PDP-11 and Motorola 68000 architecture are examples of nearly orthogonal instruction sets.

There was also the *BUNCH* (Burroughs, UNIVAC, NCR, Control Data Corporation, and Honeywell) that competed against IBM at this time; however, IBM dominated the era with S/360.

The Burroughs Corporation (which later merged with Sperry/Univac to become Unisys) offered an alternative to S/360 with their B5000 series machines. In 1961, the B5000 had virtual memory, symmetric multiprocessing, a multi-programming operating system (Master Control Program or MCP), written in ALGOL 60, and the industry's first recursive-descent compilers as early as 1963.

4.3 1970s: Large Scale Integration

In the 1960s, the Apollo guidance computer and Minuteman missile made the integrated circuit economical and practical.



An Intel 8008 Microprocessor

Around 1970, the first calculator and clock chips began to show that very small computers might be possible. One of the first commercially available microprocessors was the BCD based Intel 4004, designed in 1970 for the Japanese calculator company Busicom. A little more than a year later, in march of 1972, Intel introduced a microprocessor with a totally different and 8-bit based architecture, the 8008, an integrated pMOS re-implementation of the TTL-based based Datapoint 2200 CPU. Via the 8008, 8080 and the 8086 line of designs, the 2200 became a direct ancestor to today's ubiquitous x86 family (including Pentium, Core i7 etc.); every instruction of the 2200 or

8008 has a direct equivalent in the large x86 instruction set, although the opcode values are different in the latter.

By the mid-1970s, the use of integrated circuits in computers was commonplace. The whole decade consists of upheavals caused by the shrinking price of transistors.

It became possible to put an entire CPU on a single printed circuit board. The result was that minicomputers, usually with 16-bit words, and 4k to 64K of memory, came to be commonplace.

CISCs were believed to be the most powerful types of computers, because their microcode was small and could be stored in very high-speed memory. The CISC architecture also addressed the "semantic gap" as it was perceived at the time. This was a defined distance between the machine language, and the higher level language people used to program a machine. It was felt that compilers could do a better job with a richer instruction set.

Custom CISCs were commonly constructed using "bit slice" computer logic such as the AMD 2900 chips, with custom microcode. A bit slice component is a piece of an ALU, register file or microsequencer. Most bit-slice integrated circuits were 4-bits wide.

By the early 1970s, the PDP-11 was developed, arguably the most advanced small computer of its day. Almost immediately, wider-word CISCs were introduced, the 32-bit VAX and 36-bit PDP-10.

Intel soon developed a slightly more mini computer-like microprocessor, the 8080, largely based on customer feedback on the limited 8008. Much like the 8008, it was used for applications such as terminals, printers, cash registers and industrial robots. However, the more capable 8080 also became the original target CPU for an early de facto standard personal computer operating system called CP/M and was used for such demanding control tasks as cruise missiles, as well as many other kinds of applications. The 8080 became one of the first really widespread microprocessors.

IBM continued to make large, fast computers. However the definition of large and fast now meant more than a megabyte of RAM, clock speeds near one megahertz , and tens of megabytes of disk drives.

IBM's System 370 was a version of the 360 tweaked to run virtual computing environments. The virtual computer was developed in order to reduce the possibility of an unrecoverable software failure.

The Burroughs B5000/B6000/B7000 series reached its largest market share. It was a stack computer whose OS was programmed in a dialect of Algol.

All these different developments competed for market share.

4.4 Early 1980s: the lessons of RISC

In the early 1980s, researchers at UC Berkeley and IBM both discovered that most computer language compilers and interpreters used only a small subset of the instructions of a CISC. Much of the power of the CPU was simply being ignored in real-world use. They realized that by making the computer simpler and less orthogonal, they could make it faster and less expensive at the same time.

At the same time, CPU calculation became faster in relation to the time for necessary memory accesses. Designers also experimented with using large sets of internal registers. The idea was to cache intermediate results in the registers under the control of the compiler. This also reduced the number of addressing modes and orthogonality.

The computer designs based on this theory were called Reduced Instruction Set Computers, or RISC. RISCs generally had larger numbers of registers, accessed by simpler instructions, with a few instructions specifically to load and store data to memory. The result was a very simple core CPU running at very high speed, supporting the exact sorts of operations the compilers were using anyway.

A common variation on the RISC design employs the Harvard architecture, as opposed to the Von Neumann or Stored Program architecture common to most other designs. In a Harvard Architecture machine, the program and data occupy separate memory devices and can be accessed simultaneously. In Von Neumann machines the data and programs are mixed in a single memory device, requiring sequential accessing which produces the so-called "Von Neumann bottleneck."

One downside to the RISC design has been that the programs that run on them tend to be larger. This is because compilers have to generate longer sequences of the simpler instructions to accomplish the same results. Since these instructions need to be loaded from memory anyway, the larger code size offsets some of the RISC design's fast memory handling.

Recently, engineers have found ways to compress the reduced instruction sets so they fit in even smaller memory systems than CISCs. Examples of such compression schemes include the ARM's "Thumb" instruction set. In applications that do not need to run older binary software, compressed RISCs are coming to dominate sales.

Another approach to RISCs was the MISC, "niladic" or "zero-operand" instruction set. This approach realized that the majority of space in an instruction was used to identify the operands of the instruction. These machines placed the operands on a push-down (last-in, first out) stack. The instruction set was supplemented with a few instructions to fetch and store memory. Most used simple caching to provide extremely fast RISC machines, with

very compact code. Another benefit was that the interrupt latencies were extremely small, smaller than most CISC machines (a rare trait in RISC machines). The Burroughs large systems architecture uses this approach. The B5000 was designed in 1961, long before the term "RISC" was invented. The architecture puts six 8-bit instructions in a 48-bit word, and was a precursor to VLIW design (see below: 1990 to Today).

The Burroughs architecture was one of the inspirations for Charles H. Moore's Forth programming language, which in turn inspired his later MISC chip designs. For example, his f20 cores had 31 5-bit instructions, which fit four to a 20-bit word.

RISC chips now dominate the market for 32-bit embedded systems. Smaller RISC chips are even becoming common in the cost-sensitive 8-bit embedded-system market. The main market for RISC CPUs has been systems that require low power or small size.

Even some CISC processors (based on architectures that were created before RISC became dominant), such as newer x86 processors, translate instructions internally into a RISC-like instruction set.

These numbers may surprise many, because the "market" is perceived to be desktop computers. x86 designs dominate desktop and notebook computer sales, but desktop and notebook computers are only a tiny fraction of the computers now sold. Most people in industrialised countries own more computers in embedded systems in their car and house than on their desks.

4.5 Mid-to-late 1980s: exploiting instruction level parallelism

In the mid-to-late 1980s, designers began using a technique known as "instruction pipelining", in which the processor works on multiple instructions in different stages of completion. For example, the processor may be retrieving the operands for the next instruction while calculating the result of the current one. Modern CPUs may use over a dozen such stages. MISC processors achieve single-cycle execution of instructions without the need for pipelining.

A similar idea, introduced only a few years later, was to execute multiple instructions in parallel on separate arithmetic logic units (ALUs). Instead of operating on only one instruction at a time, the CPU will look for several similar instructions that are not dependent on each other, and execute them in parallel. This approach is called superscalar processor design.

Such techniques are limited by the degree of instruction level parallelism (ILP), the number of non-dependent instructions in the program code. Some programs are able to run very well on superscalar processors due to their inherent high ILP, notably graphics. However more general problems do not have such high ILP, thus making the achievable speedups due to these techniques to be lower.

Branching is one major culprit. For example, the program might add two numbers and branch to a different code segment if the number is bigger than a third number. In this case even if the branch operation is sent to the second ALU for processing, it still must wait for the results from the addition. It thus runs no faster than if there were only one ALU. The most common solution for this type of problem is to use a type of branch prediction.

To further the efficiency of multiple functional units which are available in superscalar designs, operand register dependencies was found to be another limiting factor. To minimize these dependencies, out-of-order execution of instructions was introduced. In such a scheme, the instruction results which complete out-of-order must be re-ordered in program order by the processor for the program to be restartable after an exception. *Out-of-Order* execution was the main advancement of the computer industry during the 1990s. A similar concept is speculative execution, where instructions from one direction of a branch (the predicted direction) are executed before the branch direction is known. When the branch direction is known, the predicted direction and the actual direction are compared. If the predicted direction was correct, the speculatively executed instructions and their results are kept; if it was incorrect, these instructions and their results are thrown out. Speculative execution coupled with an accurate branch predictor gives a large performance gain.

These advances, which were originally developed from research for RISC-style designs, allow modern CISC processors to execute twelve or more instructions per clock cycle, when traditional CISC designs could take twelve or more cycles to execute just one instruction.

The resulting instruction scheduling logic of these processors is large, complex and difficult to verify. Furthermore, the higher complexity requires more transistors, increasing power consumption and heat. In this respect RISC is superior because the instructions are simpler, have less interdependence and make superscalar implementations easier. However, as Intel has demonstrated, the concepts can be applied to a CISC design, given enough time and money.

> Historical note: Some of these techniques (e.g. pipelining) were originally developed in the late 1950s by IBM on their Stretch mainframe computer.

4.6 1990 to today: looking forward

4.6.1 VLIW and EPIC

The instruction scheduling logic that makes a superscalar processor is just boolean logic. In the early 1990s, a significant innovation was to realize that the coordination of a multiple-ALU computer could be moved into the compiler, the software that translates a programmer's instructions into machine-level instructions.

This type of computer is called a **very long instruction word** (VLIW) computer.

Statically scheduling the instructions in the compiler (as opposed to letting the processor do the scheduling dynamically) can reduce CPU complexity. This can improve performance, reduce heat, and reduce cost.

Unfortunately, the compiler lacks accurate knowledge of runtime scheduling issues. Merely changing the CPU core frequency multiplier will have an effect on scheduling. Actual operation of the program, as determined by input data, will have major effects on scheduling. To overcome these severe problems a VLIW system may be enhanced by adding the normal dynamic scheduling, losing some of the VLIW advantages.

Static scheduling in the compiler also assumes that dynamically generated code will be uncommon. Prior to the creation of Java, this was in fact true. It was reasonable to assume that slow compiles would only affect software developers. Now, with JIT virtual machines being used for many languages, slow code generation affects users as well.

There were several unsuccessful attempts to commercialize VLIW. The basic problem is that a VLIW computer does not scale to different price and performance points, as a dynamically scheduled computer can. Another issue is that compiler design for VLIW computers is extremely difficult, and the current crop of compilers (as of 2005) don't always produce optimal code for these platforms.

Also, VLIW computers optimise for throughput, not low latency, so they were not attractive to the engineers designing controllers and other computers embedded in machinery. The embedded systems markets had often pioneered other computer improvements by providing a large market that did not care about compatibility with older software.

In January 2000, Transmeta Corporation took the interesting step of placing a compiler in the central processing unit, and making the compiler translate from a reference byte code (in their case, x86 instructions) to an internal VLIW instruction set. This approach combines the hardware simplicity, low power and speed of VLIW RISC with the compact main memory system and software reverse-compatibility provided by popular CISC.

Intel's Itanium chip is based on what they call an Explicitly Parallel Instruction Computing (EPIC) design. This design supposedly provides the VLIW advantage of increased instruction throughput. However, it avoids some of the issues of scaling and complexity, by explicitly providing in each "bundle" of instructions information concerning their dependencies. This information is calculated by the compiler, as it would be in a VLIW design. The early versions are also backward-compatible with current x86 software by means of an on-chip emulation mode. Integer performance was disappointing and despite improvements, sales in volume markets continue to be low.

4.6.2 Multi-threading

Current designs work best when the computer is running only a single program, however nearly all modern operating systems allow the user to run multiple programs at the same time. For the CPU to change over and do work on another program requires expensive context switching. In contrast, multi-threaded CPUs can handle instructions from multiple programs at once.

To do this, such CPUs include several sets of registers. When a context switch occurs, the contents of the "working registers" are simply copied into one of a set of registers for this purpose.

Such designs often include thousands of registers instead of hundreds as in a typical design. On the downside, registers tend to be somewhat expensive in chip space needed to implement them. This chip space might otherwise be used for some other purpose.

4.6.3 Multi-core

Multi-core CPUs are typically multiple CPU cores on the same die, connected to each other via a shared L2 or L3 cache, an on-die bus, or an on-die crossbar switch. All the CPU cores on the die share interconnect components with which to interface to other processors and the rest of the system. These components may include a front side bus interface, a memory controller to interface with DRAM, a cache coherent link to other processors, and a non-coherent link to the southbridge and I/O devices. The terms multi-core and MPU (which stands for **Micro-Processor Unit**) have come into general usage for a single die that contains multiple CPU cores.

Intelligent RAM

One way to work around the Von Neumann bottleneck is to mix a processor and DRAM all on one chip.

- The Berkeley IRAM Project
- eDRAM
- computational RAM
- Memristor

4.6.4 Reconfigurable logic

Main article: reconfigurable computing

Another track of development is to combine reconfigurable logic with a general-purpose CPU. In this scheme, a special computer language compiles fast-running subroutines into a bit-mask to configure the logic. Slower, or less-critical parts of the program can be run by sharing their time on the CPU. This process has the capability to create devices such as software radios, by using digital signal processing to perform functions usually performed by analog electronics.

4.6.5 Open source processors

As the lines between hardware and software increasingly blur due to progress in design methodology and availability of chips such as FPGAs and cheaper production processes, even open source hardware has begun to appear. Loosely knit communities like OpenCores have recently announced completely open CPU architectures such as the OpenRISC which can be readily implemented on FPGAs or in custom produced chips, by anyone, without paying license fees, and even established processor manufacturers like Sun Microsystems have released processor designs (e.g. OpenSPARC) under open-source licenses.

4.6.6 Asynchronous CPUs

Main article: Asynchronous CPU

Yet another possibility is the "clockless CPU" (asynchronous CPU). Unlike conventional processors, clockless processors have no central clock to coordinate the progress of data through the pipeline. Instead, stages of the CPU are coordinated using logic devices called "pipe line controls" or "FIFO sequencers." Basically, the pipeline controller clocks the next stage of logic when the existing stage is complete. In this way, a central clock is unnecessary.

It might be easier to implement high performance devices in asynchronous logic as opposed to clocked logic:

- components can run at different speeds in the clockless CPU. In a clocked CPU, no component can run faster than the clock rate.
- In a clocked CPU, the clock can go no faster than the worst-case performance of the slowest stage. In a clockless CPU, when a stage finishes faster than normal, the next stage can immediately take the results rather than waiting for the next clock tick. A stage might finish faster than normal because of the particular data inputs (multiplication can be very fast if it is multiplying by 0 or 1), or because it is running at a higher voltage or lower temperature than normal.

Asynchronous logic proponents believe these capabilities would have these benefits:

- lower power dissipation for a given performance level
- highest possible execution speeds

The biggest disadvantage of the clockless CPU is that most CPU design tools assume a clocked CPU (a synchronous circuit), so making a clockless CPU (designing an asynchronous circuit) involves modifying the design tools to handle clockless logic and doing extra testing to ensure the design avoids metastable problems.

Even so, several asynchronous CPUs have been built, including

- the ORDVAC and the identical ILLIAC I (1951)
- the ILLIAC II (1962), the fastest computer in the world at the time
- The Caltech Asynchronous Microprocessor, the world-first asynchronous microprocessor (1988)
- the ARM-implementing AMULET (1993 and 2000)
- the asynchronous implementation of MIPS R3000, dubbed MiniMIPS (1998)
- the SEAforth multi-core processor from Charles H. Moore [1]

4.6.7 Optical communication

One interesting possibility would be to eliminate the front side bus. Modern vertical laser diodes enable this change. In theory, an optical computer's components could directly connect through a holographic or phased open-air switching system. This would provide a large increase in effective speed and design flexibility, and a large reduction in cost. Since a computer's connectors are also its most likely failure point, a busless system might be more reliable, as well.

In addition, current (2010) modern processors use 64- or 128-bit logic. Wavelength superposition could allow for data lanes and logic many orders of magnitude higher, without additional space or copper wires.

4.6.8 Optical processors

Another long-term possibility is to use light instead of electricity for the digital logic itself. In theory, this could

run about 30% faster and use less power, as well as permit a direct interface with quantum computational devices. The chief problem with this approach is that for the foreseeable future, electronic devices are faster, smaller (i.e. cheaper) and more reliable. An important theoretical problem is that electronic computational elements are already smaller than some wavelengths of light, and therefore even wave-guide–based optical logic may be uneconomic compared to electronic logic. The majority of development effort, as of 2006 is focused on electronic circuitry. See also optical computing.

4.6.9 Belt Machine Architecture

Main article: Belt machine

As opposed to conventional register machine or stack machine architecture, yet similar to Intel's Itanium architecture,[2] a temporal register addressing scheme has been proposed by Ivan Godard & company that is intended to greatly reduce the complexity of CPU hardware (specifically the number of internal registers and the resulting huge multiplexer trees).[3] While somewhat harder to read and debug than general-purpose register names, it is recommended that it be perceived as a moving "conveyor belt" where the oldest values "drop off" the belt into oblivion. It is implemented by the Mill CPU architecture.

4.7 Timeline of events

- 1964. IBM releases the 32-bit IBM System/360 with memory protection.
- 1971. Intel released the 4-bit Intel 4004, the world's first commercially available microprocessor.
- 1975. MOS Technology released the 8-bit MOS Technology 6502, the first integrated processor to have an affordable price of $25 when the 6800 competition demanded $175.
- 1977. First 32-bit VAX sold, a VAX-11/780.
- 1978. Intel introduces the Intel 8086 and Intel 8088, the first x86 chips.
- 1981. Stanford MIPS introduced, one of the first RISC designs.
- 1982. Intel introduces the Intel 80286, which was the first Intel processor that could run all the software written for its predecessors, the 8086 and 8088.
- 1984, Motorola introduces the Motorola 68020+68851, which enabled 32-bit instruction set and virtualization.
- 1985. Intel introduces the Intel 80386, which adds a 32-bit instruction set to the x86 microarchitecture.
- 1989. Intel introduces the Intel 80486
- 1993. Intel launches the original Pentium microprocessor, the first processor with a x86 superscalar microarchitecture.
- 1995. Intel introduces the Pentium Pro which becomes the foundation for the Pentium II, Pentium III, Pentium M, and Intel Core Architectures.
- 2000. AMD announced x86-64 extension to the x86 microarchitecture.
- 2000. AMD hits 1 GHZ with its Athlon microprocessor.
- 2000. Analog Devices introduces the Blackfin architecture.
- 2002. Intel releases a Pentium 4 with Hyper-Threading, the first modern desktop processor to implement simultaneous multithreading (SMT).
- 2003. AMD releases the Athlon 64, the first 64-bit consumer cpu.
- 2003. Intel introduced the Pentium M, a low power mobile derivative of the Pentium Pro architecture.
- 2005. AMD announced the Athlon 64 X2, the first x86 dual-core processor.
- 2006. Intel introduces the Core line of CPUs based on a modified Pentium M design.
- 2008. About ten billion CPUs were manufactured in 2008.
- 2010. Intel introduced Core i3, i5, i7 processors.
- 2011. AMD announces the appearance of the world's first 8 core CPU for desktop PC's.

4.8 See also

- Microprocessor chronology

4.9 References

[1] SEAforth Overview "... asynchronous circuit design throughout the chip. There is no central clock with billions of dumb nodes dissipating useless power. ... the processor cores are internally asynchronous themselves."

[2] http://williams.comp.ncat.edu/comp375/RISCprocessors.pdf

[3] "The Belt".

4.10 External links

- Great moments in microprocessor history by W. Warner, 2004
- Great Microprocessors of the Past and Present (V 13.4.0) by: John Bayko, 2003

Chapter 5

Processor design

Processor design is the design engineering task of creating a microprocessor, a component of computer hardware. It is a subfield of electronics engineering and computer engineering. The design process involves choosing an instruction set and a certain execution paradigm (e.g. VLIW or RISC) and results in a microarchitecture described in e.g. VHDL or Verilog. This description is then manufactured employing some of the various semiconductor device fabrication processes. This results in a die which is bonded onto some chip carrier. This chip carrier is then soldered onto some printed circuit board (PCB).

The mode of operation of any microprocessor is the execution of lists of instructions. Instructions typically include those to compute or manipulate data values using registers, change or retrieve values in read/write memory, perform relational tests between data values and to control program flow.

5.1 Details

CPU design focuses on six main areas:

1. datapaths (such as ALUs and pipelines)
2. control unit: logic which controls the datapaths
3. Memory components such as register files, caches
4. Clock circuitry such as clock drivers, PLLs, clock distribution networks
5. Pad transceiver circuitry
6. Logic gate cell library which is used to implement the logic

CPUs designed for high-performance markets might require custom designs for each of these items to achieve frequency, power-dissipation, and chip-area goals whereas CPUs designed for lower performance markets might lessen the implementation burden by acquiring some of these items by purchasing them as intellectual property. Control logic implementation techniques (logic synthesis using CAD tools) can be used to implement datapaths, register files, and clocks. Common logic styles used in CPU design include unstructured random logic, finite-state machines, microprogramming (common from 1965 to 1985), and Programmable logic arrays (common in the 1980s, no longer common).

Device types used to implement the logic include:

- Transistor-transistor logic Small Scale Integration logic chips - no longer used for CPUs
- Programmable Array Logic and Programmable logic devices - no longer used for CPUs
- Emitter-coupled logic (ECL) gate arrays - no longer common
- CMOS gate arrays - no longer used for CPUs
- CMOS mass-produced ICs - the vast majority of CPUs by volume
- CMOS ASICs - only for a minority of special applications due to expense
- Field-programmable gate arrays (FPGA) - common for soft microprocessors, and more or less required for reconfigurable computing

A CPU design project generally has these major tasks:

- Programmer-visible instruction set architecture, which can be implemented by a variety of microarchitectures
- Architectural study and performance modeling in ANSI C/C++ or SystemC
- High-level synthesis (HLS) or register transfer level (RTL, e.g. logic) implementation
- RTL verification
- Circuit design of speed critical components (caches, registers, ALUs)
- Logic synthesis or logic-gate-level design

- Timing analysis to confirm that all logic and circuits will run at the specified operating frequency
- Physical design including floorplanning, place and route of logic gates
- Checking that RTL, gate-level, transistor-level and physical-level representations are equivalent
- Checks for signal integrity, chip manufacturability

Re-designing a CPU core to a smaller die-area helps to shrink everything (a "photomask shrink"), resulting in the same number of transistors on a smaller die. It improves performance (smaller transistors switch faster), reduces power (smaller wires have less parasitic capacitance) and reduces cost (more CPUs fit on the same wafer of silicon). Releasing a CPU on the same size die, but with a smaller CPU core, keeps the cost about the same but allows higher levels of integration within one very-large-scale integration chip (additional cache, multiple CPUs, or other components), improving performance and reducing overall system cost.

As with most complex electronic designs, the logic verification effort (proving that the design does not have bugs) now dominates the project schedule of a CPU.

Key CPU architectural innovations include index register, cache, virtual memory, instruction pipelining, superscalar, CISC, RISC, virtual machine, emulators, microprogram, and stack.

5.1.1 Micro-architectural concepts

Main article: Microarchitecture

5.1.2 Research topics

Main article: History of general-purpose CPUs § 1990 to today: looking forward

A variety of new CPU design ideas have been proposed, including reconfigurable logic, clockless CPUs, computational RAM, and optical computing.

5.1.3 Performance analysis and benchmarking

Main article: Computer performance

Benchmarking is a way of testing CPU speed. Examples include SPECint and SPECfp, developed by Standard Performance Evaluation Corporation, and ConsumerMark developed by the Embedded Microprocessor Benchmark Consortium EEMBC.

Some of the commonly used metrics include:

- Instructions per second - Most consumers pick a computer architecture (normally Intel IA32 architecture) to be able to run a large base of pre-existing pre-compiled software. Being relatively uninformed on computer benchmarks, some of them pick a particular CPU based on operating frequency (see Megahertz Myth).
- FLOPS - The number of floating point operations per second is often important in selecting computers for scientific computations.
- Performance per watt - System designers building parallel computers, such as Google, pick CPUs based on their speed per watt of power, because the cost of powering the CPU outweighs the cost of the CPU itself.[1][2]
- Some system designers building parallel computers pick CPUs based on the speed per dollar.
- System designers building real-time computing systems want to guarantee worst-case response. That is easier to do when the CPU has low interrupt latency and when it has deterministic response. (DSP)
- Computer programmers who program directly in assembly language want a CPU to support a full featured instruction set.
- Low power - For systems with limited power sources (e.g. solar, batteries, human power).
- Small size or low weight - for portable embedded systems, systems for spacecraft.
- Environmental impact - Minimizing environmental impact of computers during manufacturing and recycling as well during use. Reducing waste, reducing hazardous materials. (see Green computing).

There may be tradeoffs in optimizing some of these metrics. In particular, many design techniques that make a CPU run faster make the “performance per watt”, “performance per dollar”, and “deterministic response” much worse, and vice versa.

5.2 Markets

There are several different markets in which CPUs are used. Since each of these markets differ in their requirements for CPUs, the devices designed for one market are in most cases inappropriate for the other markets.

5.2.1 General purpose computing

The vast majority of revenues generated from CPU sales is for general purpose computing, that is, desktop, laptop, and server computers commonly used in businesses and homes. In this market, the Intel IA-32 architecture dominates, with its rivals PowerPC and SPARC maintaining much smaller customer bases. Yearly, hundreds of millions of IA-32 architecture CPUs are used by this market. A growing percentage of these processors are for mobile implementations such as netbooks and laptops.[3]

Since these devices are used to run countless different types of programs, these CPU designs are not specifically targeted at one type of application or one function. The demands of being able to run a wide range of programs efficiently has made these CPU designs among the more advanced technically, along with some disadvantages of being relatively costly, and having high power consumption.

High-end processor economics

In 1984, most high-performance CPUs required four to five years to develop.[4]

5.2.2 Scientific computing

Main article: Supercomputer

Scientific computing is a much smaller niche market (in revenue and units shipped). It is used in government research labs and universities. Before 1990, CPU design was often done for this market, but mass market CPUs organized into large clusters have proven to be more affordable. The main remaining area of active hardware design and research for scientific computing is for high-speed data transmission systems to connect mass market CPUs.

5.2.3 Embedded design

Main article: Embedded system

As measured by units shipped, most CPUs are embedded in other machinery, such as telephones, clocks, appliances, vehicles, and infrastructure. Embedded processors sell in the volume of many billions of units per year, however, mostly at much lower price points than that of the general purpose processors.

These single-function devices differ from the more familiar general-purpose CPUs in several ways:

- Low cost is of high importance.
- It is important to maintain a low power dissipation as embedded devices often have a limited battery life and it is often impractical to include cooling fans.[5]
- To give lower system cost, peripherals are integrated with the processor on the same silicon chip.
- Keeping peripherals on-chip also reduces power consumption as external GPIO ports typically require buffering so that they can source or sink the relatively high current loads that are required to maintain a strong signal outside of the chip.
    - Many embedded applications have a limited amount of physical space for circuitry; keeping peripherals on-chip will reduce the space required for the circuit board.
    - The program and data memories are often integrated on the same chip. When the only allowed program memory is ROM, the device is known as a microcontroller.
- For many embedded applications, interrupt latency will be more critical than in some general-purpose processors.

Embedded processor economics

The embedded CPU family with the largest number of total units shipped is the 8051, averaging nearly a billion units per year.[6] The 8051 is widely used because it is very inexpensive. The design time is now roughly zero, because it is widely available as commercial intellectual property. It is now often embedded as a small part of a larger system on a chip. The silicon cost of an 8051 is now as low as US$0.001, because some implementations use as few as 2,200 logic gates and take 0.0127 square millimeters of silicon.[7][8]

As of 2009, more CPUs are produced using the ARM architecture instruction set than any other 32-bit instruction set.[9][10] The ARM architecture and the first ARM chip were designed in about one and a half years and 5 human years of work time.[11]

The 32-bit Parallax Propeller microcontroller architecture and the first chip were designed by two people in about 10 human years of work time.[12]

The 8-bit AVR architecture and first AVR microcontroller was conceived and designed by two students at the Norwegian Institute of Technology.

The 8-bit 6502 architecture and the first MOS Technology 6502 chip were designed in 13 months by a group of about 9 people.[13]

Research and educational CPU design

The 32 bit Berkeley RISC I and RISC II architecture and the first chips were mostly designed by a series of students

as part of a four quarter sequence of graduate courses.[14] This design became the basis of the commercial SPARC processor design.

For about a decade, every student taking the 6.004 class at MIT was part of a team—each team had one semester to design and build a simple 8 bit CPU out of 7400 series integrated circuits. One team of 4 students designed and built a simple 32 bit CPU during that semester. [15]

Some undergraduate courses require a team of 2 to 5 students to design, implement, and test a simple CPU in a FPGA in a single 15 week semester. [16]

The MultiTitan CPU was designed with 2.5 man years of effort, which was considered "relatively little design effort" at the time.[17] 24 people contributed to the 3.5 year MultiTitan research project, which included designing and building a prototype CPU.[18]

Soft microprocessor cores

Main article: Soft microprocessor

For embedded systems, the highest performance levels are often not needed or desired due to the power consumption requirements. This allows for the use of processors which can be totally implemented by logic synthesis techniques. These synthesized processors can be implemented in a much shorter amount of time, giving quicker time-to-market.

5.3 See also

- Central processing unit
- Comparison of instruction set architectures
- History of general-purpose CPUs
- Microprocessor
- Microarchitecture
- Moore's law
- Amdahl's law
- System-on-a-chip
- Reduced instruction set computer
- Complex instruction set computer
- Minimal instruction set computer
- Electronic design automation
- High-level synthesis

5.4 References

[1] "EEMBC ConsumerMark". Archived from the original on March 27, 2005.

[2] Stephen Shankland (December 9, 2005). "Power could cost more than servers, Google warns".

[3] Kerr, Justin. "AMD Loses Market Share as Mobile CPU Sales Outsell Desktop for the First Time." Maximum PC. Published 2010-10-26.

[4] "New system manages hundreds of transactions per second" article by Robert Horst and Sandra Metz, of Tandem Computers Inc., "Electronics" magazine, 1984 April 19: "While most high-performance CPUs require four to five years to develop, The NonStop TXP processor took just 2+1/2 years -- six months to develop a complete written specification, one year to construct a working prototype, and another year to reach volume production."

[5] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems", IJCAET, 6(4), 440–459, 2014.

[6] http://people.wallawalla.edu/~{}curt.nelson/engr355/lecture/8051_overview.pdf

[7] Square millimeters per 8051, 0.013 in 45nm line-widths; see

[8] To figure dollars per square millimeter, see , and note that an SOC component has no pin or packaging costs.

[9] "ARM Cores Climb Into 3G Territory" by Mark Hachman, 2002.

[10] "The Two Percent Solution" by Jim Turley 2002.

[11] "ARM's way" 1998

[12] "Why the Propeller Works" by Chip Gracey

[13] "Interview with William Mensch"

[14] 'Design and Implementation of RISC I' - original journal article by C.E. Sequin and D.A.Patterson

[15] "the VHS"

[16] "Teaching Computer Design with FPGAs" by Jan Gray

[17] Norman P. Jouppi and Jeffrey Y. F. Tang. "A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance". 1989. doi:10.1.1.85.988. p. "i".

[18] "MultiTitan: Four Architecture Papers". 1988. p. 4-5.

- Hwang, Enoch (2006). *Digital Logic and Microprocessor Design with VHDL*. Thomson. ISBN 0-534-46593-5.
- Processor Design: An Introduction

Chapter 6

Very long instruction word

Very long instruction word (**VLIW**) refers to processor architectures designed to take advantage of instruction level parallelism (ILP). Whereas conventional processors mostly allow programs only to specify instructions that will be executed in sequence, a VLIW processor allows programs to explicitly specify instructions that will be executed at the same time (that is, in *parallel*). This type of processor architecture is intended to allow higher performance without the inherent complexity of some other approaches.

6.1 Overview

Traditional approaches to improving performance in processor architectures include breaking up instructions into sub-steps so that instructions can be executed partially at the same time (known as *pipelining*), dispatching individual instructions to be executed completely independently in different parts of the processor (*superscalar architectures*), and even executing instructions in an order different from the program (*out-of-order execution*). These approaches all involve increased hardware complexity (higher cost, larger circuits, higher power consumption) because the processor must intrinsically make all of the decisions internally for these approaches to work. The VLIW approach, by contrast, depends on the programs themselves providing all the decisions regarding which instructions are to be executed simultaneously and how conflicts are to be resolved. As a practical matter this means that the compiler (software used to create the final programs) becomes much more complex, but the hardware is simpler than many other approaches to parallelism.

The acronym VLIW may also refer to *variable-length instruction word*, a criteria in instruction set design to allow for a more flexible layout of the instruction set and higher code density (depending on the instructions to be used). For example, this approach makes it possible to load an immediate value of the size of a machine word into a processor register, which would not be feasible if each instruction was limited to the size of machine word. The flexibility comes at an additional cost for instruction decoding.[1][2]

6.2 Motivation

A processor that executes every instruction one after the other (i.e. a non-pipelined scalar architecture) may use processor resources inefficiently, potentially leading to poor performance. The performance can be improved by executing different sub-steps of sequential instructions simultaneously (this is *pipelining*), or even executing multiple instructions entirely simultaneously as in superscalar architectures. Further improvement can be achieved by executing instructions in an order different from the order they appear in the program; this is called out-of-order execution.

These three techniques all come at the cost of increased hardware complexity. Before executing any operations in parallel the processor must verify that the instructions do not have interdependencies. For example, if a first instruction's result is used as a second instruction's input then they cannot execute at the same time and the second instruction can't be executed before the first. Modern out-of-order processors have increased the hardware resources which do the scheduling of instructions and determining of interdependencies.

The VLIW approach, on the other hand, executes operations in parallel based on a fixed schedule determined when programs are compiled. Since determining the order of execution of operations (including which operations can execute simultaneously) is handled by the compiler, the processor does not need the scheduling hardware that the three techniques described above require. As a result, VLIW CPUs offer significant computational power with less hardware complexity (but greater compiler complexity) than is associated with most superscalar CPUs.

6.3 Design

In superscalar designs, the number of execution units is invisible to the instruction set. Each instruction encodes only one operation. For most superscalar designs, the instruction width is 32 bits or fewer.

In contrast, one VLIW instruction encodes multiple op-

erations; specifically, one instruction encodes at least one operation for each execution unit of the device. For example, if a VLIW device has five execution units, then a VLIW instruction for that device would have five operation fields, each field specifying what operation should be done on that corresponding execution unit. To accommodate these operation fields, VLIW instructions are usually at least 64 bits wide, and on some architectures are much wider.

For example, the following is an instruction for the SHARC. In one cycle, it does a floating-point multiply, a floating-point add, and two autoincrement loads. All of this fits into a single 48-bit instruction:

f12 = f0 * f4, f8 = f8 + f12, f0 = dm(i0, m3), f4 = pm(i8, m9);

Since the earliest days of computer architecture,[3] some CPUs have added several additional arithmetic logic units (ALUs) to run in parallel. Superscalar CPUs use hardware to decide which operations can run in parallel at runtime, while VLIW CPUs use software (the compiler) to decide which operations can run in parallel in advance. Because the complexity of instruction scheduling is pushed off onto the compiler, complexity of the hardware can be substantially reduced.

A similar problem occurs when the result of a parallelisable instruction is used as input for a branch. Most modern CPUs "guess" which branch will be taken even before the calculation is complete, so that they can load up the instructions for the branch, or (in some architectures) even start to compute them speculatively. If the CPU guesses wrong, all of these instructions and their context need to be "flushed" and the correct ones loaded, which is time-consuming.

This has led to increasingly complex instruction-dispatch logic that attempts to guess correctly, and the simplicity of the original RISC designs has been eroded. VLIW lacks this logic, and therefore lacks its power consumption, possible design defects and other negative features.

In a VLIW, the compiler uses heuristics or profile information to guess the direction of a branch. This allows it to move and preschedule operations speculatively before the branch is taken, favoring the most likely path it expects through the branch. If the branch goes the unexpected way, the compiler has already generated compensatory code to discard speculative results to preserve program semantics.

Vector processor (SIMD) cores can be combined with the VLIW architecture such as in the Fujitsu FR-V microprocessor, further increasing throughput and speed.

6.4 History

The term *VLIW*, and the concept of VLIW architecture itself, were invented by Josh Fisher in his research group at Yale University in the early 1980s.[4] His original development of trace scheduling as a compilation technique for VLIW was developed when he was a graduate student at New York University. Prior to VLIW, the notion of prescheduling execution units and instruction-level parallelism in software was well established in the practice of developing horizontal microcode.

Fisher's innovations were around developing a compiler that could target horizontal microcode from programs written in an ordinary programming language. He realized that to get good performance and target a wide-issue machine, it would be necessary to find parallelism beyond that generally within a basic block. He also developed region scheduling techniques to identify parallelism beyond basic blocks. Trace scheduling is such a technique, and involves scheduling the most likely path of basic blocks first, inserting compensation code to deal with speculative motions, scheduling the second most likely trace, and so on, until the schedule is complete.

Fisher's second innovation was the notion that the target CPU architecture should be designed to be a reasonable target for a compiler — that the compiler and the architecture for a VLIW processor must be co-designed. This was partly inspired by the difficulty Fisher observed at Yale of compiling for architectures like Floating Point Systems' FPS164, which had a complex instruction set architecture (CISC) that separated instruction initiation from the instructions that saved the result, requiring very complicated scheduling algorithms. Fisher developed a set of principles characterizing a proper VLIW design, such as self-draining pipelines, wide multi-port register files, and memory architectures. These principles made it easier for compilers to write fast code.

The first VLIW compiler was described in a Ph.D. thesis by John Ellis, supervised by Fisher. The compiler was christened Bulldog, after Yale's mascot.[5] John Ruttenberg also developed certain important algorithms for scheduling.

Fisher left Yale in 1984 to found a startup company, Multiflow, along with co-founders John O'Donnell and John Ruttenberg. Multiflow produced the TRACE series of VLIW minisupercomputers, shipping their first machines in 1987. Multiflow's VLIW could issue 28 operations in parallel per instruction. The TRACE system was implemented in an MSI/LSI/VLSI mix packaged in cabinets, a technology that fell out of favor when it became more cost-effective to integrate all of the components of a processor (excluding memory) on a single chip.

Multiflow was too early to catch the following wave, when chip architectures began to allow multiple-issue CPUs . The major semiconductor companies recognized the value of Multiflow technology in this context, so the compiler and architecture were subsequently licensed to most of these companies.

6.5 Implementations

Cydrome was a company producing VLIW numeric processors using ECL technology in the same timeframe (late 1980s). This company, like Multiflow, went out of business after a few years.

One of the licensees of the Multiflow technology is Hewlett-Packard, which Josh Fisher joined after Multiflow's demise. Bob Rau, founder of Cydrome, also joined HP after Cydrome failed. These two would lead computer architecture research within Hewlett-Packard during the 1990s.

In addition to the above systems, at around the same period (i.e. 1989-1990), Intel implemented VLIW in the Intel i860, their first 64bit microprocessor; the i860 was also the first processor to implement VLIW on a single chip.[6] This processor could operate in both simple RISC mode and VLIW mode:

> In the early 1990s, Intel introduced the i860 RISC microprocessor. This simple chip had two modes of operation: a scalar mode and a VLIW mode. In the VLIW mode, the processor always fetched two instructions and assumed that one was an integer instruction and the other floating-point.[6]

The i860's VLIW mode was used extensively in embedded DSP applications since the application execution and datasets were simple, well ordered and predictable, allowing the designer to take full advantage of the parallel execution advantages that VLIW lent itself to; in VLIW mode the i860 was able to maintain floating-point performance in the range of 20-40 double-precision MFLOPS (an extremely high figure for its time and for a processor operating at 25-50Mhz).

In the 1990s, Hewlett-Packard researched this problem as a side effect of ongoing work on their PA-RISC processor family. They found that the CPU could be greatly simplified by removing the complex dispatch logic from the CPU and placing it into the compiler. Compilers of the day were much more complex than those from the 1980s, so the added complexity in the compiler was considered to be a small cost.

VLIW CPUs are usually constructed of multiple RISC-like execution units that operate independently. Contemporary VLIWs typically have four to eight main execution units. Compilers generate initial instruction sequences for the VLIW CPU in roughly the same manner that they do for traditional CPUs, generating a sequence of RISC-like instructions. The compiler analyzes this code for dependence relationships and resource requirements. It then schedules the instructions according to those constraints. In this process, independent instructions can be scheduled in parallel. Because VLIWs typically represent instructions scheduled in parallel with a longer instruction word that incorporates the individual instructions, this results in a much longer opcode (thus the term "very long") to specify what executes on a given cycle.

Examples of contemporary VLIW CPUs include the TriMedia media processors by NXP (formerly Philips Semiconductors), the SHARC DSP by Analog Devices, the C6000 DSP family by Texas Instruments, the STMicroelectronics ST200 family based on the Lx architecture (designed in Josh Fisher's HP lab by Paolo Faraboschi), and the MPPA MANYCORE family by KALRAY. These contemporary VLIW CPUs are primarily successful as embedded media processors for consumer electronic devices.

VLIW features have also been added to configurable processor cores for SoC designs. For example, Tensilica's Xtensa LX2 processor incorporates a technology dubbed FLIX (Flexible Length Instruction eXtensions) that allows multi-operation instructions. The Xtensa C/C++ compiler can freely intermix 32- or 64-bit FLIX instructions with the Xtensa processor's single-operation RISC instructions, which are 16 or 24 bits wide. By packing multiple operations into a wide 32- or 64-bit instruction word and allowing these multi-operation instructions to be intermixed with shorter RISC instructions, FLIX technology allows SoC designers to realize VLIW's performance advantages while eliminating the code bloat of early VLIW architectures. The Infineon Carmel DSP is another VLIW processor core intended for SoC; it uses a similar code density improvement technique called "configurable long instruction word" (CLIW). [7]

Outside embedded processing markets, Intel's Itanium IA-64 EPIC and Elbrus 2000 appear as the only examples of a widely used VLIW CPU architectures. However, EPIC architecture is sometimes distinguished from a pure VLIW architecture, since EPIC advocates full instruction predication, rotating register files, and a very long instruction word that can encode non-parallel instruction groups. VLIWs also gained significant consumer penetration in the GPU market, though both Nvidia and AMD have since moved to RISC architectures in order to improve performance on non-graphics workloads.

ATI's/AMD's TeraScale microarchitecture for GPUs is a VLIW microarchitecture.

In December 2015 the first shipment of PCs based on VLIW CPU Elbrus-4s was made in Russia [8]

6.6 Backward compatibility

When silicon technology allowed for wider implementations (with more execution units) to be built, the compiled programs for the earlier generation would not run on the wider implementations, as the encoding of the binary instructions depended on the number of execution units of the machine.

Transmeta addresses this issue by including a binary-to-binary software compiler layer (termed *code morphing*) in their Crusoe implementation of the x86 architecture. Basically, this mechanism is advertised to recompile, optimize, and translate x86 opcodes at runtime into the CPU's internal machine code. Thus, the Transmeta chip is *internally* a VLIW processor, effectively decoupled from the x86 CISC instruction set that it executes.

Intel's Itanium architecture (among others) solved the backward-compatibility problem with a more general mechanism. Within each of the multiple-opcode instructions, a bit field is allocated to denote dependency on the previous VLIW instruction within the program instruction stream. These bits are set at compile time, thus relieving the hardware from calculating this dependency information. Having this dependency information encoded into the instruction stream allows wider implementations to issue multiple non-dependent VLIW instructions in parallel per cycle, while narrower implementations would issue a smaller number of VLIW instructions per cycle.

Another perceived deficiency of VLIW architectures is the code bloat that occurs when not all of the execution units have useful work to do and thus have to execute NOPs. This occurs when there are dependencies in the code and the instruction pipelines must be allowed to drain before subsequent operations can proceed.

Since the number of transistors on a chip has grown, the perceived disadvantages of the VLIW have diminished in importance. The VLIW architecture is growing in popularity, particularly in the embedded market, where it is possible to customize a processor for an application in an embedded system-on-a-chip. Embedded VLIW products are available from several vendors, including the FR-V from Fujitsu, the BSP15/16 from Pixelworks, the ST231 from STMicroelectronics, the TriMedia from NXP, the CEVA-X DSP from CEVA, the Jazz DSP from Improv Systems, and Silicon Hive. The Texas Instruments TMS320 DSP line has evolved, in its C6xxx family, to look more like a VLIW, in contrast to the earlier C5xxx family.

6.7 See also

- Explicitly parallel instruction computing (EPIC)
- Transport triggered architecture (TTA)
- Elbrus processors
- Mill CPU Architecture

6.8 References

[1] Wai-Kai Chen (2000). *Memory, Microprocessor, and ASIC*. *books.google.com* (CRC Press). pp. 11–14, 11–15. Retrieved 2014-08-19.

[2] Heidi Pan; Krste Asanovic (2001). "Heads and Tails: A Variable-Length Instruction Format Supporting Parallel Fetch and Decode" (PDF). *scale.eecs.berkeley.edu*. Retrieved 2014-08-19.

[3] "CONTROL DATA 6400/6500/6600 COMPUTER SYSTEMS Reference Manual". 1969-02-21. Retrieved 7 November 2013.

[4] Fisher, Joseph A. (1983). "Very Long Instruction Word architectures and the ELI-512" (PDF). *Proceedings of the 10th annual international symposium on Computer architecture*. International Symposium on Computer Architecture. New York, NY, USA: ACM. pp. 140–150. doi:10.1145/800046.801649. ISBN 0-89791-101-6. Retrieved 2009-04-27.

[5] "ACM 1985 Doctoral Dissertation Award". ACM. Retrieved 2007-10-15. For his dissertation *Bulldog: A Compiler for VLIW Architecture*.

[6] "An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture" (PDF). Philips Semiconductors. Archived from the original (PDF) on 2011-09-29.

[7] "EEMBC Publishes Benchmark Scores for Infineon Technologies' Carmel DSP Core and TriCore TC11IB Microcontroller"

[8] http://tass.ru/ekonomika/2498729

6.9 External links

- Paper That Introduced VLIWs
- Book on the history of Multiflow Computer, VLIW pioneering company
- ISCA "Best Papers" Retrospective On Paper That Introduced VLIWs
- VLIW and Embedded Processing
- FR500 VLIW-architecture High-performance Embedded Microprocessor
- Historical background for EPIC instruction set architectures

Chapter 7

Dataflow architecture

Dataflow architecture is a computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. Dataflow architectures do not have a program counter, or (at least conceptually) the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable: i. e. behavior is indeterministic.

Although no commercially successful general-purpose computer hardware has used a dataflow architecture, it has been successfully implemented in specialized hardware such as in digital signal processing, network routing, graphics processing, telemetry, and more recently in data warehousing. It is also very relevant in many software architectures today including database engine designs and parallel computing frameworks.

Synchronous dataflow architectures tune to match the workload presented by real-time data path applications such as wire speed packet forwarding. Dataflow architectures that are deterministic in nature enable programmers to manage complex tasks such as processor load balancing, synchronization and accesses to common resources.[1]

Meanwhile, there is a clash of terminology, since the term *dataflow* is used for a subarea of parallel programming: for dataflow programming.

7.1 History

Hardware architectures for dataflow was a major topic in computer architecture research in the 1970s and early 1980s. Jack Dennis of MIT pioneered the field of static dataflow architectures while the Manchester Dataflow Machine[2] and MIT Tagged Token architecture were major projects in dynamic dataflow.

The research, however, never overcame the problems related to:

- Efficiently broadcasting data tokens in a massively parallel system.
- Efficiently dispatching instruction tokens in a massively parallel system.
- Building CAMs large enough to hold all of the dependencies of a real program.

Instructions and their data dependencies proved to be too fine-grained to be effectively distributed in a large network. That is, the time for the instructions and tagged results to travel through a large connection network was longer than the time to actually do the computations.

Nonetheless, Out-of-order execution (OOE) has become the dominant computing paradigm since the 1990s. It is a form of restricted dataflow. This paradigm introduced the idea of an *execution window*. The *execution window* follows the sequential order of the von Neumann architecture, however within the window, instructions are allowed to be completed in data dependency order. This is accomplished in CPUs that dynamically tag the data dependencies of the code in the execution window. The logical complexity of dynamically keeping track of the data dependencies, restricts *OOE* CPUs to a small number of execution units (2-6) and limits the execution window sizes to the range of 32 to 200 instructions, much smaller than envisioned for full dataflow machines.

7.2 Dataflow architecture topics

7.2.1 Static and dynamic dataflow machines

Designs that use conventional memory addresses as data dependency tags are called static dataflow machines. These machines did not allow multiple instances of the same routines to be executed simultaneously because the simple tags could not differentiate between them.

Designs that use content-addressable memory (CAM) are called dynamic dataflow machines. They use tags in memory to facilitate parallelism.

7.2.2 Compiler

Normally, in the control flow architecture, compilers analyze program source code for data dependencies between

instructions in order to better organize the instruction sequences in the binary output files. The instructions are organized sequentially but the dependency information itself is not recorded in the binaries. Binaries compiled for a dataflow machine contain this dependency information.

A dataflow compiler records these dependencies by creating unique tags for each dependency instead of using variable names. By giving each dependency a unique tag, it allows the non-dependent code segments in the binary to be executed *out of order* and in parallel.

7.2.3 Programs

Programs are loaded into the CAM of a dynamic dataflow computer. When all of the tagged operands of an instruction become available (that is, output from previous instructions and/or user input), the instruction is marked as ready for execution by an execution unit.

This is known as *activating* or *firing* the instruction. Once an instruction is completed by an execution unit, its output data is send (with its tag) in the CAM. Any instructions that are dependent upon this particular datum (identified by its tag value) are then marked as ready for execution. In this way, subsequent instructions are executed in proper order, avoiding race conditions. This order may differ from the sequential order envisioned by the human programmer, the programmed order.

7.2.4 Instructions

An instruction, along with its required data operands, is transmitted to an execution unit as a packet, also called an *instruction token*. Similarly, output data is transmitted back to the CAM as a *data token*. The packetization of instructions and results allows for parallel execution of ready instructions on a large scale.

Dataflow networks deliver the instruction tokens to the execution units and return the data tokens to the CAM. In contrast to the conventional von Neumann architecture, data tokens are not permanently stored in memory, rather they are transient messages that only exist when in transit to the instruction storage.

7.3 See also

- Dataflow
- Parallel Computing
- SISAL
- Systolic array
- Transport triggered architecture

7.4 References

[1] "HX300 Family of NPUs and Programmable Ethernet Switches to the Fiber Access Market", *EN-Genius*, June 18 2008.

[2] Manchester Dataflow Research Project, Research Reports: Abstracts, September 1997

Chapter 8

Systolic array

In parallel computer architectures, a **systolic array** is a homogeneous network of tightly coupled Data Processing Units (DPUs) called cells or nodes. Each node or DPU independently computes a partial result as a function of the data received from its upstream neighbors, stores the result within itself and passes it downstream. Systolic arrays were invented by Richard P. Brent and H.T. Kung, who developed them to compute Greatest common divisors of integers and polynomials. [1] They are sometimes classified as Multiple Instruction Single Data (MISD) architectures under Flynn's Taxonomy, but this classification is questionable because a strong argument can be made to distinguish systolic arrays from any of Flynn's four categories: SISD, SIMD, MISD, MIMD, as discussed later in this article.

The parallel input data flows through a network of hardwired processor nodes, resembling the human brain which combine, process, merge or sort the input data into a derived result. Because the wave-like propagation of data through a systolic array resembles the pulse of the human circulatory system, the name systolic was coined from medical terminology. The name is derived from Systole (medicine) as an analogy to the regular pumping of blood by the heart.

8.1 Applications

Systolic arrays are often hard-wired for specific operations, such as "multiply and accumulate", to perform massively parallel integration, convolution, correlation, matrix multiplication or data sorting tasks.

8.2 Architecture

A systolic array typically consists of a large monolithic network of primitive computing nodes which can be hardwired or software configured for a specific application. The nodes are usually fixed and identical, while the interconnect is programmable. The more general **wavefront** processors, by contrast, employ sophisticated and individually programmable nodes which may or may not be monolithic, depending on the array size and design parameters. The other distinction is that systolic arrays rely on synchronous data transfers, while wavefront tend to work asynchronously.

Unlike the more common Von Neumann architecture, where program execution follows a script of instructions stored in common memory, addressed and sequenced under the control of the CPU's program counter (PC), the individual nodes within a systolic array are triggered by the arrival of new data and always process the data in exactly the same way. The actual processing within each node may be hard wired or block microcoded, in which case the common node personality can be block programmable.

The systolic array paradigm with data-streams driven by data counters, is the counterpart of the Von Neumann architecture with instruction-stream driven by a program counter. Because a systolic array usually sends and receives multiple data streams, and multiple data counters are needed to generate these data streams, it supports data parallelism.

The actual nodes can be simple and hardwired or consist of more sophisticated units using micro code, which may be block programmable.

8.3 Goals and benefits

A major benefit of systolic arrays is that all operand data and partial results are stored within (passing through) the processor array. There is no need to access external buses, main memory or internal caches during each operation as is the case with Von Neumann or Harvard sequential machines. The sequential limits on parallel performance dictated by Amdahl's Law also do not apply in the same way, because data dependencies are implicitly handled by the programmable node interconnect and there are no sequential steps in managing the highly parallel data flow.

Systolic arrays are therefore extremely good at artificial intelligence, image processing, pattern recognition, computer vision and other tasks which animal brains do so particularly well. Wavefront processors in general can

also be very good at machine learning by implementing self configuring neural nets in hardware.

8.4 Classification controversy

While systolic arrays are officially classified as MISD, their classification is somewhat problematic. Because the input is typically a vector of independent values, the systolic array is definitely not SISD. Since these input values are merged and combined into the result(s) and do not maintain their independence as they would in a SIMD vector processing unit, the array cannot be classified as such. Consequently, the array cannot be classified as a MIMD either, because MIMD can be viewed as a mere collection of smaller SISD and SIMD machines.

Finally, because the data swarm is transformed as it passes through the array from node to node, the multiple nodes are not operating on the same data, which makes the MISD classification a misnomer. The other reason why a systolic array should not qualify as a **MISD** is the same as the one which disqualifies it from the SISD category: The input data is typically a vector not a **s**ingle **d**ata value, although one could argue that any given input vector is a single data set.

In spite of all of the above, systolic arrays are often offered as a classic example of MISD architecture in textbooks on parallel computing and in the engineering class. If the array is viewed from the outside as atomic it should perhaps be classified as **SFMuDMeR** = Single Function, Multiple Data, Merged Result(s).

8.5 Detailed description

A systolic array is composed of matrix-like rows of data processing units called cells. Data processing units (DPUs) are similar to central processing units (CPUs), (except for the usual lack of a program counter,[2] since operation is transport-triggered, i.e., by the arrival of a data object). Each cell shares the information with its neighbors immediately after processing. The systolic array is often rectangular where data flows across the array between neighbour DPUs, often with different data flowing in different directions. The data streams entering and leaving the ports of the array are generated by auto-sequencing memory units, ASMs. Each ASM includes a data counter. In embedded systems a data stream may also be input from and/or output to an external source.

An example of a systolic algorithm might be designed for matrix multiplication. One matrix is fed in a row at a time from the top of the array and is passed down the array, the other matrix is fed in a column at a time from the left hand side of the array and passes from left to right. Dummy values are then passed in until each processor has seen one whole row and one whole column. At this point, the result of the multiplication is stored in the array and can now be output a row or a column at a time, flowing down or across the array.[3]

Systolic arrays are arrays of DPUs which are connected to a small number of nearest neighbour DPUs in a mesh-like topology. DPUs perform a sequence of operations on data that flows between them. Because the traditional systolic array synthesis methods have been practiced by algebraic algorithms, only uniform arrays with only linear pipes can be obtained, so that the architectures are the same in all DPUs. The consequence is, that only applications with regular data dependencies can be implemented on classical systolic arrays. Like SIMD machines, clocked systolic arrays compute in "lock-step" with each processor undertaking alternate compute | communicate phases. But systolic arrays with asynchronous handshake between DPUs are called *wavefront arrays*. One well-known systolic array is Carnegie Mellon University's iWarp processor, which has been manufactured by Intel. An iWarp system has a linear array processor connected by data buses going in both directions.

8.6 History

Systolic arrays (< **wavefront** processors), were first described by H. T. Kung and Charles E. Leiserson, who published the first paper describing systolic arrays in 1978. However, the first machine known to have used a similar technique was the Colossus Mark II in 1944.

8.7 Application example

An application Example - Polynomial Evaluation

Horner's rule for evaluating a polynomial is:

$y = (...(((a_n * x + a_{n-1}) * x + a_{n-2}) * x + a_{n-3}) * x + ... + a_1) * x + a_0$

A linear systolic array in which the processors are arranged in pairs: one multiplies its input by x and passes the result to the right, the next adds a_j and passes the result to the right:

8.8 Advantages and disadvantages

Pros

- Faster
- Scalable

Cons

- Expensive

- Highly specialized, custom hardware is required often application specific.
- Not widely implemented
- Limited code base of programs and algorithms.

8.9 Implementations

Cisco PXF network processor is internally organized as systolic array.[4]

8.10 See also

- MISD - Multiple Instruction Single Data, Example: Systolic Arrays
- iWarp - Systolic Array Computer, VLSI, Intel/CMU
- WARP (systolic array) - Systolic Array Computer, GE/CMU

8.11 Notes

[1] http://www.eecs.harvard.edu/~{}htk/publication/1984-ieeetoc-brent-kung.pdf

[2] The Paracel GeneMatcher series of systolic array processors do have a program counter. More complicated algorithms are implemented as a series of simple steps, with shifts specified in the instructions.

[3] Systolic Array Matrix Multiplication

[4] http://www.cisco.com/en/US/prod/collateral/routers/ps133/prod_white_paper09186a008008902a.html

8.12 References

- H. T. Kung, C. E. Leiserson: Algorithms for VLSI processor arrays; in: C. Mead, L. Conway (eds.): Introduction to VLSI Systems; Addison-Wesley, 1979
- S. Y. Kung: VLSI Array Processors; Prentice-Hall, Inc., 1988
- N. Petkov: Systolic Parallel Processing; North Holland Publishing Co, 1992

8.13 External links

- *Instruction Systolic Array (ISA)*
- 'A VLSI Architecture for Image Registration in Real Time' (Based on systolic array), Vol. 15, September 2007

8.14 Text and image sources, contributors, and licenses

8.14.1 Text

- **Computer architecture** *Source:* https://en.wikipedia.org/wiki/Computer_architecture?oldid=718566729 *Contributors:* Robert Merkel, Rjstott, Youssefsan, Toby Bartels, William Avery, Mudlock, Ray Van De Walker, SimonP, Hannes Hirzel, Stevertigo, Edward, RTC, Michael Hardy, Mahjongg, Ixfd64, Dori, CesarB, Mdebets, Ahoerstemeier, Ideyal, Cameronc, Raul654, Robbot, Murray Langton, Jmabel, JesseW, Iain.mcclatchie, Fabiform, Giftlite, Brouhaha, DavidCary, Harp, Lee J Haywood, VampWillow, Neilc, ConradPino, Togo~enwiki, Rich Farmbrough, Guanabot, Pj.de.bruin, Dyl, ESkog, ZeroOne, Neko-chan, LeonardoGregianin, MPerel, Quaternion, Mdd, Honeycake, Alansohn, Liao, Atlant, Pion, Hu, Bart133, Andrewmu, Wtmitchell, Velella, Brock, Cburnett, Bsadowski1, Oleg Alexandrov, Justinlebar, Uncle G, Ruud Koot, JeremyA, Wikiklrsc, Dionyziz, Eyreland, Graham87, Kbdank71, Reisio, Quiddity, ABot, The wub, FlaBot, Gnikhil, Margosbot~enwiki, BMF81, YurikBot, Salsia, Gaius Cornelius, Stassats, Danny31415, Nick, Matthiku, Aaron Schulz, DeadEyeArrow, Tetracube, Neomagus00, LeonardoRob0t, Whaa?, GrinBot~enwiki, Dkasak, SpLoT, SmackBot, Kellen, Incnis Mrsi, Prodego, Gilliam, Hmains, Jcarroll, Chris the speller, Kurykh, TimBentley, Thumperward, EncMstr, Newbyman, Nbarth, DHN-bot~enwiki, Dfletter, Can't sleep, clown will eat me, RyanEberhart, David Morón, Frap, Q uant, AcidPenguin9873, JonHarder, Zvar, Edivorce, Allan McInnes, SundarBot, Zachbenman, Krashlandon, Feradz, NongBot~enwiki, Robert Bond, InedibleHulk, Kvng, Shoeofdeath, Igoldste, Tawkerbot2, SkyWalker, CRGreathouse, Ahy1, Unixguy, Tuvas, Rdv, HenkeB, Michael B. Trausch, Xaariz, Tawkerbot4, Akhilesh043658647, Thijs!bot, Kubanczyk, Renaissongsman, Marek69, Ideogram, Liquid-aim-bot, Prolog, Dylan Lake, Skarkkai, Res2216firestar, JAnDbot, Gopal1035, The Transhumanist, Cmgomes, Bongwarrior, VoABot II, Nickmalik, CommonsDelinker, J.delanoy, Trusilver, Daufman, McSly, Plasticup, L.W.C. Nirosh, Su-steveri, VolkovBot, Su-steve, Jigabooda, MagicBanana, TedColes, Miko3k, AlleborgoBot, Biasoli, Jimmi Hugh, Logan, Hazel77, TheStarman, ThorstenStaerk~enwiki, Fanatix, Chickendude1313, Meldor, Gerakibot, Mark w69, Jerryobject, Masgatotkaca, Allmightyduck, Cépey, Vanished user kijsdion3i4jf, Kumioko, Svick, Denisarona, TheWILSE, Conniejlewis, ClueBot, Rilak, Czarkoff, Excirial, Ykhwong, NuclearWarfare, Dmyersturnbull, Razorflame, Rrccflores, Gereon K., Xxray03, Dsimic, Addbot, Melab-1, AkhtaBot, Leszek Jańczuk, Fluffernutter, Glane23, AnnaFrance, Favonian, Upulcranga, Tide rolls, Lightbot, Luckas-bot, Yobot, OrgasGirl, Fraggle81, Nanju123, Amirobot, Pcap, Mmxx, AnomieBOT, Li3939108, Fahadsadah, Materialscientist, Citation bot, Tharindunisal, Groovenstein, Joehms22, Shadowjams, TheAmplidude, Erik9bot, FrescoBot, Aubencheulobois, Alxeedo, HJ Mitchell, DrilBot, Pinethicket, Rameshngbot, Strenshon, Qazwsxedcrfv1, Merlion444, FoxBot, GlikD, SchreyP, ⁇⁇, Vrenator, Quafios, DARTH SIDIOUS 2, Anurag golipkar, RjwilmsiBot, EmausBot, Nuujinn, Racerx11, Primefac, Sohaib.mohd, Slawekb, Cogiati, Fæ, Alpha Quadrant, Elektrik Shoos, A930913, Microprocessor Man, L Kensington, Donner60, Tot12, ClamDip, 28bot, ClueBot NG, CocuBot, Phonedigs, Satellizer, Nickspoon0, Vacation9, Delusion23, Zynwyx, Robin400, Widr, Jgowdy, Helpful Pixie Bot, HMSSolent, Aalomaim, Wbm1058, AvocatoBot, Neutral current, Benzband, Dentalplanlisa, XIN3N, Orderkim, BattyBot, L8starter, Pratyya Ghosh, ChrisGualtieri, Codename Lisa, ZaferXYZ, Phamnhatkhanh, Mahbubur-r-aaman, Faizan, Forgot to put name, Greengreengreenred, Wenzchen, Pokechu22, VirtualAssist, ScotXW, G S Palmer, Olenyash, Abc 123 def 456, Trax support, Lich counter, Haosjaboeces, Esicam, Kylemanel, Bobby1234abcd, KasparBot, Compassionate727, Boehm, SandmanKing42, Jpskycak, Srinivas blaze and Anonymous: 335
- **Complex instruction set computing** *Source:* https://en.wikipedia.org/wiki/Complex_instruction_set_computing?oldid=705073844 *Contributors:* Graham Chapman, Mudlock, SimonP, Edward, Kwertii, Collabi, Furrykef, Carbuncle, Robbot, Murray Langton, Jason Quinn, Nayuki, VampWillow, Neilc, Thincat, Karl-Henner, Rdnk, Urhixidur, DmitryKo, Chris Howard, Slady, Rich Farmbrough, Galain, Swiftly, Dyl, Andrejj, CanisRufus, Joanjoc~enwiki, Pgquiles, WhiteTimberwolf, R. S. Shaw, Ejrrjs, James Foster, Jumbuck, Liao, Guy Harris, Stephan Leeds, Kelly Martin, MattGiuca, MFH, Eyreland, Eras-mus, Palica, Kbdank71, Virtualphtn, Bubba73, FlaBot, Quuxplusone, Chobot, Lion10, DVdm, Jpfagerback, RobotE, Arado, Gardar Rurak, Wiki alf, Ergbert, PS2pcGAMER, Bota47, Whaa?, Buybooks Marius, Rwwww, SmackBot, Prodego, Unyoyega, Eskimbot, DMTagatac, Chris the speller, Jprg1966, EncMstr, Sct72, Frap, JonHarder, Cybercobra, Blazar, Vina-iwbot~enwiki, Flying Bishop, Optakeover, RekishiEJ, Wws, Cooljeanius, Nikto parcheesy, HenkeB, Davnor, Saaya, Sopoforic, Xaariz, Skittleys, Thijs!bot, Epbr123, Fejesjoco, Alimentarywatson, JAnDbot, Deflective, Arifsaha, NapoliRoma, Destynova, R'n'B, JonathonReinhart, DorganBot, UnicornTapestry, VolkovBot, EvanCarroll, Nxavar, Jackfork, Mike4ty4, PokeYourHeadOff, Ethancleary, Bentogoa, Flyer22 Reborn, OKBot, Tesi1700, EoGuy, Rilak, Cassie Puma, Dthomsen8, Non-dropframe, TutterMouse, ChenzwBot, Legobot, Luckas-bot, Yobot, OrgasGirl, Amirobot, Nyat, AnomieBOT, Materialscientist, Xqbot, RibotBOT, DaleDe, Cdleary, Qbeep, Arndbergmann, Kallikanzarid, EmausBot, ZéroBot, Thine Antique Pen, SimDoc, Petrb, ClueBot NG, Tirppa~enwiki, Helpful Pixie Bot, DBigXray, BattyBot, Lemnaminor, Nehasharma28, Comp.arch, Trixie05, AmirrezaN, Sofia Koutsouveli, Ilias.fotopoulos, Shiv51292, Salmanmalik418 and Anonymous: 138
- **Reduced instruction set computing** *Source:* https://en.wikipedia.org/wiki/Reduced_instruction_set_computing?oldid=719038464 *Contributors:* Damian Yerrick, Derek Ross, WojPob, Eloquence, Mav, Uriyan, Bryan Derksen, Koyaanis Qatsi, Drj, Andre Engels, Josh Grosse, Nate Silva, Ray Van De Walker, Maury Markowitz, Fonzy, Hephaestos, Mrwojo, Edward, Michael Hardy, Kwertii, Modster, Pnm, Liftarn, Lquilter, Stan Shebs, Snoyes, Pratyeka, Jiang, Jengod, Charles Matthews, Adam Bishop, Dcoetzee, Dmsar, Wik, Mrand, Furrykef, David Shay, Wernher, Xyb, Finlay McWalter, RadicalBender, Jni, Phil Boswell, Murray Langton, Fredrik, Kristof vt, RedWolf, Donreed, Nurg, Romanm, Phil webster, Stewartadcock, Hadal, Wikibot, Iain.mcclatchie, MikeCapone, Tea2min, David Gerard, Ancheta Wis, Giftlite, DavidCary, Mintleaf~enwiki, Levin, Mark Richards, Nayuki, Solipsist, VampWillow, Bobblewik, Neilc, Knutux, Cliffster1, G3pro, Wehe, AndrewTheLott, Moxfyre, Kate, Corti, Imroy, Rich Farmbrough, Guanabot, Pixel8, Dyl, Bender235, ZeroOne, Evice, Kaszeta, Kwamikagami, Parklandspanaway, Susvolans, Sietse Snel, Pgquiles, Thunderbrand, Smalljim, Cmdrjameson, R. S. Shaw, Koper, Maurreen, Speedy-Gonsales, Trevj, Ianw, Zachlipton, Liao, Guy Harris, Nasukaren, Thaddeusw, Larowebr, Stephan Leeds, SimonW, A D Monroe III, Kelly Martin, Simetrical, Thorpe, MattGiuca, Robert K S, ^demon, Ruud Koot, Dkanter, Scootey, MFH, GregorB, Eyreland, Eras-mus, OCNative, Alecv, Toussaint, Weevil, Palica, Bcaff05, Graham87, BD2412, Qwertyus, Kbdank71, Yurik, Josh Parris, JVz, Patrick Gill, Ligulem, Bubba73, MarnetteD, Watcharakorn, Jamesmusik, StuartBrady, Lorkki, Toresbe, RAMChYLD, Quuxplusone, Chobot, DVdm, YurikBot, Retodon8, Arado, SpuriousQ, Gaius Cornelius, TheMandarin, Rat144, DragonHawk, JulesH, Mikeblas, Beanyk, Dbfirs, DeadEyeArrow, Bota47, Jasongagich, Vishwastengse, 6a4fe8aa039615ebd9ddb83d6acf9a1dc1b684f7, Dspradau, JoanneB, Marcosw, Kevin, Gesslein, Paul D. Anderson, NeilN, Rwwww, GrinBot~enwiki, Drcwright, SmackBot, Henriok, Unyoyega, Clpo13, Hardyplants, Gjs238, Btwied, Brianski, Betacommand, TimBentley, QTCaptain, Thumperward, PrimeHunter, EdgeOfEpsilon, The Appleton, Worthawholebean, Dro Kulix, Frap, Christan80, JonHarder, Cybercobra, TheiNhibition, Rajrajmarley, Nutschig, Mattpat, Davipo, SashatoBot, Anss123, Johncatsoulis, Jay.slovak, Littleman TAMU, Shirifan, 16@r, Dicklyon, Optakeover, Flibble, MTSbot~enwiki, IvanLanin, UncleDouggie, Paul Foxworthy, DavidConner, Raysonho, Wcooley, Xose.vazquez, Wws, Jesse Viviano, HenkeB, Saaya, Cambrant, Surturz, X201, Philippe, Greg L, Radimvice, Rehnn83, Scepia, Mdz, Ptoboley, JAnDbot, Deflective, Davewho2, MER-C, RogierBrussee, VoABot II, Eclipsed aurora, JNW, Nikevich, Cbturner46, HubmaN, EdBever, Wideshanks, ISC PB, Sbierwagen, GCFreak2, Jevansen, Remi0o, Orichalque, Aninhumer, UnicornTapestry, VolkovBot, ICE77, TXiKiBoT, A4bot, Andrew.baine, T-bonham, JayC, Gazno, Plr4ever, ^demonBot2,

Cybermaster~enwiki, Self-Perfection, Labalius, Milan Keršláger, VanishedUserABC, Senpai71, Mike4ty4, Logan, SieBot, Miremare, Jeryobject, Bobanater, EnOreg, Jack1956, Hello71, Joey Eads, OKBot, MarkMLl, ClueBot, C xong, PipepBot, Micky750k, Sonu mangla, Rilak, Matsuiny2004, Owengerig, Martyulrich, Weeniewhite, DumZiBoT, C. A. Russell, Fujimuji, Dr zepsuj~enwiki, Addbot, AvayaLive, Magus732, SpellingBot, Kman543210, Leszek Jańczuk, Download, ChenzwBot, AgadaUrbanit, Lightbot, Windward1, Legobot, Luckas-bot, Yobot, Ptbotgourou, TaBOT-zerem, Wonderfl, AnomieBOT, Rubinbot, Shieldforyoureyes, Fromageestciel, MehrdadAfshari, ArthurBot, Xqbot, Capricorn42, RibotBOT, Darkink, Surv1v4l1st, MetaNest, In2thats12, McHildinger, Jfmantis, Ale07, EmausBot, WikitanvirBot, Autarchprinceps, Dewritech, Thecheesykid, Tksharpless, WikipedianD, Odysseus1479, Bomazi, ThePowerofX, ClueBot NG, MelbourneStar, Robin400, Helpful Pixie Bot, Wbm1058, Goldenshimmer, Isacdaavid, BattyBot, The Illusive Man, Tagremover, Cibban, Ducknish, Dexbot, Fishbone99, Jodosma, Alonduro, Comp.arch, R.tullyjr, Sofia Koutsouveli, Biblioworm, Pyrotle, Pink love 1998, NomanAliArain, Mantraman701, MusikBot, We talk about PAE, Fmadd and Anonymous: 373

- **History of general-purpose CPUs** *Source:* https://en.wikipedia.org/wiki/History_of_general-purpose_CPUs?oldid=717079748 *Contributors:* RTC, Thue, David Edgar, DavidCary, Andreas Kaufmann, Smyth, Dyl, Smalljim, Chbarts, Liao, Guy Harris, Arthena, Dominic, Woohookitty, Eyreland, Pmj, Mr.Unknown, StuartBrady, Rwwww, Bcbell, SmackBot, Chris the speller, QTCaptain, Frap, Harryboyles, JHunterJ, Twas Now, Electron9, Widefox, IanOsgood, Michaeldim, Anaxial, R'n'B, J.delanoy, Public Menace, Osndok, Babedacus, Ajfweb, TedColes, Andy Dingley, Flyer22 Reborn, Sfan00 IMG, Rilak, Czarkoff, Niceguyedc, Muhherfuhher, Dsimic, Ghettoblaster, Lightbot, Jarble, Vincent stehle, Yobot, Fraggle81, AnomieBOT, Jim1138, Champlax, Smallman12q, FrescoBot, Bookalign, Dewritech, Wikipelli, Dcirovic, ClueBot NG, Ernest3.141, Nucius, BG19bot, Fredo699, Kyzor, Mark Arsten, Hebert Peró, Featherwinglove, Archonof and Anonymous: 56

- **Processor design** *Source:* https://en.wikipedia.org/wiki/Processor_design?oldid=713757757 *Contributors:* AxelBoldt, Derek Ross, Mav, Ap, Aldie, Nate Silva, Mudlock, Ray Van De Walker, SimonP, Hannes Hirzel, Maury Markowitz, David spector, Rlee0001, Mintguy, Edward, RTC, Oystein, Michael Hardy, Tim Starling, GABaker, Kwertii, Theanthrope, Alfio, Goatasaur, TUF-KAT, BigFatBuddha, Jiang, Dmsar, Agent Smith, Andrewman327, Furrykef, David Shay, Thue, AnthonyQBachler, Murray Langton, Fredrik, Chris Roy, Tim Ivorson, Iain.mcclatchie, Wjbeaty, Ancheta Wis, DavidCary, Captain Rotundo, Sunny256, DO'Neil, AlistairMcMillan, Solipsist, VampWillow, C17GMaster, Beland, Hgfernan, Sam Hocevar, Pm215, Frankchn, Jkl, Zombiejesus, Tristan Schmelcher, Pixel8, Dyl, Clement Cherlin, Sietse Snel, R. S. Shaw, Matt Britt, Cohesion, Liao, Guy Harris, YDZ, Jm51, ChrisJMoor, Kelly Martin, JeremyA, Trevor Andersen, ??????, MC MasterChef, Kbdank71, Olivier Teuliere, Grammarbot, Ketiltrout, Angusmclellan, Drrngrvy, Intgr, Banaticus, YurikBot, RobotE, Charles Gaudette, Stephenb, CarlHewitt, ZacBowling, JulesH, Panscient, Neomagus00, Benandorsqueaks, Benhoyt, SmackBot, Blue520, Scifiintel, ActiveSelective, Patrickdepinguin, Kurykh, TimBentley, QTCaptain, Frap, AcidPenguin9873, JonHarder, LouScheffer, UU, Judeh101, TechPurism, CRACK-A-BACK, Johncatsoulis, Ivan Kulkov, 16@r, Dicklyon, Waggers, Tawkerbot2, Mattbr, Andkore, Driver01z, JacobBramley, AntiVandalBot, Gioto, JAnDbot, Arch dude, Lebroyl, GermanX, Gwern, LordAnubisBOT, Niks1024, Warut, Su-steveri, VolkovBot, AlnoktaBOT, Jamelan, Nagy, Mipsseo, Yngvarr, Jerryobject, Enochhwang, Jdaloner, Miguel.mateo, Huku-chan, Tuxa, PipepBot, Snigbrook, Rilak, PixelBot, Ianbeveridge, DumZiBoT, Augustojv, Zodon, Legobot, Luckas-bot, Yobot, AnomieBOT, Citation bot, Obersachsebot, Agasta, Crzer07, FrescoBot, Jujutacular, Knoppson, Ripchip Bot, DSW-X-Groove, Dewritech, MonoALT, Cogiati, Ego White Tray, ChuispastonBot, Ashish Gaikwad, Curb Chain, Northamerica1000, Meatsgains, Cyberbot II, ChrisGualtieri, Tagremover, Kolarp, BurritoBazooka, ScotXW, Encrypto1, Laurengranath, Jaffacakemonster53, Bhattasamuel and Anonymous: 149

- **Very long instruction word** *Source:* https://en.wikipedia.org/wiki/Very_long_instruction_word?oldid=708767132 *Contributors:* Zundark, The Anome, Youssefsan, Roadrunner, Rade Kutil, Maury Markowitz, Hephaestos, Michael Hardy, Furrykef, Jnc, Raul654, DavidCary, VampWillow, Neilc, Chowbok, Phe, AlexanderWinston, Hellisp, Abdull, DmitryKo, Jason Carreiro, Perey, Real NC, Smyth, Chub~enwiki, Dyl, CanisRufus, R. S. Shaw, Downchuck, Liao, Fg, Andrewpmk, Suruena, RJFJR, Mr z, AlbertCahalan~enwiki, Alecv, Marudubshinki, Kbdank71, FlaBot, Fresheneesz, Anrie Nord, Personman, Gaius Cornelius, Jeff Carr, Zwobot, JLaTondre, SmackBot, Nihonjoe, Joshfisher, RichardYoung, QTCaptain, Thumperward, JonHarder, CTho, Treforsouthwell, Pizzadeliveryboy, Dbtfz, Darktemplar, RekishiEJ, Zarex, Nczempin, Hasturkun, Thijs!bot, Nonagonal Spider, Geniac, Destynova, EagleFan, Calltech, Olsonist, Itegem, Katalaveno, Jrodor, Ajfweb, Mbinu, Zidonuke, Sleibson, Benoit.dinechin, Bichito, Mikeharris111, Sfan00 IMG, PixelBot, B.Zsolt, Dsimic, Addbot, Mortense, Download, Sillyfolkboy, Lightbot, Legobot, Amirobot, Nallimbot, AnomieBOT, JohnnyHom, Xqbot, Capricorn42, Wcoole, Psychlohexane, Maggyero, Mreftel, RjwilmsiBot, Dewritech, Serketan, Brainflakes, Trveller, Rezabot, Mdann52, Jimw338, Tagremover, Jrmrjnck, Mark.piglet.chambers, ScotXW, 0xFEEDBACC, Satorit and Anonymous: 108

- **Dataflow architecture** *Source:* https://en.wikipedia.org/wiki/Dataflow_architecture?oldid=702700423 *Contributors:* Sander~enwiki, Sam Hocevar, Dyl, R. S. Shaw, Mdd, RJFJR, Firsfron, Perrella, Korg, Dleonard, Albedo, Sfnhltb, Rwwww, SmackBot, Chris the speller, Bluebot, Can't sleep, clown will eat me, Frap, Racklever, Whpq, Radagast83, Brittannica, MHMcCabe, Krauss, Widefox, PhilKnight, Olsonist, Panas, EmilioB, MaD70, PeterChubb, Skadron, EnOreg, MarkMLl, MarieLG, Addbot, Yobot, AnomieBOT, Kavanden, Xqbot, Omnipaedista, DataflowTech, Gf uip, EmausBot, ChuispastonBot, Mizaoku, Mohamed-Ahmed-FG, Gauravsaxena81 and Anonymous: 21

- **Systolic array** *Source:* https://en.wikipedia.org/wiki/Systolic_array?oldid=715290443 *Contributors:* Michael Hardy, Kku, Dysprosia, Doradus, Zoicon5, Jnc, Alexandre Cruz, Stewartadcock, Hadal, Ruakh, SpellBott, Lockeownzj00, Conte.carli, Dyl, Whosyourjudas, Pearle, Cscott, BD2412, MartinRudat, Petri Krohn, SmackBot, Jcarroll, Optikos, TimBentley, Oli Filth, Letdorf, Frap, Cybercobra, DMacks, Abraxa~enwiki, Electron9, Gah4, Tiggerjay, VolkovBot, RainierHa, Natg 19, Karl-tech, ParallelWolverine, Rainier3, Lightmouse, Niceguyedc, Addbot, Nachimdr, Jarble, Yobot, Editor711, AnomieBOT, Nisheethg, DrilBot, MastiBot, Ales-76, Ajbonkoski, Klbrain, Wbm1058, Frze, DPL bot, AmiArnab, Garfield Garfield, Socaacos and Anonymous: 43

8.14.2 Images

- **File:Ambox_important.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/b/b4/Ambox_important.svg *License:* Public domain *Contributors:* Own work, based off of Image:Ambox scales.svg *Original artist:* Dsmurat (talk · contribs)

- **File:Commons-logo.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/4/4a/Commons-logo.svg *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?

- **File:Computer-aj_aj_ashton_01.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/d/d7/Desktop_computer_clipart_-_Yellow_theme.svg *License:* CC0 *Contributors:* https://openclipart.org/detail/105871/computeraj-aj-ashton-01 *Original artist:* AJ from openclipart.org

- **File:Edit-clear.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/f/f2/Edit-clear.svg *License:* Public domain *Contributors:* The *Tango! Desktop Project*. *Original artist:*

The people from the Tango! project. And according to the meta-data in the file, specifically: “Andreas Nilsson, and Jakub Steiner (although minimally).”

- **File:IBMVacuumTubeModule.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/e/e7/IBMVacuumTubeModule.jpg *License:* CC BY 2.0 *Contributors:* ? *Original artist:* ?
- **File:IBM_PowerPC601_PPC601FD-080-2_top.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/7/7d/IBM_PowerPC601_PPC601FD-080-2_top.jpg *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Internet_map_1024.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/d/d2/Internet_map_1024.jpg *License:* CC BY 2.5 *Contributors:* Originally from the English Wikipedia; description page is/was here. *Original artist:* The Opte Project
- **File:KL_Intel_C8008-1.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/b/ba/KL_Intel_C8008-1.jpg *License:* GFDL *Contributors:* CPU Collection Konstantin Lanzet
  Camera: Canon EOS 400D *Original artist:* Konstantin Lanzet
- **File:KL_Sun_UltraSparc.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/9/95/KL_Sun_UltraSparc.jpg *License:* CC-BY-SA-3.0 *Contributors:* CPU collection Konstantin Lanzet *Original artist:* Konstantin Lanzet (with permission)
- **File:MIPS_Architecture_(Pipelined).svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/e/ea/MIPS_Architecture_%28Pipelined%29.svg *License:* Public domain *Contributors:* Own work *Original artist:* Inductiveload
- **File:Mergefrom.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/0/0f/Mergefrom.svg *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Nuvola_apps_ksim.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/8/8d/Nuvola_apps_ksim.png *License:* LGPL *Contributors:* http://icon-king.com *Original artist:* David Vignoni / ICON KING
- **File:Question_book-new.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg *License:* Cc-by-sa-3.0 *Contributors:*
  Created from scratch in Adobe Illustrator. Based on Image:Question book.png created by User:Equazcion *Original artist:* Tkgd2007
- **File:Text_document_with_red_question_mark.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/a/a4/Text_document_with_red_question_mark.svg *License:* Public domain *Contributors:* Created by bdesham with Inkscape; based upon Text-x-generic.svg from the Tango project. *Original artist:* Benjamin D. Esham (bdesham)
- **File:Wikibooks-logo-en-noslogan.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/d/df/Wikibooks-logo-en-noslogan.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Bastique, User:Ramac et al.
- **File:Yunsup_Lee_holding_RISC_V_prototype_chip.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/7/7a/Yunsup_Lee_holding_RISC_V_prototype_chip.jpg *License:* CC0 *Contributors:* Yunsup Lee holding RISC V prototype chip *Original artist:* Derrick Coetzee (User:Dcoetzee)

8.14.3 Content license

- Creative Commons Attribution-Share Alike 3.0