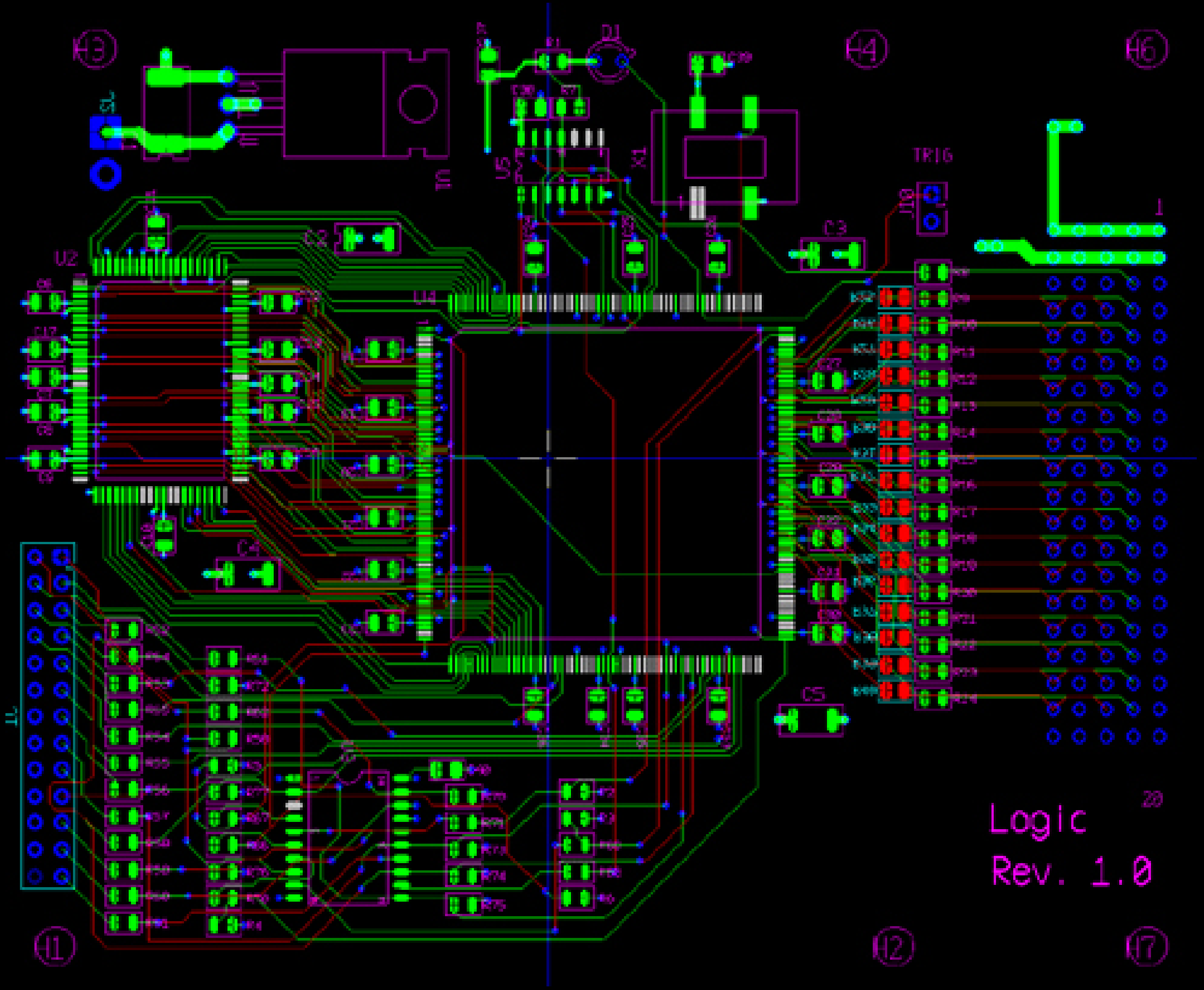
Handbook of

Gate Arrays and Digital Registers



Twanna Christian

Marvin Childs

First Edition, 2012

ISBN 978-81-323-0912-3

Published by:
Academic Studio
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Field–Programmable Gate Array

Chapter 2 - Application–Specific Integrated Circuit

Chapter 3 - Programmable Array Logic

Chapter 4 - Programmable Logic Device

Chapter 5 - Delay–Locked Loop & Complex Programmable Logic Device

Chapter 6 - Partial Re–Configuration & Rent's Rule

Chapter 7 - Analog-to-Digital Converter

Chapter 8 - Digital-to-Analog Converter

Chapter 9 - Control Register

Chapter 10 - Hardware Register and Processor Register

Chapter 11 - Flag Word and FLAGS Register (Computing)

Chapter 12 - Linear Feedback Shift Register

Chapter 13 - Program Counter and Register File

Chapter 14 - Register Renaming

Chapter 15 - Register Window and Shift Register

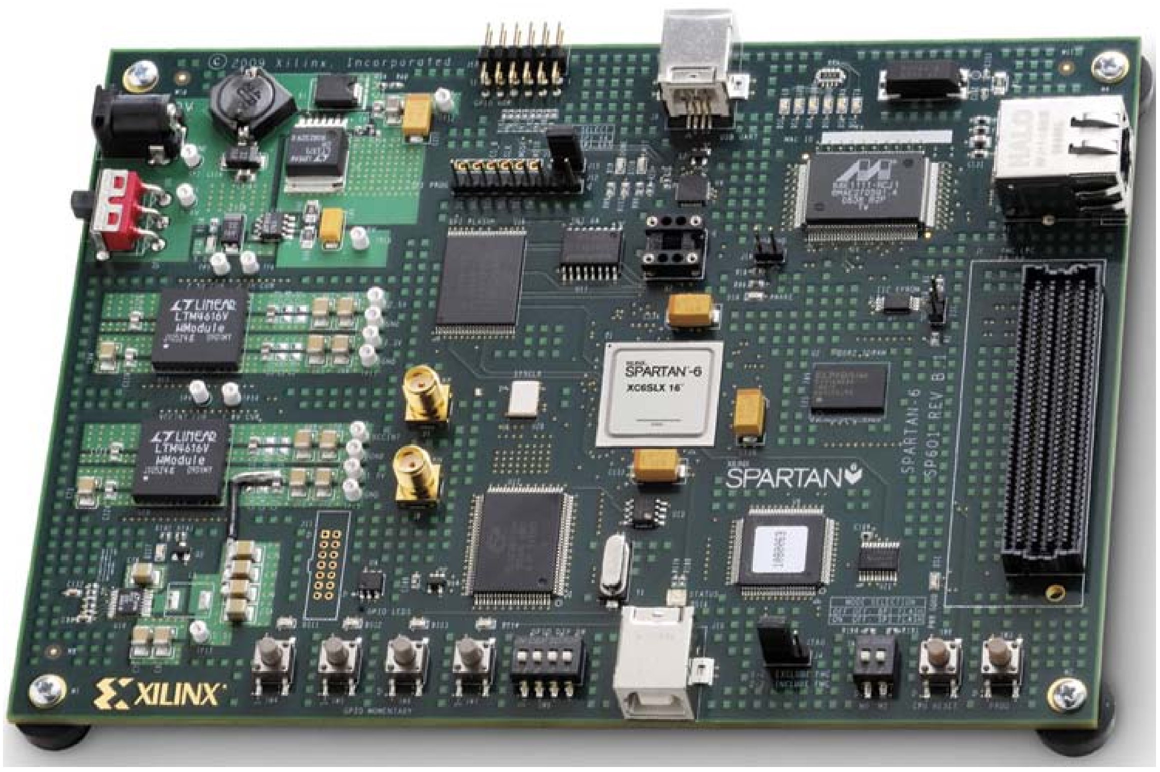
Chapter 16 - Word (Computing)

Chapter 17 - Byte

Chapter 1

Field-Programmable Gate Array

An Altera Stratix IV GX FPGA



An example of a Xilinx Spartan 6 FPGA programming/evaluation board

A **Field-programmable Gate Array** (**FPGA**) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare). FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, partial re-configuration of the portion of the design and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications.

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—somewhat like many (changeable) logic gates that can be inter-wired in (many) different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

In addition to digital functions, some FPGAs have analog features. The most common analog feature is programmable slew rate and drive strength on each output pin, allowing the engineer to set slow rates on lightly loaded pins that would otherwise ring unacceptably, and to set stronger, faster rates on heavily loaded pins on high-speed channels that would otherwise run too slow. Another relatively common analog feature is differential comparators on input pins designed to be connected to differential signaling channels. A few "mixed signal FPGAs" have integrated peripheral Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) with analog signal conditioning blocks allowing them to operate as a system-on-a-chip. Such devices blur the line between an FPGA, which carries digital ones and zeros on its internal programmable interconnect fabric, and field-programmable analog array (FPAA), which carries analog values on its internal programmable interconnect fabric.

History

The FPGA industry sprouted from programmable read-only memory (PROM) and programmable logic devices (PLDs). PROMs and PLDs both had the option of being programmed in batches in a factory or in the field (field programmable), however programmable logic was hard-wired between logic gates.

In the late 1980s the Naval Surface Warfare Department funded an experiment proposed by Steve Casselman to develop a computer that would implement 600,000 reprogrammable gates. Casselman was successful and a patent related to the system was issued in 1992.

Some of the industry's foundational concepts and technologies for programmable logic arrays, gates, and logic blocks are founded in patents awarded to David W. Page and LuVerne R. Peterson in 1985.

Xilinx Co-Founders, Ross Freeman and Bernard Vonderschmitt, invented the first commercially viable field programmable gate array in 1985 – the XC2064. The XC2064 had programmable gates and programmable interconnects between gates, the beginnings of a new technology and market. The XC2064 boasted a mere 64 configurable logic blocks (CLBs), with two 3-input lookup tables (LUTs). More than 20 years later, Freeman was entered into the National Inventors Hall of Fame for his invention.

Xilinx continued unchallenged and quickly growing from 1985 to the mid-1990s, when competitors sprouted up, eroding significant market-share. By 1993, Actel was serving about 18 percent of the market.

The 1990s were an explosive period of time for FPGAs, both in sophistication and the volume of production. In the early 1990s, FPGAs were primarily used in telecommunications and networking. By the end of the decade, FPGAs found their way into consumer, automotive, and industrial applications.

FPGAs got a glimpse of fame in 1997, when Adrian Thompson, a researcher working at the University of Sussex, merged genetic algorithm technology and FPGAs to create a sound recognition device. Thomson's algorithm configured an array of 10 x 10 cells in a Xilinx FPGA chip to discriminate between two tones, utilising analogue features of the digital chip. The application of genetic algorithms to the configuration of devices like FPGA's is now referred to as Evolvable hardware

Modern developments

A recent trend has been to take the coarse-grained architectural approach a step further by combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and related peripherals to form a complete "system on a programmable chip". This work mirrors the architecture by Ron Perlof and Hana Potash of Burroughs Advanced Systems Group which combined a reconfigurable CPU architecture on a single chip called the SB24. That work was done in 1982. Examples of such hybrid technologies can be found in the Xilinx Virtex-II PRO and Virtex-4 devices, which include one or more PowerPC processors embedded within the FPGA's logic fabric. The Atmel FPSLIC is another such device, which uses an AVR processor in combination with Atmel's programmable logic architecture. The Actel SmartFusion devices incorporate an ARM architecture Cortex-M3 hard processor core (with up to 512kB of flash and 64kB of RAM) and analog peripherals such as a multi-channel ADC and DACs to their flash-based FPGA fabric.

An alternate approach to using hard-macro processors is to make use of soft processor cores that are implemented within the FPGA logic.

As previously mentioned, many modern FPGAs have the ability to be reprogrammed at "run time," and this is leading to the idea of reconfigurable computing or reconfigurable systems — CPUs that reconfigure themselves to suit the task at hand. The Mitrion Virtual Processor from Mitrionics is an example of a reconfigurable soft processor, implemented on FPGAs. However, it does not support dynamic reconfiguration at runtime, but instead adapts itself to a specific program.

Additionally, new, non-FPGA architectures are beginning to emerge. Software-configurable microprocessors such as the Stretch S5000 adopt a hybrid approach by providing an array of processor cores and FPGA-like programmable cores on the same chip.

Gates

- 1987: 9,000 gates, Xilinx
- 1992: 600,000, Naval Surface Warfare Department
- Early 2000s: Millions

Market size

- 1985: First commercial FPGA technology invented by Xilinx
- 1987: $14 million
- ~1993: >$385 million
- 2005: $1.9 billion

- 2010 estimates: $2.75 billion

FPGA design starts

- 10,000
- 2005: 80,000
- 2008: 90,000

FPGA comparisons

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts. A study has shown that designs implemented on FPGAs need on average 18 times as much area, draw 7 times as much dynamic power, and are 3 times slower than the corresponding ASIC implementations.

An Altera Cyclone II FPGA, on an Altera teraSIC DE1 Prototyping board.

Advantages include the ability to re-program in the field to fix bugs, and may include a shorter time to market and lower non-recurring engineering costs. Vendors can also take a middle road by developing their hardware on ordinary FPGAs, but manufacture their final version so it can no longer be modified after the design has been committed.

Xilinx claims that several market and technology dynamics are changing the ASIC/FPGA paradigm:

- Integrated circuit costs are rising aggressively
- ASIC complexity has lengthened development time
- R&D resources and headcount are decreasing

- Revenue losses for slow time-to-market are increasing
- Financial constraints in a poor economy are driving low-cost technologies

These trends make FPGAs a better alternative than ASICs for a larger number of higher-volume applications than they have been historically used for, to which the company attributes the growing number of FPGA design starts.

Some FPGAs have the capability of partial re-configuration that lets one portion of the device be re-programmed while other portions continue running.

Versus complex programmable logic devices

The primary differences between CPLDs (Complex Programmable Logic Devices) and FPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnect. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) but also far more complex to design for.

Another notable difference between CPLDs and FPGAs is the presence in most FPGAs of higher-level embedded functions (such as adders and multipliers) and embedded memories, as well as to have logic blocks implement decoders or mathematical functions.

Security considerations

With respect to security, FPGAs have both advantages and disadvantages as compared to ASICs or secure microprocessors. FPGAs' flexibility makes malicious modifications during fabrication a lower risk. For many FPGAs, the loaded design is exposed while it is loaded (typically on every power-on). To address this issue, some FPGAs support bitstream encryption.

Applications

Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.

FPGAs especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture. One such area is code breaking, in particular brute-force attack, of cryptographic algorithms.

FPGAs are increasingly used in conventional high performance computing applications where computational kernels such as FFT or Convolution are performed on the FPGA instead of a microprocessor.

The inherent parallelism of the logic resources on an FPGA allows for considerable computational throughput even at a low MHz clock rates. The flexibility of the FPGA allows for even higher performance by trading off precision and range in the number format for an increased number of parallel arithmetic units. This has driven a new type of processing called reconfigurable computing, where time intensive tasks are offloaded from software to FPGAs.

The adoption of FPGAs in high performance computing is currently limited by the complexity of FPGA design compared to conventional software and the turn-around times of current design tools.

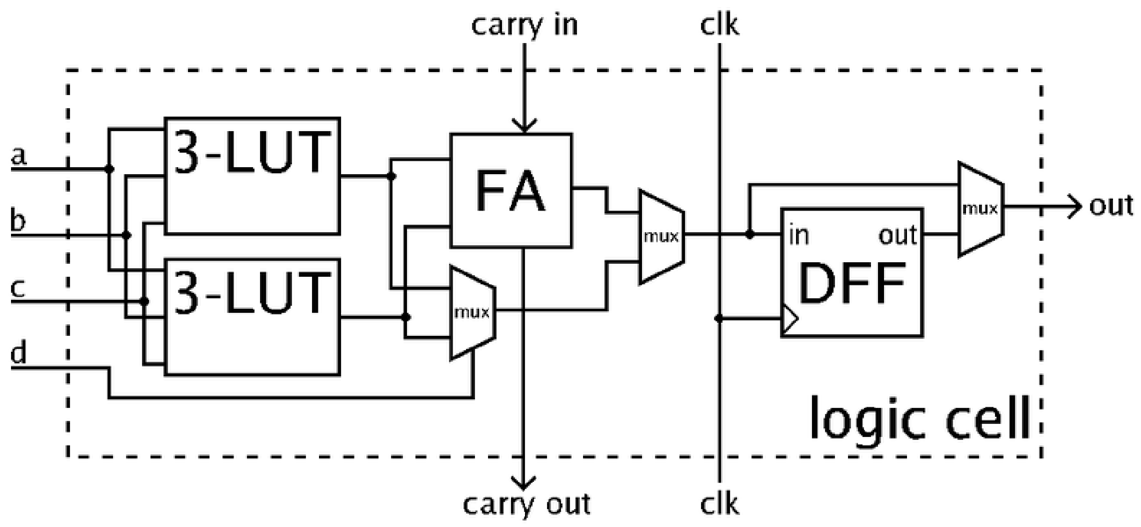
Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

Architecture

The most common FPGA architecture consists of an array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.

An application circuit must be mapped into an FPGA with adequate resources. While the number of CLBs/LABs and I/Os required is easily determined from the design, the number of routing tracks needed may vary considerably even among designs with the same amount of logic. For example, a crossbar switch requires much more routing than a systolic array with the same gate count. Since unused routing tracks increase the cost (and decrease the performance) of the part without providing any benefit, FPGA manufacturers try to provide just enough tracks so that most designs that will fit in terms of LUTs and IOs can be routed. This is determined by estimates such as those derived from Rent's rule or by experiments with existing designs.

In general, a logic block (CLB or LAB) consists of a few logical cells (called ALM, LE, Slice etc). A typical cell consists of a 4-input Lookup table (LUT), a Full adder (FA) and a D-type flip-flop, as shown below. The LUTs are in this figure split into two 3-input LUTs. In *normal mode* those are combined into a 4-input LUT through the left mux. In *arithmetic* mode, their outputs are fed to the FA. The selection of mode is programmed into the middle mux. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space.



Simplified example illustration of a logic cell

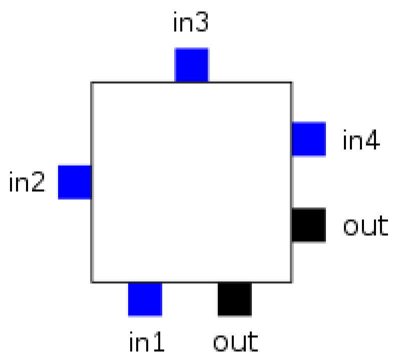
ALMs and Slices usually contains 2 or 4 structures similar to the example figure, with some shared signals.

CLBs/LABs typically contains a few ALMs/LEs/Slices.

In recent years, manufacturers have started moving to 6-input LUTs in their high performance parts, claiming increased performance.

Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed.

For this example architecture, the locations of the FPGA logic block pins are shown below.



Logic Block Pin Locations

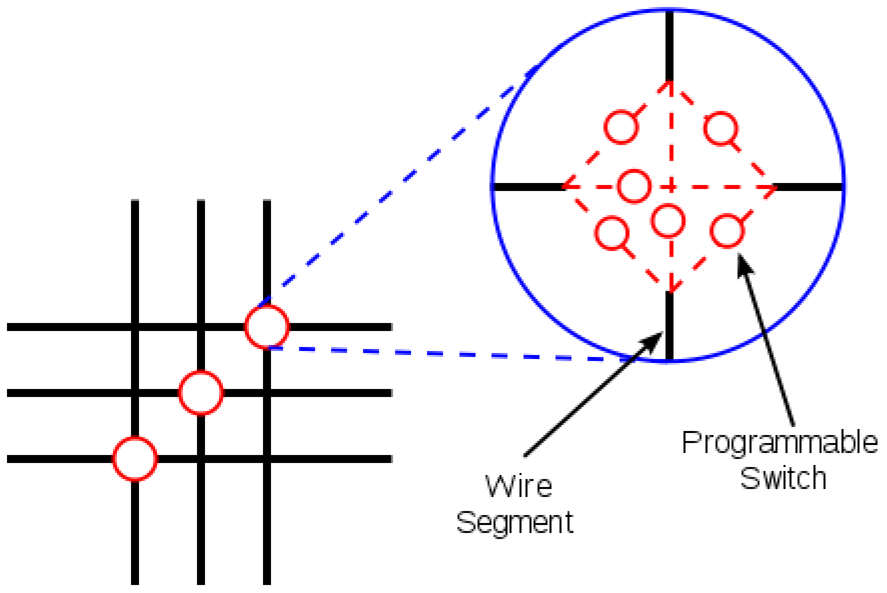
Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block.

Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

Similarly, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

Whenever a vertical and a horizontal channel intersect, there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure below illustrates the connections in a switch box.



Switch box topology

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed IO logic and embedded memories.

FPGAs are also widely used for systems validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, reducing the time-to-market.

FPGA design and programming

To define the behavior of the FPGA, the user provides a hardware description language (HDL) or a schematic design. The HDL form is more suited to work with large structures because it's possible to just specify them numerically rather than having to draw every piece by hand. However, schematic entry can allow for easier visualisation of a design.

Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA.

Going from schematic/HDL source files to actual configuration: The source files are fed to a software suite from the FPGA/CPLD vendor that through different steps will produce a file. This file is then transferred to the FPGA/CPLD via a serial interface (JTAG) or to an external memory device like an EEPROM.

The most common HDLs are VHDL and Verilog, although in an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level through the introduction of alternative languages. National Instrument's LabVIEW graphical programming language (sometimes referred to as "G") has an FPGA add-in module available to target and program FPGA hardware.

To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process. These predefined circuits are commonly called *IP cores*, and are available from FPGA vendors and third-party IP suppliers (rarely free, and typically released under proprietary licenses). Other predefined circuits are available from developer communities such as OpenCores (typically released under free and open source licenses such as the GPL, BSD or similar license), and other sources.

In a typical design flow, an FPGA application developer will simulate the design at multiple stages throughout the design process. Initially the RTL description in VHDL or Verilog is simulated by creating test benches to simulate the system and observe results. Then, after the synthesis engine has mapped the design to a netlist, the netlist is translated to a gate level description where simulation is repeated to confirm the synthesis proceeded without errors. Finally the design is laid out in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist.

Basic process technology types

- SRAM - based on static memory technology. In-system programmable and re-programmable. Requires external boot devices. CMOS.
- Antifuse - One-time programmable. CMOS.
- PROM - Programmable Read-Only Memory technology. One-time programmable because of plastic packaging.
- EPROM - Erasable Programmable Read-Only Memory technology. One-time programmable but with window, can be erased with ultraviolet (UV) light. CMOS.
- EEPROM - Electrically Erasable Programmable Read-Only Memory technology. Can be erased, even in plastic packages. Some but not all EEPROM devices can be in-system programmed. CMOS.
- Flash - Flash-erase EPROM technology. Can be erased, even in plastic packages. Some but not all flash devices can be in-system programmed. Usually, a flash cell is smaller than an equivalent EEPROM cell and is therefore less expensive to manufacture. CMOS.
- Fuse - One-time programmable. Bipolar.

Major manufacturers

Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. Together, they control over 80 percent of the market, with Xilinx alone representing over 50 percent.

Both Xilinx and Altera provide free Windows and Linux design software.

Other competitors include Lattice Semiconductor (SRAM based with integrated configuration Flash, instant-on, low power, live reconfiguration), Actel (antifuse, flash-based, mixed-signal), SiliconBlue Technologies (extremely low power SRAM-based FPGAs with option integrated nonvolatile configuration memory), Achronix (RAM based, 1.5 GHz fabric speed) who will be building their chips on Intels' state-of-the art 22 nm process, and QuickLogic (handheld focused CSSP, no general purpose FPGAs).

In March 2010, Tabula announced their new FPGA technology that uses time-multiplexed logic and interconnect for greater potential cost savings for high-density applications.

Chapter 2

Application-Specific Integrated Circuit

An **application-specific integrated circuit** (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. For example, a chip designed solely to run a cell phone is an ASIC. Application-specific standard products (ASSPs) are intermediate between ASICs and industry standard integrated circuits like the 7400 or the 4000 series.

As feature sizes have shrunk and design tools improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from 5,000 gates to over 100 million. Modern ASICs often include entire 32-bit processors, memory blocks including ROM, RAM, EEPROM, Flash and other large building blocks. Such an ASIC is often termed a SoC (system-on-a-chip). Designers of digital ASICs use a hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs.

Field-programmable gate arrays (FPGA) are the modern-day technology for building a breadboard or prototype from standard parts; programmable logic blocks and programmable interconnects allow the same FPGA to be used in many different applications. For smaller designs and/or lower production volumes, FPGAs may be more cost effective than an ASIC design even in production. The non-recurring engineering cost of an ASIC can run into the millions of dollars.

History

The initial ASICs used gate array technology. Ferranti produced perhaps the first gate-array, the ULA (Uncommitted Logic Array), around 1980. An early successful commercial application was the ULA circuitry found in the 8-bit ZX81 and ZX Spectrum low-end personal computers, introduced in 1981 and 1982. These were used by Sinclair Research (UK) essentially as a low-cost I/O solution aimed at handling the computer's graphics. Some versions of ZX81/Timex Sinclair 1000 used just four chips (ULA, 2Kx8 RAM, 8Kx8 ROM, Z80A CPU) to implement an entire mass-market personal computer with built-in BASIC interpreter.

Customization occurred by varying the metal interconnect mask. ULAs had complexities of up to a few thousand gates. Later versions became more generalized, with different base dies customised by both metal and polysilicon layers. Some base dies include RAM elements.

Standard cell design

In the mid 1980s, a designer would choose an ASIC manufacturer and implement their design using the design tools available from the manufacturer. While third-party design tools were available, there was not an effective link from the third-party design tools to the layout and actual semiconductor process performance characteristics of the various ASIC manufacturers. Most designers ended up using factory-specific tools to complete the implementation of their designs. A solution to this problem, which also yielded a much higher density device, was the implementation of Standard Cells. Every ASIC manufacturer could create functional blocks with known electrical characteristics, such as propagation delay, capacitance and inductance, that could also be represented in third-party tools. Standard Cell design is the utilization of these functional blocks to achieve very high gate density and good electrical performance. Standard cell design fits between Gate Array and Full Custom design in terms of both its NRE (Non-Recurring Engineering) and recurring component cost.

By the late 1990s, logic synthesis tools became available. Such tools could compile HDL descriptions into a gate-level netlist. Standard-cell Integrated Circuits (ICs) are designed in the following conceptual stages, although these stages overlap significantly in practice.

1. A team of design engineers starts with a non-formal understanding of the required functions for a new ASIC, usually derived from Requirements analysis.
2. The design team constructs a description of an ASIC to achieve these goals using an HDL. This process is analogous to writing a computer program in a high-level language. This is usually called the RTL (Register transfer level) design.
3. Suitability for purpose is verified by functional verification. This may include such techniques as logic simulation, formal verification, emulation, or creating an equivalent pure software model. Each technique has advantages and disadvantages, and often several methods are used.
4. Logic synthesis transforms the RTL design into a large collection of lower-level constructs called standard cells. These constructs are taken from a standard-cell library consisting of pre-characterized collections of gates (such as 2 input nor, 2 input nand, inverters, etc.). The standard cells are typically specific to the planned manufacturer of the ASIC. The resulting collection of standard cells, plus the needed electrical connections between them, is called a gate-level netlist.
5. The gate-level netlist is next processed by a placement tool which places the standard cells onto a region representing the final ASIC. It attempts to find a placement of the standard cells, subject to a variety of specified constraints.
6. The routing tool takes the physical placement of the standard cells and uses the netlist to create the electrical connections between them. Since the search space is large, this process will produce a “sufficient” rather than “globally-optimal” solution. The output is a file which can be used to create a set of photomasks

enabling a semiconductor fabrication facility (commonly called a 'fab') to produce physical ICs.

7. Given the final layout, circuit extraction computes the parasitic resistances and capacitances. In the case of a digital circuit, this will then be further mapped into delay information, from which the circuit performance can be estimated, usually by static timing analysis. This, and other final tests such as design rule checking and power analysis (collectively called signoff) are intended to ensure that the device will function correctly over all extremes of the process, voltage and temperature. When this testing is complete the photomask information is released for chip fabrication.

These steps, implemented with a level of skill common in the industry, almost always produce a final device that correctly implements the original design, unless flaws are later introduced by the physical fabrication process.

The design steps (or flow) are also common to standard product design. The significant difference is that Standard Cell design uses the manufacturer's cell libraries that have been used in potentially hundreds of other design implementations and therefore are of much lower risk than full custom design. Standard Cells produce a design density that is cost effective, and they can also integrate IP cores and SRAM (Static Random Access Memory) effectively, unlike Gate Arrays.

Gate array design

Gate array design is a manufacturing method in which the diffused layers, i.e. transistors and other active devices, are predefined and wafers containing such devices are held in stock prior to metallization — in other words, unconnected. The physical design process then defines the interconnections of the final device. For most ASIC manufacturers, this consists of from two to as many as nine metal layers, each metal layer running perpendicular to the one below it. Non-recurring engineering costs are much lower, as photolithographic masks are required only for the metal layers, and production cycles are much shorter, as metallization is a comparatively quick process.

Gate array ASICs are always a compromise as mapping a given design onto what a manufacturer held as a stock wafer never gives 100% utilization. Often difficulties in routing the interconnect require migration onto a larger array device with consequent increase in the piece part price. These difficulties are often a result of the layout software used to develop the interconnect.

Pure, logic-only gate array design is rarely implemented by circuit designers today, having been replaced almost entirely by field-programmable devices, such as field-programmable gate arrays (FPGAs), which can be programmed by the user and thus offer minimal tooling charges (non-recurring engineering (NRE)), only marginally increased piece part cost, and comparable performance. Today, gate arrays are evolving into structured ASICs that consist of a large IP core like a CPU, DSP unit, peripherals, standard interfaces, integrated memories SRAM, and a block of reconfigurable, uncommited logic. This shift is largely because ASIC devices are capable of integrating such large blocks of system functionality and "system-on-a-chip" requires far more than just logic blocks.

In their frequent usages in the field, the terms "gate array" and "semi-custom" are synonymous. Process engineers more commonly use the term "semi-custom", while "gate-array" is more commonly used by logic (or gate-level) designers.

Full-custom design

By contrast, full-custom ASIC design defines all the photolithographic layers of the device. Full-custom design is used for both ASIC design and for standard product design.

The benefits of full-custom design usually include reduced area (and therefore recurring component cost), performance improvements, and also the ability to integrate analog components and other pre-designed — and thus fully verified — components, such as microprocessor cores that form a system-on-chip.

The disadvantages of full-custom design can include increased manufacturing and design time, increased non-recurring engineering costs, more complexity in the computer-aided design (CAD) system, and a much higher skill requirement on the part of the design team.

For digital-only designs, however, "standard-cell" cell libraries, together with modern CAD systems, can offer considerable performance/cost benefits with low risk. Automated layout tools are quick and easy to use and also offer the possibility to "hand-tweak" or manually optimize any performance-limiting aspect of the design.

Structured design

Structured ASIC design (also referred to as "platform ASIC design"), is a relatively new term in the industry, resulting in some variation in its definition. However, the basic premise of a structured ASIC is that both manufacturing cycle time and design cycle time are reduced compared to cell-based ASIC, by virtue of there being pre-defined metal layers (thus reducing manufacturing time) and pre-characterization of what is on the silicon (thus reducing design cycle time). One definition states that

> In a "structured ASIC" design, the logic mask-layers of a device are predefined by the ASIC vendor (or in some cases by a third party). Design differentiation and customization is achieved by creating custom metal layers that create custom connections between predefined lower-layer logic elements. "Structured ASIC" technology is seen as bridging the gap between field-programmable gate arrays and "standard-cell" ASIC designs. Because only a small number of chip layers must be custom-produced, "structured ASIC" designs have much smaller non-recurring expenditures (NRE) than "standard-cell" or "full-custom" chips, which require that a full mask set be produced for every design.

This is effectively the same definition as a gate array. What makes a structured ASIC different is that in a gate array, the predefined metal layers serve to make manufacturing turnaround faster. In a structured ASIC, the use of predefined metallization is primarily to reduce cost of the mask sets as well as making the design cycle time significantly shorter. For example, in a cell-based or gate-array design the user must often design power, clock, and test structures themselves; these are predefined in most structured ASICs and therefore can save time and expense for the designer compared to gate-array. Likewise, the design tools used for structured ASIC can be substantially lower cost and easier (faster) to use than cell-based tools, because they do not have to perform all the functions that cell-based tools do. In some cases, the structured ASIC vendor requires that customized tools for their device (e.g., custom physical synthesis) be used, also allowing for the design to be brought into manufacturing more quickly.

One other important aspect about structured ASIC is that it allows intellectual property (IP) that is common to certain applications or industry segments to be "built in", rather than "designed in". By building the IP directly into the architecture the designer can again save both time and money compared to designing IP into a cell-based ASIC.

Cell libraries, IP-based design, hard and soft macros

Cell libraries of logical primitives are usually provided by the device manufacturer as part of the service. Although they will incur no additional cost, their release will be covered by the terms of a non-disclosure agreement (NDA) and they will be regarded as intellectual property by the manufacturer. Usually their physical design will be pre-defined so they could be termed "hard macros".

What most engineers understand as "intellectual property" are IP cores, designs purchased from a third-party as sub-components of a larger ASIC. They may be provided as an HDL description (often termed a "soft macro"), or as a fully routed design that could be printed directly onto an ASIC's mask (often termed a hard macro). Many organizations now sell such pre-designed cores — CPUs, Ethernet, USB or telephone interfaces — and larger organizations may have an entire department or division to produce cores for the rest of the organization. Indeed, the wide range of functions now available is a significant factor in the phenomenal increase in electronics in the late 1990s and early 2000s; as a core takes a lot of time and investment to create, its re-use and further development cuts product cycle times dramatically and creates better products. Additionally, organizations such as OpenCores are collecting free IP cores paralleling the open source movement in software.

Soft macros are often process-independent, i.e., they can be fabricated on a wide range of manufacturing processes and different manufacturers. Hard macros are process-limited and usually further design effort must be invested to migrate (port) to a different process or manufacturer.

Multi-project wafers

Some manufacturers offer Multi-Project Wafers (MPW) as a method of obtaining low cost prototypes. Often called shuttles, these MPW, containing several designs, run at regular, scheduled intervals on a "cut and go" basis, usually with very little liability on the part of the manufacturer. The contract involves the assembly and packaging of a handful of devices. The service usually involves the supply of a physical design data base i.e. masking information or Pattern Generation (PG) tape. The manufacturer is often referred to as a "silicon foundry" due to the low involvement it has in the process. .

ASIC suppliers

There are two different types of ASIC suppliers, IDM and fabless. An IDM supplier's ASIC product is based in large part on proprietary technology such as design tools, IP, packaging, and usually although not necessarily the process technology. Fabless ASIC suppliers rely almost exclusively on outside suppliers for their technology. The classification can be confusing since several IDM's are also fabless semiconductor companies.

IDM ASIC suppliers

- Avago Technologies
- Elmos Semiconductor
- Cavium Networks
- Fujitsu
- Freescale
- HITACHI
- IBM
- Infineon Technologies
- LSI Corporation
- Marvell Semiconductor
- NEC
- NXP Semiconductors
- ON Semiconductor
- Renesas
- Samsung
- STMicroelectronics
- Texas Instruments
- Toshiba
- TSMC

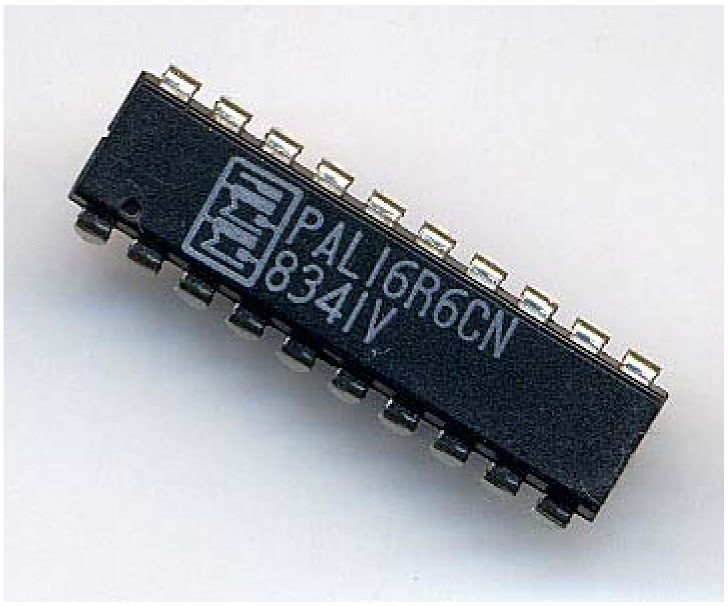
Fabless ASIC suppliers

- Alchip
- Aeroflex Colorado Springs
- Brite Semiconductor

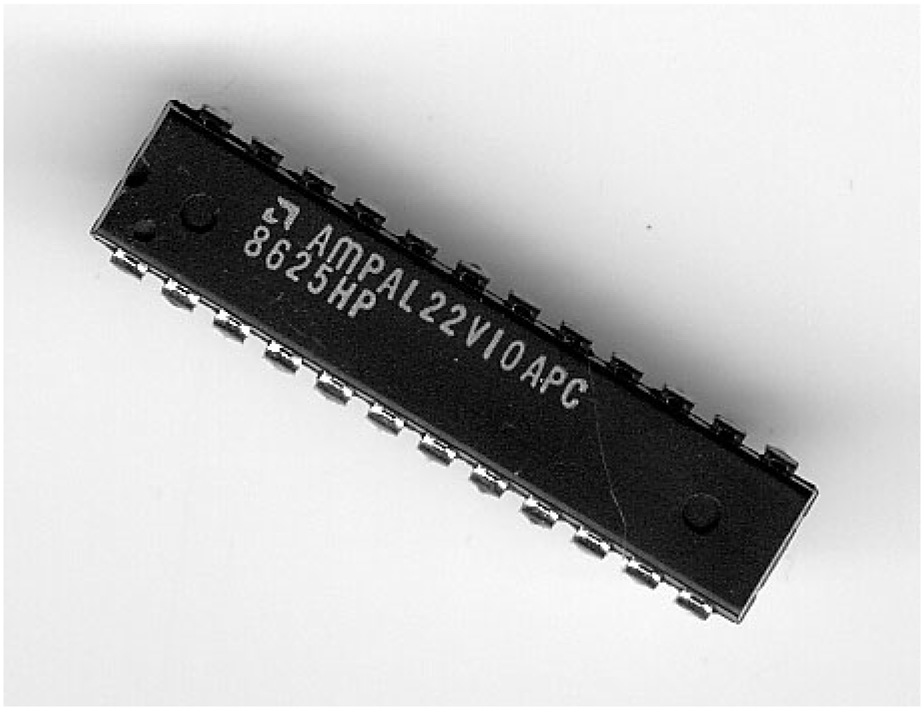
- Custom Silicon Solutions
- eASIC
- eSilicon
- Faraday Technology
- Hong Kong Science and Technology Parks Corporation
- JVD Inc.
- Marvell Semiconductor
- MOSIS
- Nvidia
- Open-Silicon
- PMC Sierra
- Qualcomm
- Socle
- System to ASIC

Chapter 3

Programmable Array Logic



MMI PAL 16R6 in 20-pin DIP



AMD 22V10 in 24-pin DIP

The term **Programmable Array Logic** (PAL) is used to describe a family of programmable logic device semiconductors used to implement logic functions in digital circuits introduced by Monolithic Memories, Inc. (MMI) in March 1978. MMI obtained a registered trademark on the term PAL for use in "Programmable Semiconductor Logic Circuits". The trademark is currently held by Lattice Semiconductor.

PAL devices consisted of a small PROM (programmable read-only memory) core and additional output logic used to implement particular desired logic functions with few components.

Using specialized machines, PAL devices were "field-programmable". Each PAL device was "one-time programmable" (OTP), meaning that it could not be updated and reused after its initial programming. (MMI also offered a similar family called HAL, or "hard array logic", which were like PAL devices except that they were mask-programmed at the factory.)

Early history

Before PALs were introduced, designers of digital logic circuits would use small-scale integration (SSI) components, such as those in the 7400 series TTL (transistor-transistor logic) family; the 7400 family included a variety of logic building blocks, such as gates (NOT, NAND, NOR, AND, OR), multiplexers (MUXes) and demultiplexers (DEMUXes), flip flops (D-type, JK, etc.) and others. One PAL device would typically replace dozens of such "discrete" logic packages, so the SSI business went into decline as the PAL business took off. PALs were used advantageously in many products, such as minicomputers, as documented in Tracy Kidder's best-selling book "The Soul of a New Machine."

PALs were not the first commercial programmable logic devices; Signetics had been selling its field programmable logic array (FPLA) since 1975. These devices were completely unfamiliar to most circuit designers and were perceived to be too difficult to use. The FPLA had a relatively slow maximum operating speed (due to having both programmable-AND and programmable-OR arrays), was expensive, and had a poor reputation for testability. Another factor limiting the acceptance of the FPLA was the large package, a 600-mil (0.6", or 15.24 mm) wide 28-pin dual in-line package (DIP).

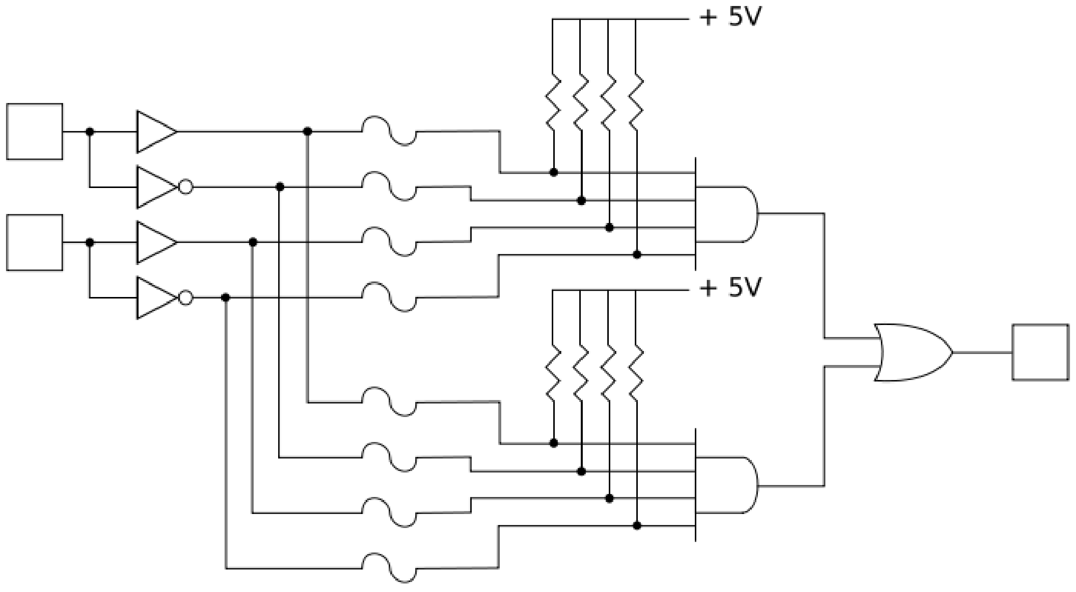
The project to create the PAL device was managed by John Birkner and the actual PAL circuit was designed by H. T. Chua. In a previous job, Birkner had developed a 16-bit processor using 80 standard logic devices. His experience with standard logic led him to believe that user programmable devices would be more attractive to users if the devices were designed to replace standard logic. This meant that the package sizes had to be more typical of the existing devices, and the speeds had to be improved. The PAL met these requirements and was a huge success and were "second sourced" by National Semiconductor, Texas Instruments, and Advanced Micro Devices.

Process technologies

Early PALs were 20-pin DIP components fabricated in silicon using bipolar transistor technology with one-time programmable (OTP) titanium-tungsten programming fuses. Later devices were manufactured by Lattice Semiconductor and Advanced Micro Devices using CMOS technology.

The original 20 and 24-pin PALs were described by MMI as medium-scale integration (MSI) devices.

PAL architecture



Simplified programmable logic device

The programmable elements (shown as a fuse) connect both the true and complemented inputs to the AND gates. These AND gates, also known as *product terms*, are ORed together to form a *sum-of-products* logic array.

The PAL architecture consists of two main components: a logic plane and output logic macrocells.

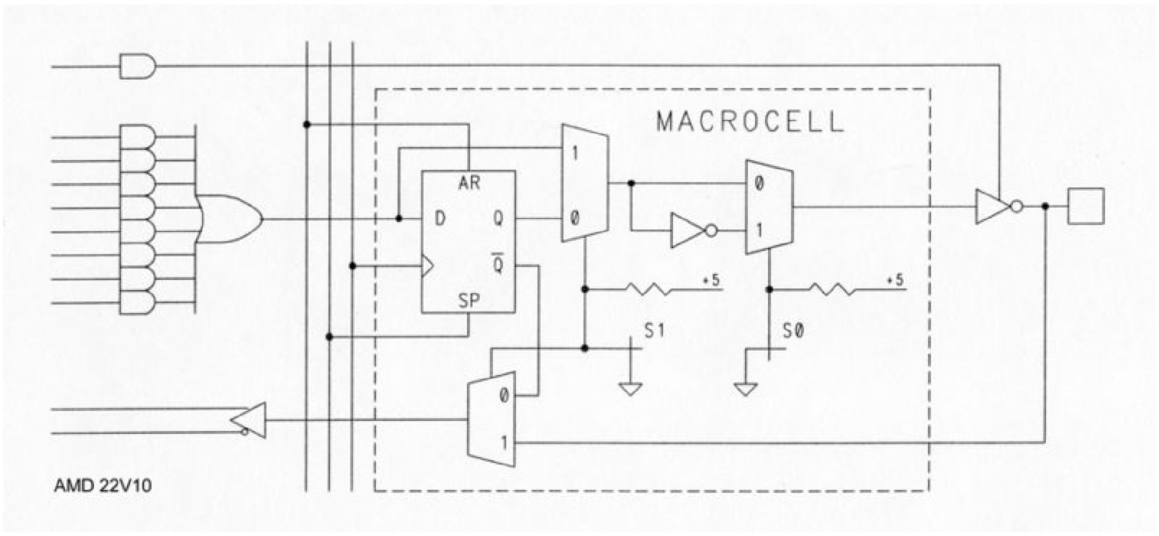
Programmable logic plane

The programmable logic plane is a programmable read-only memory (PROM) array that allows the signals present on the devices pins (or the logical complements of those signals) to be routed to an output logic macrocell.

PAL devices have arrays of transistor cells arranged in a "fixed-OR, programmable-AND" plane used to implement "sum-of-products" binary logic equations for each of the outputs in terms of the inputs and either synchronous or asynchronous feedback from the outputs.

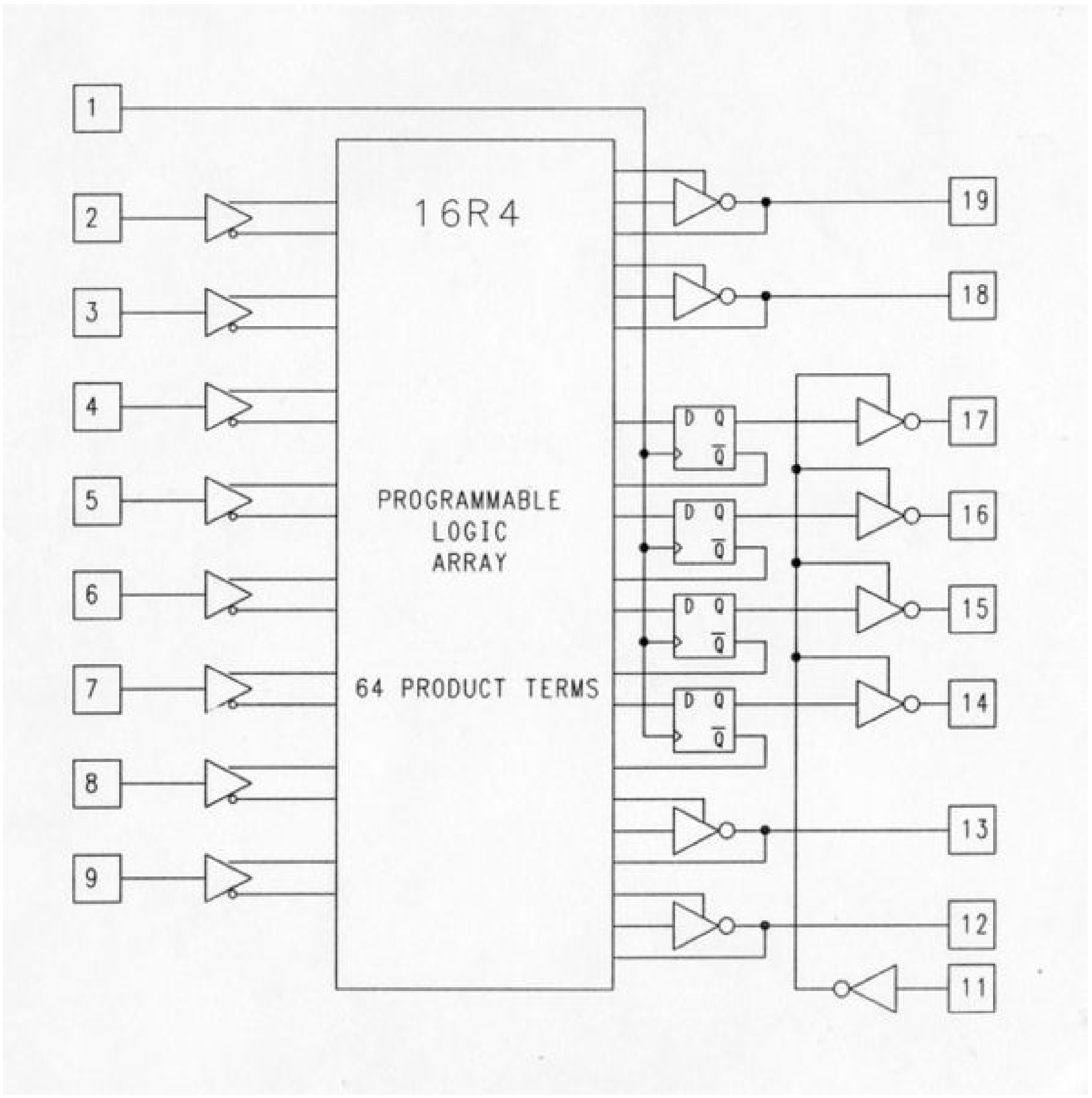
Output logic

The early 20-pin PALs had 10 inputs and 8 outputs. The outputs were active low and could be registered or combinational. Members of the PAL family were available with various output structures called "output logic macrocells" or OLMCs. Prior to the introduction of the "V" (for "variable") series, the types of OLMCs available in each PAL were fixed at the time of manufacture. (The PAL16L8 had 8 combinational outputs and the PAL16R8 had 8 registered outputs. The PAL16R6 had 6 registered and 2 combinational while the PAL16R4 had 4 of each.) Each output could have up to 8 product terms (effectively AND gates), however the combinational outputs used one of the terms to control a bidirectional output buffer. There were other combinations that had fewer outputs with more product term per output and were available with active high outputs. The 16X8 family or registered devices had an XOR gate before the register. There were also similar 24-pin versions of these PALs.

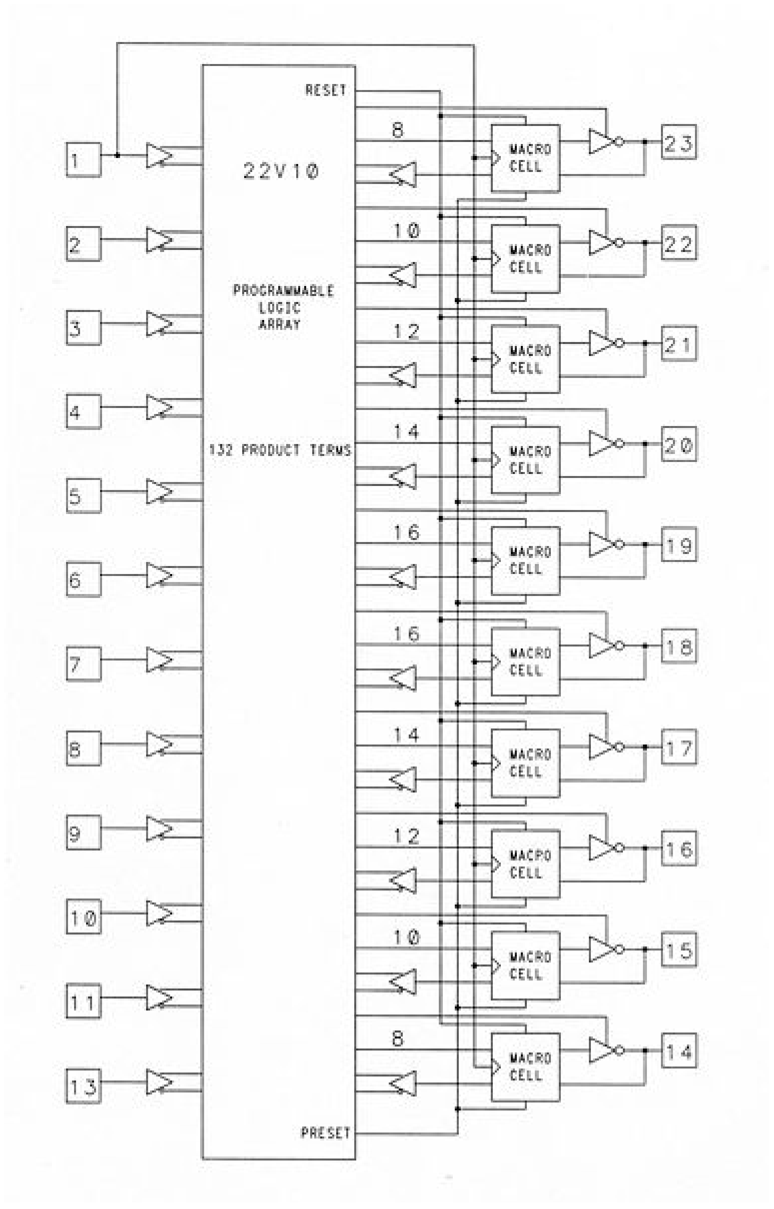


AMD 22V10 Output Macrocell

This fixed output structure often frustrated designers attempting to optimize the utility of PAL devices because output structures of different types were often required by their applications. (For example, one could not get 5 registered outputs with 3 active high combinational outputs.) So, in June 1983 AMD introduced the 22V10, a 24 pin device with 10 output logic macrocells. Each macrocell could be configured by the user to be combinational or registered, active high or active low. The number of product term allocated to an output varied from 8 to 16. This one device could replace all of the 24 pin fixed function PAL devices. Members of the PAL "V" ("variable") series included the PAL16V8, PAL20V8 and PAL22V10.



PAL 16R4 Block Diagram

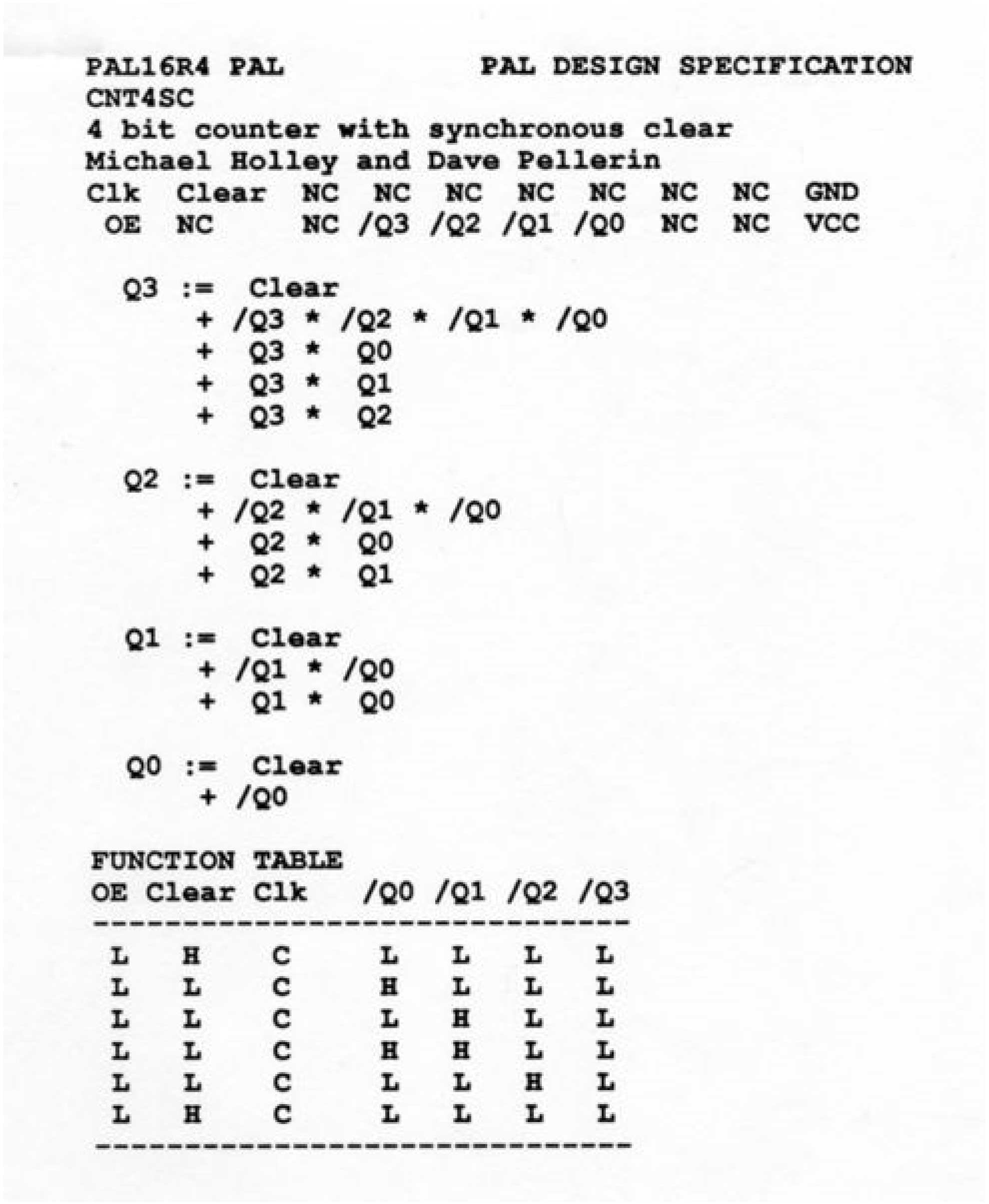


AMD 22V10 Block Diagram

Programming PALs

PALs were programmed electrically using binary patterns (as JEDEC ASCII/hexadecimal files) and a special electronic programming system available from either the manufacturer or a third-party, such as DATA/IO. In addition to single-unit device programmers, device feeders and gang programmers were often used when more than just a few PALs needed to be programmed. (For large volumes, electrical programming costs could be eliminated by having the manufacturer fabricate a custom metal mask used to program the customers' patterns at the time of manufacture; MMI used the term "hard array logic" (HAL) to refer to devices programmed in this way.)

Programming languages



```
PAL16R4 PAL          PAL DESIGN SPECIFICATION
CNT4SC
4 bit counter with synchronous clear
Michael Holley and Dave Pellerin
Clk  Clear  NC  NC  NC  NC  NC  NC  NC  GND
 OE  NC     NC /Q3 /Q2 /Q1 /Q0  NC  NC  VCC

  Q3 :=  Clear
      + /Q3 * /Q2 * /Q1 * /Q0
      +  Q3 *  Q0
      +  Q3 *  Q1
      +  Q3 *  Q2

  Q2 :=  Clear
      + /Q2 * /Q1 * /Q0
      +  Q2 *  Q0
      +  Q2 *  Q1

  Q1 :=  Clear
      + /Q1 * /Q0
      +  Q1 *  Q0

  Q0 :=  Clear
      + /Q0

FUNCTION TABLE
OE Clear Clk   /Q0 /Q1 /Q2 /Q3
------------------------------
 L   H    C     L   L   L   L
 L   L    C     H   L   L   L
 L   L    C     L   H   L   L
 L   L    C     H   H   L   L
 L   L    C     L   L   H   L
 L   H    C     L   L   L   L
------------------------------
```

PALASM design of a 4-bit counter.

Though some engineers programmed PAL devices by manually editing files containing the binary fuse pattern data, most opted to design their logic using a hardware description language (HDL) such as Data I/O's ABEL, Logical Devices' CUPL, or MMI's PALASM. These were computer-assisted design (CAD) (now referred to as "electronic design automation") programs which translated (or "compiled") the designers' logic equations into binary fuse map files used to program (and often test) each device.

PALASM

The PALASM (from "PAL assembler") language was used to express boolean equations for the outputs pins in a text file which was then converted to the 'fuse map' file for the programming system using a vendor-supplied program; later the option of translation from schematics became common, and later still, 'fuse maps' could be 'synthesized' from an HDL (hardware description language,) such as Verilog.

The PALASM compiler was written by MMI in FORTRAN IV on an IBM 370/168. MMI made the source code available to users at no cost. By 1983, MMI customers ran versions on the DEC PDP-11, Data General NOVA, Hewlett-Packard HP2100, MDS800 and others.

ABEL

Data I/O Corporation released ABEL.

CUPL

Logical Devices, Inc. released the Universal Compiler for Programmable Logic (CUPL), which ran under MSDOS on the IBM PC and is currently available as an integrated development package for Microsoft Windows.

Device programmers

Popular device programmers included Data I/O Corporation's Model 60A Logic Programmer and Model 2900.

One of the very first PAL Programmers was the Structured Design "SD-20". They had the PALASM software built-in and only required a CRT terminal to enter the equations and view the fuse plots. After fusing, the outputs of the PAL could be verified if test vectors were entered in the source file.

Successors

After MMI succeeded with the 20-pin PAL parts introduced circa 1978, AMD introduced the 24-pin 22V10 PAL with additional features. After buying out MMI (circa 1987), AMD spun off a consolidated operation as Vantis, and that business was acquired by Lattice Semiconductor in 1989.

Altera introduced the EP300 (first CMOS PAL) in 1983 and later moved into the FPGA business.

Lattice Semiconductor introduced the generic array logic (GAL) family in 1985, with functional equivalents of the "V" series PALs that used reprogrammable logic planes based on EEPROM (electrically eraseable programmable read-only memory) technology.

National Semiconductor was a "second source" of GAL parts. AMD introduced a similar family called PALCE. In general one GAL part is able to function as any of the similar family PAL devices. For example the 16V8 GAL is able to replace the 16L8, 16H8, 16H6, 16H4, 16H2 and 16R8 PALs (and many others besides).

ICT (International CMOS Technology) introduced the PEEL 18CV8 in 1986. The 20-pin CMOS EEPROM part could be used in place of any of the registered-output bipolar PALs and used much less power.

Larger-scale programmable logic devices were introduced by Atmel, Lattice Semiconductor, and others. These devices extended the PAL architecture by including multiple logic planes and/or burying logic macrocells within the logic plane(s). The term "complex programmable logic device" (CPLD) was introduced to differentiate these devices from their PAL and GAL predecessors, which were then sometimes referred to as "simple programmable logic devices" or SPLDs.

Another large programmable logic device is the "field-programmable gate array" or FPGA. This term is often used to describe devices currently made by Altera and Xilinx.

Chapter 4

Programmable Logic Device

A **programmable logic device** or PLD is an electronic component used to build reconfigurable digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed, that is, reconfigured.

Using a ROM as a PLD

Before PLDs were invented, read-only memory (ROM) chips were used to create arbitrary combinational logic functions of a number of inputs. Consider a ROM with *m* inputs (the address lines) and *n* outputs (the data lines). When used as a memory, the ROM contains 2^m words of *n* bits each. Now imagine that the inputs are driven not by an *m*-bit address, but by *m* independent logic signals. Theoretically, there are 2^m possible Boolean functions of these *m* signals, but the structure of the ROM allows just 2^n of these functions to be produced at the output pins. The ROM therefore becomes equivalent to *n* separate logic circuits, each of which generates a chosen function of the *m* inputs.

The advantage of using a ROM in this way is that any conceivable function of the *m* inputs can be made to appear at any of the *n* outputs, making this the most general-purpose combinational logic device available. Also, PROMs (programmable ROMs), EPROMs (ultraviolet-erasable PROMs) and EEPROMs (electrically erasable PROMs) are available that can be programmed using a standard PROM programmer without requiring specialised hardware or software. However, there are several disadvantages:

- they are usually much slower than dedicated logic circuits,
- they cannot necessarily provide safe "covers" for asynchronous logic transitions so the PROM's outputs may glitch as the inputs switch,
- they consume more power,
- they are often more expensive than programmable logic, especially if high speed is required.

Since most ROMs do not have input or output registers, they cannot be used stand-alone for sequential logic. An external TTL register was often used for sequential designs such as state machines. Common EPROMs, for example the 2716, are still sometimes used in this way by hobby circuit designers, who often have some lying around. This use is sometimes called a 'poor man's PAL'.

Early programmable logic

In 1969, Motorola offered the XC157, a mask-programmed gate array with 12 gates and 30 uncommitted input/output pins.

In 1970, Texas Instruments developed a mask-programmable IC based on the IBM read-only associative memory or ROAM. This device, the TMS2000, was programmed by altering the metal layer during the production of the IC. The TMS2000 had up to 17 inputs and 18 outputs with 8 JK flip flop for memory. TI coined the term Programmable Logic Array for this device.

In 1971, General Electric Company (GE) was developing a programmable logic device based on the new PROM technology. This experimental device improved on IBM's ROAM by allowing multilevel logic. Intel had just introduced the floating-gate UV erasable PROM so the researcher at GE incorporated that technology. The GE device was the first erasable PLD ever developed, predating the Altera EPLD by over a decade. GE obtained several early patents on programmable logic devices.

In 1973 National Semiconductor introduced a mask-programmable PLA device (DM7575) with 14 inputs and 8 outputs with no memory registers. This was more popular than the TI part but cost of making the metal mask limited its use. The device is significant because it was the basis for the field programmable logic array produced by Signetics in 1975, the 82S100. (Intersil actually beat Signetics to market but poor yield doomed their part.)

In 1974 GE entered into an agreement with Monolithic Memories to develop a mask-programmable logic device incorporating the GE innovations. The device was named the 'Programmable Associative Logic Array' or PALA. The MMI 5760 was completed in 1976 and could implement multilevel or sequential circuits of over 100 gates. The device was supported by a GE design environment where Boolean equations would be converted to mask patterns for configuring the device. The part was never brought to market.

PAL

MMI introduced a breakthrough device in 1978, the Programmable Array Logic or PAL. The architecture was simpler than that of Signetics FPLA because it omitted the programmable OR array. This made the parts faster, smaller and cheaper. They were available in 20 pin 300 mil DIP packages while the FPLAs came in 28 pin 600 mil packages. The PAL Handbook demystified the design process. The PALASM design software (PAL Assembler) converted the engineers' Boolean equations into the fuse pattern required to program the part. The PAL devices were soon second-sourced by National Semiconductor, Texas Instruments and AMD.

After MMI succeeded with the 20-pin PAL parts, AMD introduced the 24-pin 22V10 PAL with additional features. After buying out MMI (1987), AMD spun off a consolidated operation as Vantis, and that business was acquired by Lattice Semiconductor in 1999.

There are also PLA's : Programmable Logic Array.

GALs

Lattice GAL 16V8 and 20V8

An innovation of the PAL was the **generic array logic** device, or **GAL**, invented by Lattice Semiconductor in 1985. This device has the same logical properties as the PAL but can be erased and reprogrammed. The GAL is very useful in the prototyping stage of a design, when any bugs in the logic can be corrected by reprogramming. GALs are programmed and reprogrammed using a PAL programmer, or by using the in-circuit programming technique on supporting chips.

Lattice GALs combine CMOS and electrically erasable (E^2) floating gate technology for a high-speed, low-power logic device.

A similar device called a **PEEL** (**programmable electrically erasable logic**) was introduced by the International CMOS Technology (ICT) corporation.

CPLDs

PALs and GALs are available only in small sizes, equivalent to a few hundred logic gates. For bigger logic circuits, complex PLDs or CPLDs can be used. These contain the equivalent of several PALs linked by programmable interconnections, all in one integrated circuit. CPLDs can replace thousands, or even hundreds of thousands, of logic gates.

Some CPLDs are programmed using a PAL programmer, but this method becomes inconvenient for devices with hundreds of pins. A second method of programming is to solder the device to its printed circuit board, then feed it with a serial data stream from a personal computer. The CPLD contains a circuit that decodes the data stream and configures the CPLD to perform its specified logic function.

Each manufacturer has a proprietary name for this programming system. For example, Lattice Semiconductor calls it "in-system programming". However, these proprietary systems are beginning to give way to a standard from the Joint Test Action Group, JTAG.

FPGAs

While PALs were busy developing into GALs and CPLDs (all discussed above), a separate stream of development was happening. This type of device is based on gate array technology and is called the field-programmable gate array (FPGA). Early examples of FPGAs are the 82s100 array, and 82S105 sequencer, by Signetics, introduced in the late 1970s. The 82S100 was an array of AND terms. The 82S105 also had flip flop functions.

FPGAs use a grid of logic gates, and once stored, the data doesn't change, similar to that of an ordinary gate array. The term "field-programmable" means the device is programmed by the customer, not the manufacturer.

FPGAs are usually programmed after being soldered down to the circuit board, in a manner similar to that of larger CPLDs. In most larger FPGAs the configuration is volatile, and must be re-loaded into the device whenever power is applied or different functionality is required. Configuration is typically stored in a configuration PROM or EEPROM. EEPROM versions may be in-system programmable (typically via JTAG).

The difference between FPGAs and CPLDs is that FPGAs are internally based on Look-up tables (LUTs) whereas CPLDs form the logic functions with sea-of-gates (e.g. sum of products). CPLDs are meant for simpler designs while FPGAs are meant for more complex designs. In general, CPLDs are a good choice for wide combinational logic applications, whereas FPGAs are more suitable for large state machines (i.e. microprocessors).

Other variants

At present, much interest exists in reconfigurable systems. These are microprocessor circuits that contain some fixed functions and other functions that can be altered by code running on the processor. Designing self-altering systems requires engineers to learn new methods, and that new software tools be developed.

PLDs are being sold now that contain a microprocessor with a fixed function (the so-called *core*) surrounded by programmable logic. These devices let designers concentrate on adding new features to designs without having to worry about making the microprocessor work.

How PLDs retain their configuration

A PLD is a combination of a logic device and a memory device. The memory is used to store the pattern that was given to the chip during programming. Most of the methods for storing data in an integrated circuit have been adapted for use in PLDs. These include:

- Silicon antifuses
- SRAM
- EPROM or EEPROM cells
- Flash memory

Silicon antifuses are the storage elements used in the PAL, the first type of PLD. These are connections that are made by applying a voltage across a modified area of silicon inside the chip. They are called antifuses because they work in the opposite way to normal fuses, which begin life as connections until they are broken by an electric current.

SRAM, or static RAM, is a volatile type of memory, meaning that its contents are lost each time the power is switched off. SRAM-based PLDs therefore have to be programmed every time the circuit is switched on. This is usually done automatically by another part of the circuit.

An EPROM cell is a MOS (metal-oxide-semiconductor) transistor that can be switched on by trapping an electric charge permanently on its gate electrode. This is done by a PAL programmer. The charge remains for many years and can only be removed by exposing the chip to strong ultraviolet light in a device called an EPROM eraser.

Flash memory is non-volatile, retaining its contents even when the power is switched off. It can be erased and reprogrammed as required. This makes it useful for PLD memory.

As of 2005, most CPLDs are electrically programmable and erasable, and non-volatile. This is because they are too small to justify the inconvenience of programming internal SRAM cells every time they start up, and EPROM cells are more expensive due to their ceramic package with a quartz window.

PLD programming languages

Many PAL programming devices accept input in a standard file format, commonly referred to as 'JEDEC files'.They are analogous to software compilers. The languages used as source code for logic compilers are called hardware description languages, or HDLs.

PALASM and ABEL are frequently used for low-complexity devices, while Verilog and VHDL are popular higher-level description languages for more complex devices. The more limited ABEL is often used for historical reasons, but for new designs VHDL is more popular, even for low-complexity designs.

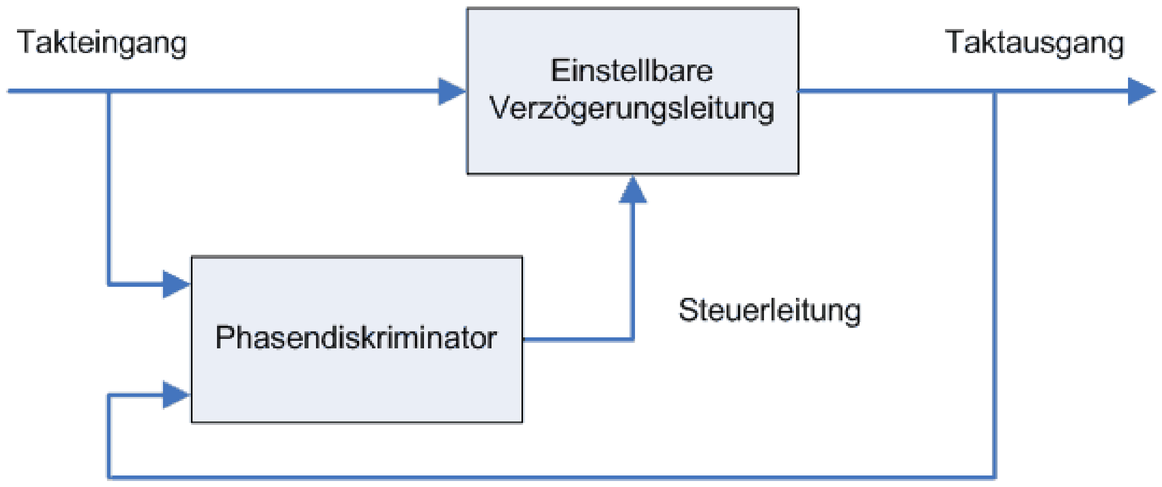
PLD programming devices

A device programmer is used to transfer the boolean logic pattern into the programmable device. In the early days of programmable logic, every PLD manufacturer also produced a specialized device programmer for its family of logic devices. Later, universal device programmers came onto the market that supported several logic device families from different manufacturers. Today's device programmers usually can program common PLDs (mostly PAL/GAL equivalents) from all existing manufacturers. Common file formats used to store the boolean logic pattern (fuses) are JEDEC, Altera POF (Programmable Object File), or Xilinx BITstream.

Chapter 5

Delay-Locked Loop & Complex Programmable Logic Device

Delay-Locked Loop



In electronics, a **delay-locked loop** (DLL) is a digital circuit similar to a phase-locked loop (PLL), with the main difference being the absence of an internal voltage-controlled oscillator. A DLL can be used to change the phase of a clock signal (a signal with a periodic waveform), usually to enhance the *clock rise*-to-*data output valid* timing characteristics of integrated circuits (such as DRAM devices). DLLs can also be used for clock recovery (CDR). From the outside, a DLL can be seen as a negative-delay gate placed in the clock path of a digital circuit.

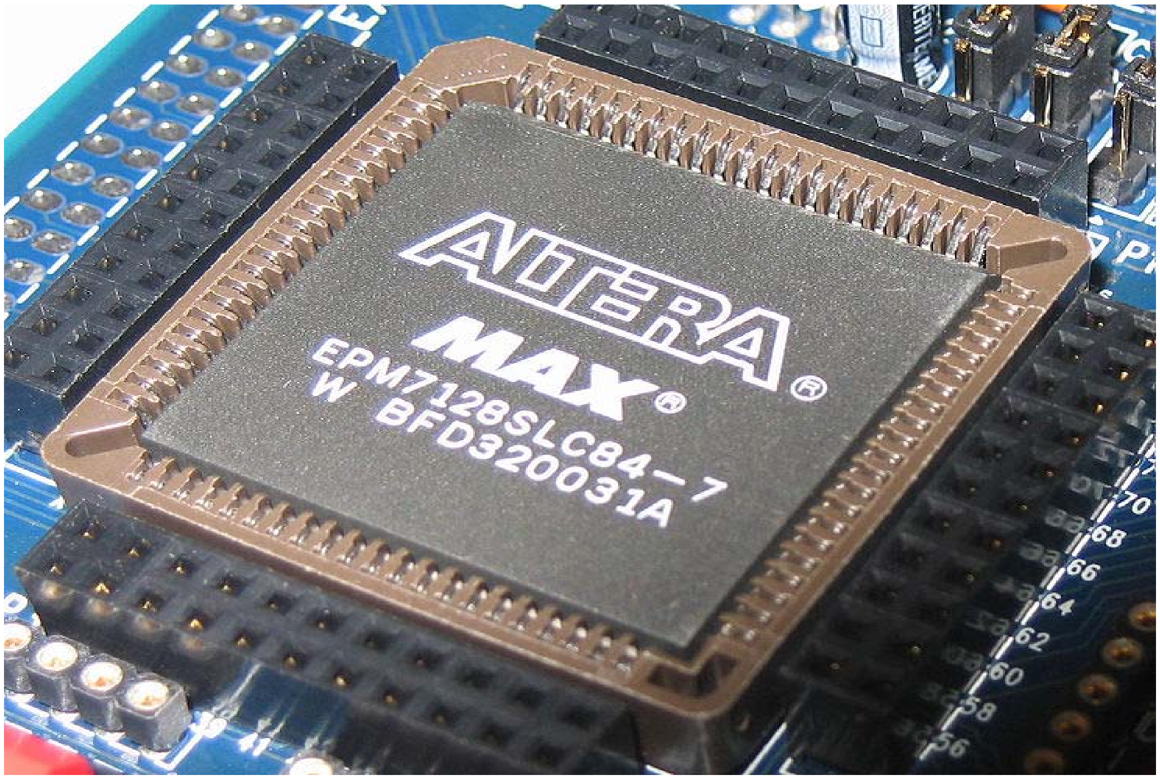
Another way to view the difference between a DLL and a PLL is that a DLL is a first order loop and a PLL is a second order loop. A DLL compares the phase of one of its outputs to the input clock to generate an error signal which is then integrated and fed back as the control to all of the delay elements. The integration allows the error to go to zero while keeping the control signal, and thus the delays, where they need to be for phase lock. Since the control signal directly impacts the phase this is all that is required. A PLL compares the phase of its oscillator with the incoming signal to generate an error signal which is then integrated to create a control signal for the voltage-controlled oscillator. The control signal impacts the frequency of the oscillator, and phase is the integral of frequency, so a second integration is unavoidably performed by the oscillator itself. A first order feedback system is significantly easier to stabilize than a second order feedback system, which is a major advantage of DLLs.

The main component of a DLL is a delay chain composed of many delay gates connected front-to-back. The input of the chain (and thus of the DLL) is connected to the clock that is to be negatively delayed. A multiplexer is connected to each stage of the delay chain; the selector of this multiplexer is automatically updated by a control circuit to produce the negative delay effect. The output of the DLL is the resulting, negatively delayed clock signal.

The phase shift can be specified either in absolute terms (in delay chain gate units), or as a proportion of the clock period, or both.

Compared to phase-locked loops, delay-locked loops are a relatively recent innovation, first found in Dr. Combes' work in the early 1990s, then popularized by Xilinx in their Virtex family of FPGA products.

Complex Programmable Logic Device



An Altera MAX 7000-series CPLD with 2500 gates.

A **complex programmable logic device (CPLD)** is a programmable logic device with complexity between that of PALs and FPGAs, and architectural features of both. The building block of a CPLD is the macrocell, which contains logic implementing disjunctive normal form expressions and more specialized logic operations.

Features in common with PALs:

- Non-volatile configuration memory. Unlike many FPGAs, an external configuration ROM isn't required, and the CPLD can function immediately on system start-up.
- For many legacy CPLD devices, routing constrains most logic blocks to have input and output signals connected to external pins, reducing opportunities for internal state storage and deeply layered logic. This is usually not a factor for larger CPLDs and newer CPLD product families.

Features in common with FPGAs:

- Large number of gates available. CPLDs typically have the equivalent of thousands to tens of thousands of logic gates, allowing implementation of moderately complicated data processing devices. PALs typically have a few hundred gate equivalents at most, while FPGAs typically range from tens of thousands to several million.
- Some provisions for logic more flexible than sum-of-product expressions, including complicated feedback paths between macro cells, and specialized logic for implementing various commonly-used functions, such as integer arithmetic.

The most noticeable difference between a large CPLD and a small FPGA is the presence of on-chip non-volatile memory in the CPLD. This distinction is rapidly becoming less relevant, as several of the latest FPGA products also offer models with embedded configuration memory.

The characteristic of non-volatility makes the CPLD the device of choice in modern digital designs to perform 'boot loader' functions before handing over control to other devices not having this capability. A good example is where a CPLD is used to load configuration data for an FPGA from non-volatile memory.

CPLDs were an evolutionary step from even smaller devices that preceded them, PLAs (first shipped by Signetics), and PALs. These in turn were preceded by standard logic products, that offered no programmability and were "programmed" by wiring several standard logic chips together.

The main distinction between FPGA and CPLD device architectures is that FPGAs are internally based on Look-up tables (LUTs) while CPLDs form the logic functions with sea-of-gates (e.g. sum of products).

Chapter 6

Partial Re-Configuration & Rent's Rule

Partial Re-Configuration

Partial Reconfiguration is the process of configuring a portion of a field programmable gate array while the other part is still running/operating.

Hardware, like software, can be designed modularly, by creating subcomponents and then higher-level components to instantiate them. In many cases it is useful to be able to swap out one or several of these subcomponents while the FPGA is still operating.

Normally, reconfiguring an FPGA requires it to be held in reset while an external controller reloads a design onto it. Partial reconfiguration allows for critical parts of the design to continue operating while a controller either on the FPGA or off of it loads a partial design into a reconfigurable module. Partial reconfiguration also can be used to save space for multiple designs by only storing the partial designs that change between designs.

A common example for when partial reconfiguration would be useful is the case of a communication device. If the device is controlling multiple connections, some of which require encryption, it would be useful to be able to load different encryption cores without bringing the whole controller down.

Partial reconfiguration is not supported on all FPGAs. A special software flow with emphasis on modular design is required. Typically the design modules are built along well defined boundaries inside the FPGA that require the design to be specially mapped to the internal hardware.

From the functionality of the design, partial reconfiguration can be divided into two groups:

- **dynamic partial reconfiguration**, also known as an **active** partial reconfiguration - permits to change the part of the device while the rest of an FPGA is still running;
- **static partial reconfiguration** - the device is not active during the reconfiguration process. While the partial data is sent into the FPGA, the rest of

the device is stopped (in the shutdown mode) and brought up after the configuration is completed.

There are two styles of partial reconfiguration of FPGA devices from Xilinx: **module-based** and **difference-based**.

Module-based partial reconfiguration permits to reconfigure distinct modular parts of the design. To ensure the communication across the reconfigurable module boundaries, special bus macros ought to be prepared. It works as a fixed routing bridge that connects the reconfigurable module with the rest part of the design. Module-based partial reconfiguration requires to perform a set of specific guidelines during at the stage of design specification. Finally for each reconfigurable module of the design, separate bit-stream is created. Such a bit-stream is used to perform the partial reconfiguration of an FPGA.

Difference-based partial reconfiguration can be used when a small change is made to the design. It is especially useful in case of changing LUT equations or dedicated memory blocks content. The partial bit-stream contains only information about differences between the current design structure (that resides in the FPGA) and the new content of an FPGA. There are two ways of difference-based reconfiguration known as a front-end and back-end. The first one is based on the modification of the design in the hardware description languages (HDLs). It is clear that such a solution requires full repeating of the synthesis and implementation processes. The back-end difference-based partial reconfiguration permits to make changes at the implementation stage of the prototyping flow. Therefore there is no need for re-synthesis of the design. The usage of both methods (either front-end and back-end) leads to creation of a partial bit-stream that can be used for a partial reconfiguration of the FPGA.

Rent's Rule

Rent's rule pertains to the organization of computing logic, specifically the relationship between the number of external signal connections to a logic block (i.e., the number of "pins") with the number of logic gates in the logic block, and has been applied to circuits ranging from small digital circuits to mainframe computers.

E.F. Rent's discovery and first publications

In the 1960's, E.F. Rent, an IBM employee, found a remarkable trend between the number of pins (terminals T) at the boundaries of integrated circuit designs at IBM and the number of internal components (g), such as logic gates or standard cells. On a log − log plot, these datapoints were on a straight line, implying a power-law relation $T = tg^p$ where t and p are constants ($p < 1.0$, and generally $0.5 < p < 0.8$). Rent disclosed his findings in IBM-internal memoranda, but the relation was described in 1971 by Landman and Russo. They performed a hierarchical circuit partitioning in such a way that at each hierarchical level (top-down) the least number of interconnections had to be cut to partition the circuit (in more or less equal parts). At each partitioning step, they noted the number of terminals and the number of components in each partition and then partitioned the sub-partitions further. They found the power law rule applied to the resulting T versus g plot and named it "Rent's rule". It is crucial to recognise that Rent's rule is an empirical result based on observations of existing designs, and therefore it is less applicable to the analysis of non-traditional circuit architectures. Having said that, it does provide a useful framework with which to compare similar architectures.

Theoretical basis

Christie and Stroobandt later derived Rent's rule theoretically for homogeneous systems and pointed out that the amount of optimization achieved in placement is reflected by the parameter p, the "Rent exponent", which also depends on the circuit topology. In particular, values $p < 1$ correspond to a greater fraction of short interconnects. The constant t in Rent's rule can be viewed as the average number of terminals required by a single logic block since $T = t$ when $g = 1$.

Special cases and applications

Random arrangement of logic blocks typically have $p = 1$. Larger values are impossible since the maximum number of terminals for any region containing g logic components in a homogeneous system is given by $T = tg$. Lower bounds on p depend on the interconnection topology since it is generally impossible to make all wires short. This lower bound p * is often called the "intrinsic Rent exponent", a notion first introduced by Hagen et al. It can be used to characterize optimal placements and also measure the interconnection complexity of a circuit. Higher (intrinsic) Rent exponent values correspond to a higher topological complexity. One extreme example ($p = 0$) is a long chain of logic blocks, while a clique has $p = 1$. In realistic 2D circuits, p * ranges from 0.5 for highly-regular circuits (such as SRAM) to 0.75 for random logic.

System performance analysis tools such as BACPAC typically use Rent's rule to calculate expected wiring lengths and wiring demands.

Estimating Rent's exponent

To estimate Rent's exponent, one can use top-down partitioning, as used in min-cut placement. For every partition, count the number of terminals connected to the partition and compare it to the number of logic blocks in the partition. Rent's exponent can then be found by fitting these datapoints on a log-log plot, resulting in an exponent p'. For optimally partitioned circuits, $p' = p$ * but this is no longer the case for practical (heuristic) partitioning approaches. For partitioning-based placement algorithms $p^* \leq p' \leq p$.

Region II of Rent's rule

Landman and Russo found a deviation of Rent's rule near the "far end", i.e., for partitions with a large number of blocks, which is known as "Region II" of Rent's Rule . A similar deviation exists at for small partitions, and has been found by Stroobandt who called it Region III.

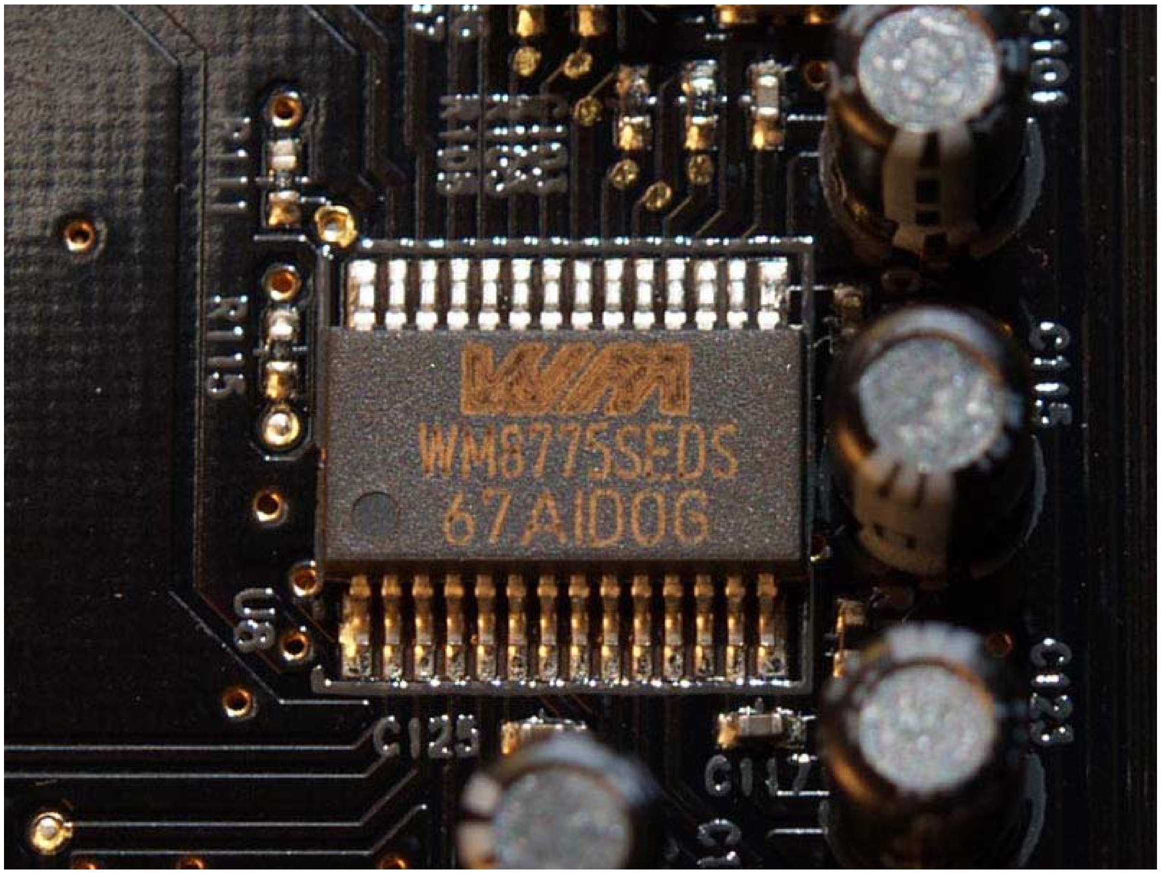
Rentian wirelength estimation

Another IBM employee, Donath, discovered that Rent's rule can be used to estimate the average wirelength and the wirelength distribution in VLSI chips. This motivated the System Level Interconnect Prediction workshop, founded in 1999, and an entire community working on wirelength prediction. The resulting wirelength estimates have been improved significantly since then and are now used for "technology exploration." The use of Rent's rule allows to perform such estimates *a priori* (i.e., before actual placement) and thus predict the properties of future technologies (clock frequencies, number of routing layers needed, area, power) based on limited information about future circuits and technologies.

A comprehensive overview of work based on Rent's rule has been published by Stroobandt.

Chapter 7

Analog-to-Digital Converter



4-channel stereo multiplexed analog-to-digital converter WM8775SEDS made by Wolfson Microelectronics placed on a X-Fi Fatal1ty Pro sound card.

An **analog-to-digital converter** (abbreviated **ADC**, **A/D** or **A to D**) is a device which converts a continuous quantity to a discrete time digital representation. An ADC may also provide an isolated measurement. The reverse operation is performed by a digital-to-analog converter (**DAC**).

Typically, an ADC is an electronic device that converts an input analog voltage or current to a digital number proportional to the magnitude of the voltage or current. However, some non-electronic or only partially electronic devices, such as rotary encoders, can also be considered ADCs.

The digital output may use different coding schemes. Typically the digital output will be a two's complement binary number that is proportional to the input, but there are other possibilities. An encoder, for example, might output a Gray code.

Concepts

Resolution

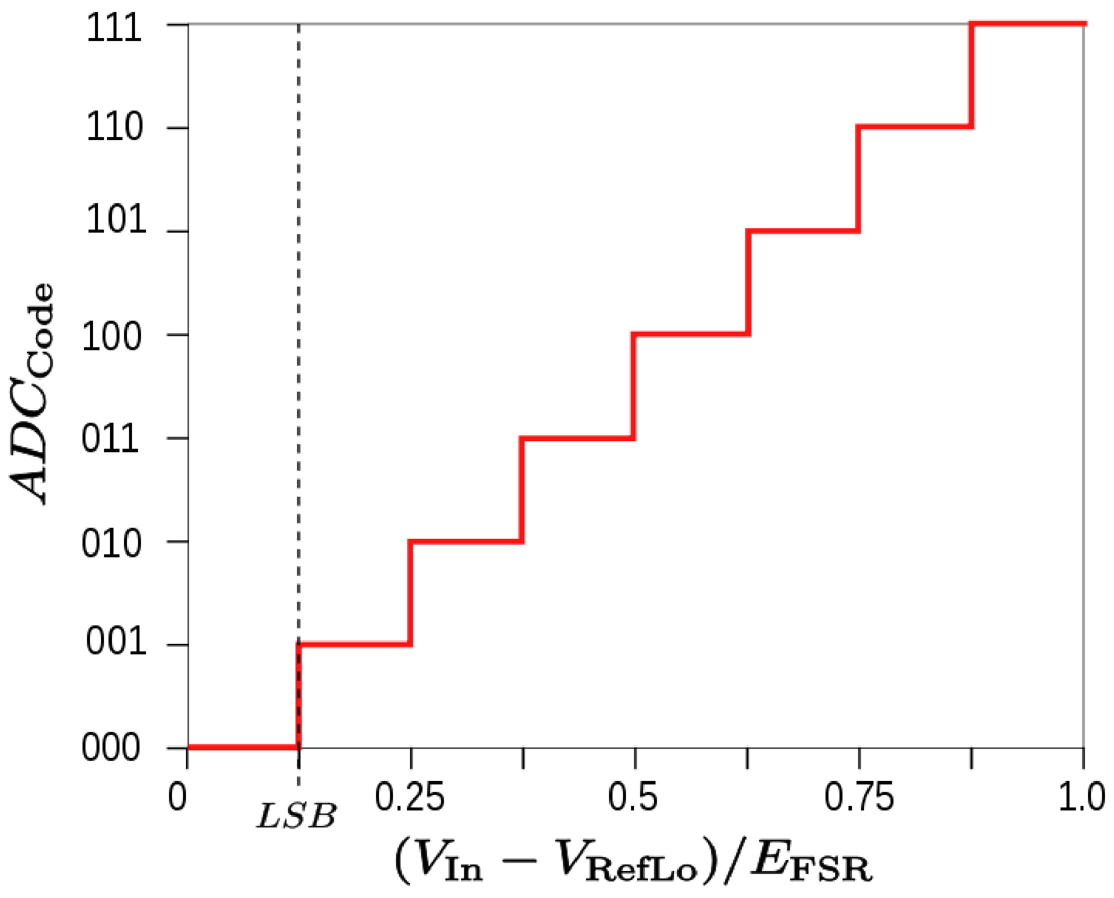


Fig. 1. An 8-level ADC coding scheme.

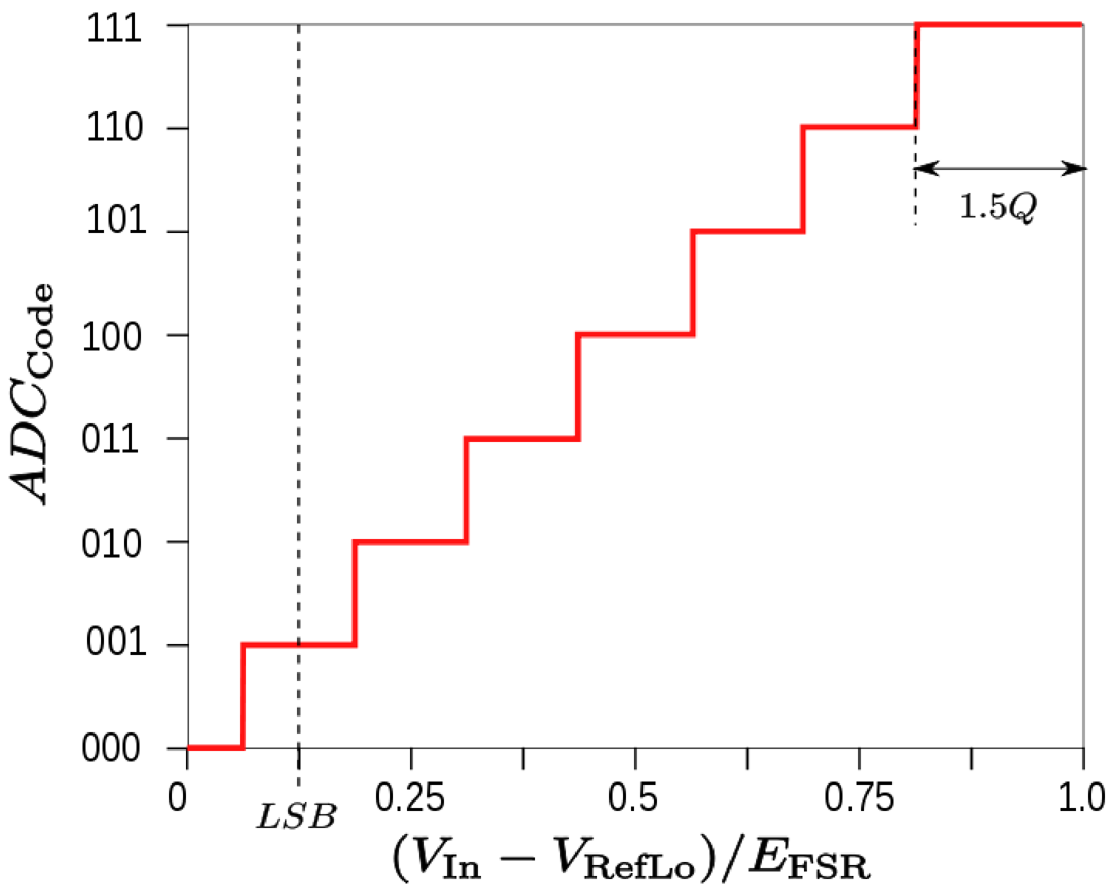


Fig. 2. An 8-level ADC coding scheme. As in figure 1 but with mid-tread coding.

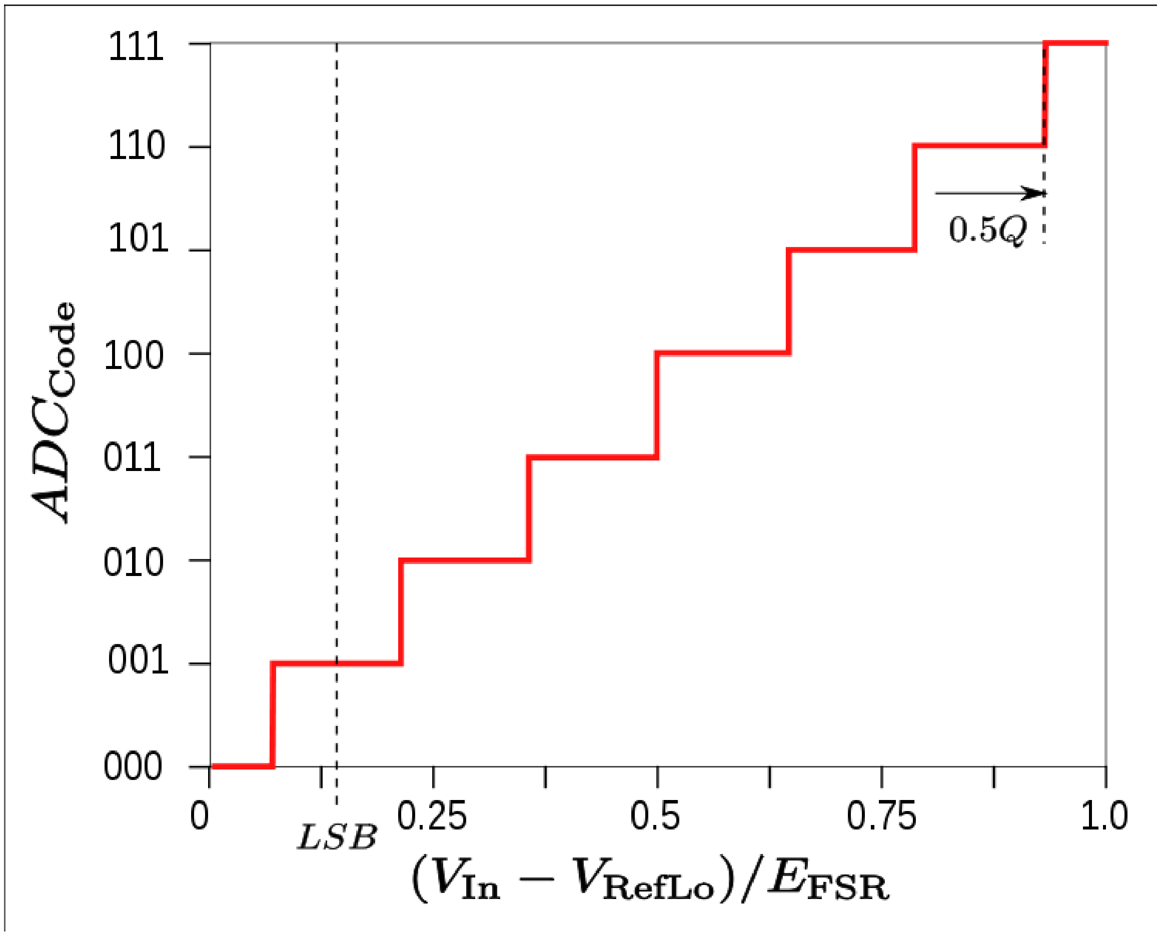


Fig. 3. An 8-level ADC mid-tread coding scheme. As in figure 2 but with equal half-*LSB* intervals at the highest and lowest codes. Note that *LSB* is now slightly larger than in figures 1 and 2.

The resolution of the converter indicates the number of discrete values it can produce over the range of analog values. The values are usually stored electronically in binary form, so the resolution is usually expressed in bits. In consequence, the number of discrete values available, or "levels", is usually a power of two. For example, an ADC with a resolution of 8 bits can encode an analog input to one in 256 different levels, since $2^8 = 256$. The values can represent the ranges from 0 to 255 (i.e. unsigned integer) or from −128 to 127 (i.e. signed integer), depending on the application.

Resolution can also be defined electrically, and expressed in volts. The minimum change in voltage required to guarantee a change in the output code level is called the *LSB* (least significant bit, since this is the voltage represented by a change in the LSB). The resolution *Q* of the ADC is equal to the *LSB* voltage. The voltage resolution of an ADC is equal to its overall voltage measurement range divided by the number of discrete voltage intervals:

$$Q = \frac{E_{\mathrm{FSR}}}{N},$$

where N is the number of voltage intervals and E_{FSR} is the full scale voltage range. E_{FSR} is given by

$$E_{\mathrm{FSR}} = V_{\mathrm{RefHi}} - V_{\mathrm{RefLow}},$$

where V_{RefHi} and V_{RefLow} are the upper and lower extremes, respectively, of the voltages that can be coded.

Normally, the number of voltage intervals is given by

$$N = 2^M,$$

where M is the ADC's resolution in bits.

That is, one voltage interval is assigned per code level. However, figure 3 shows a situation where

$$N = 2^M - 1$$

Some examples:

- Example 1
  - Coding scheme as in figure 1
  - Full scale measurement range = 0 to 10 volts
  - ADC resolution is 12 bits: 2^{12} = 4096 quantization levels (codes)
  - ADC voltage resolution, Q = (10 V − 0 V) / 4096 = 10 V / 4096 ≈ 0.00244 V ≈ 2.44 mV.

- Example 2
  - Coding scheme as in figure 2
  - Full scale measurement range = -10 to +10 volts
  - ADC resolution is 14 bits: 2^{14} = 16384 quantization levels (codes)
  - ADC voltage resolution is, Q = (10 V − (−10 V)) / 16384 = 20 V / 16384 ≈ 0.00122 V ≈ 1.22 mV.

- Example 3
  - Coding scheme as in figure 3
  - Full scale measurement range = 0 to 7 volts
  - ADC resolution is 3 bits: 2^3 = 8 quantization levels (codes)
  - ADC voltage resolution is, Q = (7 V − 0 V)/7 = 7 V/7 = 1 V = 1000 mV

In most ADCs, the smallest output code ("0" in an unsigned system) represents a voltage range which is 0.5Q, that is, half the ADC voltage resolution (Q). The largest code represents a range of 1.5Q as in figure 2 (if this were 0.5Q also, the result would be as figure 3). The other $N - 2$ codes are all equal in width and represent the ADC voltage resolution (Q) calculated above. Doing this centers the code on an input voltage that represents the M th division of the input voltage range. This practice is called "mid-tread" operation. This type of ADC can be modeled mathematically as:

$$ADC_{\text{Code}} = \text{round}\left(\left(\frac{2^M}{V_{\text{RefHi}} - V_{\text{RefLow}}}\right) \cdot (V_{\text{In}} - V_{\text{RefLow}})\right)$$

The exception to this convention seems to be the Microchip PIC processor, where all M steps are equal width, as shown in figure 1. This practice is called "Mid-Rise with Offset" operation.

$$ADC_{\text{Code}} = \text{floor}\left(\left(\frac{2^M}{V_{\text{RefHi}} - V_{\text{RefLow}}}\right) \cdot (V_{\text{In}} - V_{\text{RefLow}})\right)$$

In practice, the useful resolution of a converter is limited by the best signal-to-noise ratio (SNR) that can be achieved for a digitized signal. An ADC can resolve a signal to only a certain number of bits of resolution, called the effective number of bits (ENOB). One effective bit of resolution changes the signal-to-noise ratio of the digitized signal by 6 dB, if the resolution is limited by the ADC. If a preamplifier has been used prior to A/D conversion, the noise introduced by the amplifier can be an important contributing factor towards the overall SNR.

Response type

Linear ADCs

Most ADCs are of a type known as linear The term *linear* implies here the range of the input values that map to each output value has a linear relationship with the output value, i.e., that the output value k is used for the range of input values from

$$m(k + b)$$

to

$$m(k + 1 + b),$$

where m and b are constants. Here b is typically 0 or −0.5. When $b = 0$, the ADC is referred to as *mid-rise*, and when $b = -0.5$ it is referred to as *mid-tread*.

Non-linear ADCs

If the probability density function of a signal being digitized is uniform, then the signal-to-noise ratio relative to the quantization noise is the best possible. Because this is often not the case, it is usual to pass the signal through its cumulative distribution function (CDF) before the quantization. This is good because the regions that are more important get quantized with a better resolution. In the dequantization process, the inverse CDF is needed.

This is the same principle behind the companders used in some tape-recorders and other communication systems, and is related to entropy maximization.

For example, a voice signal has a Laplacian distribution. This means that the region around the lowest levels, near 0, carries more information than the regions with higher amplitudes. Because of this, logarithmic ADCs are very common in voice communication systems to increase the dynamic range of the representable values while retaining fine-granular fidelity in the low-amplitude region.

An eight-bit A-law or the μ-law logarithmic ADC covers the wide dynamic range and has a high resolution in the critical low-amplitude region, that would otherwise require a 12-bit linear ADC.

Accuracy

An ADC has several sources of errors. Quantization error and (assuming the ADC is intended to be linear) non-linearity are intrinsic to any analog-to-digital conversion. There is also a so-called *aperture error* which is due to a clock jitter and is revealed when digitizing a time-variant signal (not a constant value).

These errors are measured in a unit called the *LSB*, which is an abbreviation for least significant bit. In the above example of an eight-bit ADC, an error of one LSB is 1/256 of the full signal range, or about 0.4%.

Quantization error

Quantization error (or quantization noise) is the difference between the original signal and the digitized signal. Hence, The magnitude of the quantization error at the sampling instant is between zero and half of one LSB. Quantization error is due to the finite resolution of the digital representation of the signal, and is an unavoidable imperfection in all types of ADCs.

Non-linearity

All ADCs suffer from non-linearity errors caused by their physical imperfections, causing their output to deviate from a linear function (or some other function, in the case of a deliberately non-linear ADC) of their input. These errors can sometimes be mitigated by calibration, or prevented by testing.

Important parameters for linearity are integral non-linearity (INL) and differential non-linearity (DNL). These non-linearities reduce the dynamic range of the signals that can be digitized by the ADC, also reducing the effective resolution of the ADC.

Aperture error

Imagine that we are digitizing a sine wave $x(t) = A\sin(2\pi f_0 t)$. Provided that the actual sampling time *uncertainty* due to the *clock jitter* is Δt, the error caused by this phenomenon can be estimated as $E_{ap} \leq |x'(t)\Delta t| \leq 2A\pi f_0 \Delta t$.

The error is zero for DC, small at low frequencies, but significant when high frequencies have high amplitudes. This effect can be ignored if it is drowned out by the *quantizing error*. Jitter requirements can be calculated using the following formula: $\Delta t < \frac{1}{2^q \pi f_0}$, where q is a number of ADC bits.

ADC resolution in bit	**input frequency**						
	1 Hz	**44.1 kHz**	**192 kHz**	**1 MHz**	**10 MHz**	**100 MHz**	**1 GHz**
8	1243 µs	28.2 ns	6.48 ns	1.24 ns	124 ps	12.4 ps	1.24 ps
10	311 µs	7.05 ns	1.62 ns	311 ps	31.1 ps	3.11 ps	0.31 ps
12	77.7 µs	1.76 ns	405 ps	77.7 ps	7.77 ps	0.78 ps	0.08 ps
14	19.4 µs	441 ps	101 ps	19.4 ps	1.94 ps	0.19 ps	0.02 ps
16	4.86 µs	110 ps	25.3 ps	4.86 ps	0.49 ps	0.05 ps	–
18	1.21 µs	27.5 ps	6.32 ps	1.21 ps	0.12 ps	–	–
20	304 ns	6.88 ps	1.58 ps	0.16 ps	–	–	–
24	19.0 ns	0.43 ps	0.10 ps	–	–	–	–
32	74.1 ps	–	–	–	–	–	–

This table shows, for example, that it is not worth using a precise 24-bit ADC for sound recording if there is not an *ultra low jitter* clock. One should consider taking this phenomenon into account before choosing an ADC.

Clock jitter is caused by phase noise. The resolution of ADCs with a digitization bandwidth between 1 MHz and 1 GHz is limited by jitter.

When sampling audio signals at 44.1 kHz, the anti-aliasing filter should have eliminated all frequencies above 22 kHz. The input frequency (in this case, 22 kHz), not the ADC clock frequency, is the determining factor with respect to jitter performance.

Sampling rate

The analog signal is continuous in time and it is necessary to convert this to a flow of digital values. It is therefore required to define the rate at which new digital values are sampled from the analog signal. The rate of new values is called the *sampling rate* or *sampling frequency* of the converter.

A continuously varying bandlimited signal can be sampled (that is, the signal values at intervals of time T, the sampling time, are measured and stored) and then the original signal can be *exactly* reproduced from the discrete-time values by an interpolation formula. The accuracy is limited by quantization error. However, this faithful reproduction is only possible if the sampling rate is higher than twice the highest frequency of the signal. This is essentially what is embodied in the Shannon-Nyquist sampling theorem.

Since a practical ADC cannot make an instantaneous conversion, the input value must necessarily be held constant during the time that the converter performs a conversion (called the *conversion time*). An input circuit called a sample and hold performs this task—in most cases by using a capacitor to store the analog voltage at the input, and using an electronic switch or gate to disconnect the capacitor from the input. Many ADC integrated circuits include the sample and hold subsystem internally.

Aliasing

All ADCs work by sampling their input at discrete intervals of time. Their output is therefore an incomplete picture of the behaviour of the input. There is no way of knowing, by looking at the output, what the input was doing between one sampling instant and the next. If the input is known to be changing slowly compared to the sampling rate, then it can be assumed that the value of the signal between two sample instants was somewhere between the two sampled values. If, however, the input signal is changing rapidly compared to the sample rate, then this assumption is not valid.

If the digital values produced by the ADC are, at some later stage in the system, converted back to analog values by a digital to analog converter or DAC, it is desirable that the output of the DAC be a faithful representation of the original signal. If the input signal is changing much faster than the sample rate, then this will not be the case, and spurious signals called *aliases* will be produced at the output of the DAC. The frequency of the aliased signal is the difference between the signal frequency and the sampling rate. For example, a 2 kHz sine wave being sampled at 1.5 kHz would be reconstructed as a 500 Hz sine wave. This problem is called *aliasing*.

To avoid aliasing, the input to an ADC must be low-pass filtered to remove frequencies above half the sampling rate. This filter is called an *anti-aliasing* filter, and is essential for a practical ADC system that is applied to analog signals with higher frequency content.

Although aliasing in most systems is unwanted, it should also be noted that it can be exploited to provide simultaneous down-mixing of a band-limited high frequency signal.

Dither

In A-to-D converters, performance can usually be improved using dither. This is a very small amount of random noise (white noise) which is added to the input before conversion. Its amplitude is set to be twice the value of the least significant bit. Its effect is to cause the state of the LSB to randomly oscillate between 0 and 1 in the presence of very low levels of input, rather than sticking at a fixed value. Rather than the signal simply getting cut off altogether at this low level (which is only being quantized to a resolution of 1 bit), it extends the effective range of signals that the A-to-D converter can convert, at the expense of a slight increase in noise - effectively the quantization error is diffused across a series of noise values which is far less objectionable than a hard cutoff. The result is an accurate representation of the signal over time. A suitable filter at the output of the system can thus recover this small signal variation.

An audio signal of very low level (with respect to the bit depth of the ADC) sampled without dither sounds extremely distorted and unpleasant. Without dither the low level may cause the least significant bit to "stick" at 0 or 1. With dithering, the true level of the audio may be calculated by averaging the actual quantized sample with a series of other samples [the dither] that are recorded over time.

A virtually identical process, also called dither or dithering, is often used when quantizing photographic images to a fewer number of bits per pixel—the image becomes noisier but to the eye looks far more realistic than the quantized image, which otherwise becomes banded. This analogous process may help to visualize the effect of dither on an analogue audio signal that is converted to digital.

Dithering is also used in integrating systems such as electricity meters. Since the values are added together, the dithering produces results that are more exact than the LSB of the analog-to-digital converter.

Note that dither can only increase the resolution of a sampler, it cannot improve the linearity, and thus accuracy does not necessarily improve.

Oversampling

Usually, signals are sampled at the minimum rate required, for economy, with the result that the quantization noise introduced is white noise spread over the whole pass band of the converter. If a signal is sampled at a rate much higher than the Nyquist frequency and then digitally filtered to limit it to the signal bandwidth there are the following advantages:

- digital filters can have better properties (sharper rolloff, phase) than analogue filters, so a sharper anti-aliasing filter can be realised and then the signal can be downsampled giving a better result
- a 20-bit ADC can be made to act as a 24-bit ADC with 256× oversampling
- the signal-to-noise ratio due to quantization noise will be higher than if the whole available band had been used. With this technique, it is possible to obtain an effective resolution larger than that provided by the converter alone
- The improvement in SNR is 3 dB (equivalent to 0.5 bits) per octave of oversampling which is not sufficient for many applications. Therefore, oversampling is usually coupled with noise shaping. With noise shaping, the improvement is 6L+3 dB per octave where L is the order of loop filter used for noise shaping. e.g. - a 2nd order loop filter will provide an improvement of 15 dB/octave.

Relative speed and precision

The speed of an ADC varies by type. The Wilkinson ADC is limited by the clock rate which is processable by current digital circuits. Currently, frequencies up to 300 MHz are possible. The conversion time is directly proportional to the number of channels. For a successive approximation ADC, the conversion time scales with the logarithm of the number of channels. Thus for a large number of channels, it is possible that the successive approximation ADC is faster than the Wilkinson. However, the time consuming steps in the Wilkinson are digital, while those in the successive approximation are analog. Since analog is inherently slower than digital, as the number of channels increases, the time required also increases. Thus there are competing processes at work. Flash ADCs are certainly the fastest type of the three. The conversion is basically performed in a single parallel step. For an 8-bit unit, conversion takes place in a few tens of nanoseconds.

There is, as expected, somewhat of a trade off between speed and precision. Flash ADCs have drifts and uncertainties associated with the comparator levels, which lead to poor uniformity in channel width. Flash ADCs have a resulting poor linearity. For successive approximation ADCs, poor linearity is also apparent, but less so than for flash ADCs. Here, non-linearity arises from accumulating errors from the subtraction processes. Wilkinson ADCs are the best of the three. These have the best differential non-linearity. The other types require channel smoothing in order to achieve the level of the Wilkinson.

The sliding scale principle

The sliding scale or randomizing method can be employed to greatly improve the channel width uniformity and differential linearity of any type of ADC, but especially flash and successive approximation ADCs. Under normal conditions, a pulse of a particular amplitude is always converted to a certain channel number. The problem lies in that channels are not always of uniform width, and the differential linearity decreases proportionally with the divergence from the average width. The sliding scale principle uses an averaging effect to overcome this phenomenon. A random, but known analog voltage is added to the input pulse. It is then converted to digital form, and the equivalent digital version is subtracted, thus restoring it to its original value. The advantage is that the conversion has taken place at a random point. The statistical distribution of the final channel numbers is decided by a weighted average over a region of the range of the ADC. This in turn desensitizes it to the width of any given channel.

ADC structures

These are the most common ways of implementing an electronic ADC:

- A **direct conversion ADC** or **flash ADC** has a bank of comparators sampling the input signal in parallel, each firing for their decoded voltage range. The comparator bank feeds a logic circuit that generates a code for each voltage range. Direct conversion is very fast, capable of gigahertz sampling rates, but usually has only 8 bits of resolution or fewer, since the number of comparators needed, $2^N - 1$, doubles with each additional bit, requiring a large expensive circuit. ADCs of this type have a large die size, a high input capacitance, high power dissipation, and are prone to produce glitches on the output (by outputting an out-of-sequence code). Scaling to newer submicrometre technologies does not help as the device mismatch is the dominant design limitation. They are often used for video, wideband communications or other fast signals in optical storage.

- A **successive-approximation ADC** uses a comparator to reject ranges of voltages, eventually settling on a final voltage range. Successive approximation works by constantly comparing the input voltage to the output of an internal digital to analog converter (DAC, fed by the current value of the approximation) until the best approximation is achieved. At each step in this process, a binary value of the approximation is stored in a successive approximation register (SAR). The SAR uses a reference voltage (which is the largest signal the ADC is to convert) for comparisons. For example if the input voltage is 60 V and the reference voltage is 100 V, in the 1st clock cycle, 60 V is compared to 50 V (the reference, divided by two. This is the voltage at the output of the internal DAC when the input is a '1' followed by zeros), and the voltage from the comparator is positive (or '1') (because 60 V is greater than 50 V). At this point the first binary digit (MSB) is set to a '1'. In the 2nd clock cycle the input voltage is compared to 75 V (being halfway between 100 and 50 V: This is the output of the internal DAC when its input is '11' followed by zeros) because 60 V is less than 75 V, the comparator output is now negative (or '0'). The second binary digit is therefore set to a '0'. In the 3rd clock cycle, the input voltage is compared with 62.5 V (halfway between 50 V and 75 V: This is the output of the internal DAC when its input is '101' followed by zeros). The output of the comparator is negative or '0' (because 60 V is less than 62.5 V) so the third binary digit is set to a 0. The fourth clock cycle similarly results in the fourth digit being a '1' (60 V is greater than 56.25 V, the DAC output for '1001' followed by zeros). The result of this would be in the binary form 1001. This is also called *bit-weighting conversion*, and is similar to a binary search. The analogue value is rounded to the nearest binary value below, meaning this converter type is mid-rise. Because the approximations are successive (not simultaneous), the conversion takes one clock-cycle for each bit of resolution desired. The clock frequency must be equal to the sampling frequency multiplied by the number of bits of resolution desired. For example, to sample audio at 44.1 kHz with 32 bit resolution, a clock frequency of over 1.4 MHz would be required. ADCs of this type have good resolutions and quite wide ranges. They are more complex than some other designs.

- A **ramp-compare ADC** produces a saw-tooth signal that ramps up or down then quickly returns to zero. When the ramp starts, a timer starts counting. When the ramp voltage matches the input, a comparator fires, and the timer's value is recorded. Timed ramp converters require the least number of transistors. The ramp time is sensitive to temperature because the circuit generating the ramp is often just some simple oscillator. There are two solutions: use a clocked counter driving a DAC and then use the comparator to preserve the counter's value, or calibrate the timed ramp. A special advantage of the ramp-compare system is that comparing a second signal just requires another comparator, and another register to store the voltage value. A very simple (non-linear) ramp-converter can be implemented with a microcontroller and one resistor and capacitor. Vice versa, a filled capacitor can be taken from an integrator, time-to-amplitude converter, phase detector, sample and hold circuit, or peak and hold circuit and discharged. This has the advantage that a slow comparator cannot be disturbed by fast input changes.

- The **Wilkinson ADC** was designed by D. H. Wilkinson in 1950. The Wilkinson ADC is based on the comparison of an input voltage with that produced by a charging capacitor. The capacitor is allowed to charge until its voltage is equal to the amplitude of the input pulse. (A comparator determines when this condition has been reached.) Then, the capacitor is allowed to discharge linearly, which produces a ramp voltage. At the point when the capacitor begins to discharge, a gate pulse is initiated. The gate pulse remains on until the capacitor is completely discharged. Thus the duration of the gate pulse is directly proportional to the amplitude of the input pulse. This gate pulse operates a linear gate which receives pulses from a high-frequency oscillator clock. While the gate is open, a discrete number of clock pulses pass through the linear gate and are counted by the address register. The time the linear gate is open is proportional to the amplitude of the input pulse, thus the number of clock pulses recorded in the address register is proportional also. Alternatively, the charging of the capacitor could be monitored, rather than the discharge.

- An **integrating ADC** (also **dual-slope** or **multi-slope** ADC) applies the unknown input voltage to the input of an integrator and allows the voltage to ramp for a fixed time period (the run-up period). Then a known reference voltage of opposite polarity is applied to the integrator and is allowed to ramp until the integrator output returns to zero (the run-down period). The input voltage is computed as a function of the reference voltage, the constant run-up time period, and the measured run-down time period. The run-down time measurement is usually made in units of the converter's clock, so longer integration times allow for higher resolutions. Likewise, the speed of the converter can be improved by sacrificing resolution. Converters of this type (or variations on the concept) are used in most digital voltmeters for their linearity and flexibility.

- A **delta-encoded ADC** or Counter-ramp has an up-down counter that feeds a digital to analog converter (DAC). The input signal and the DAC both go to a comparator. The comparator controls the counter. The circuit uses negative feedback from the comparator to adjust the counter until the DAC's output is close enough to the input signal. The number is read from the counter. Delta converters have very wide ranges, and high resolution, but the conversion time is dependent on the input signal level, though it will always have a guaranteed worst-case. Delta converters are often very good choices to read real-world signals. Most signals from physical systems do not change abruptly. Some converters combine the delta and successive approximation approaches; this works especially well when high frequencies are known to be small in magnitude.

- A **pipeline ADC** (also called **subranging quantizer**) uses two or more steps of subranging. First, a coarse conversion is done. In a second step, the difference to the input signal is determined with a digital to analog converter (DAC). This difference is then converted finer, and the results are combined in a last step. This can be considered a refinement of the successive approximation ADC wherein the feedback reference signal consists of the interim conversion of a whole range of bits (for example, four bits) rather than just the next-most-significant bit. By combining the merits of the successive approximation and flash ADCs this type is fast, has a high resolution, and only requires a small die size.

- A **Sigma-Delta ADC** (also known as a Delta-Sigma ADC) oversamples the desired signal by a large factor and filters the desired signal band. Generally, a smaller number of bits than required are converted using a Flash ADC after the filter. The resulting signal, along with the error generated by the discrete levels of the Flash, is fed back and subtracted from the input to the filter. This negative feedback has the effect of noise shaping the error due to the Flash so that it does not appear in the desired signal frequencies. A digital filter (decimation filter) follows the ADC which reduces the sampling rate, filters off unwanted noise signal and increases the resolution of the output (sigma-delta modulation, also called delta-sigma modulation).

- A **Time-interleaved ADC** uses M parallel ADCs where each ADC sample data every M:th cycle of the effective sample clock. The result is that the sample rate is increased M times compared to what each individual ADC can manage. In practice, the individual differences between the M ADCs degrade the overall performance reducing the SFDR. However, technologies exist to correct for these time-interleaving mismatch errors.

- An **ADC with intermediate FM stage** first uses a voltage-to-frequency converter to converts the desired signal into an oscillating signal with a frequency proportional to the voltage of the desired signal, and then uses a frequency counter to convert that frequency into a digital count proportional to the desired signal voltage. Longer integration times allow for higher resolutions. Likewise, the speed of the converter can be improved by sacrificing resolution. The two parts of the ADC may be widely separated, with the frequency signal passed through a opto-isolator or transmitted wirelessly. Some such ADCs use sine wave or square wave frequency modulation; others use pulse-frequency modulation. Such ADCs were once the most popular way to show a digital display of the status of a remote analog sensor.

There can be other ADCs that use a combination of electronics and other technologies:

- A **Time-stretch analog-to-digital converter (TS-ADC)** digitizes a very wide bandwidth analog signal, that cannot be digitized by a conventional electronic ADC, by time-stretching the signal prior to digitization. It commonly uses a photonic preprocessor frontend to time-stretch the signal, which effectively slows the signal down in time and compresses its bandwidth. As a result, an electronic backend ADC, that would have been too slow to capture the original signal, can now capture this slowed down signal. For continuous capture of the signal, the frontend also divides the signal into multiple segments in addition to time-stretching. Each segment is individually digitized by a separate electronic ADC. Finally, a digital signal processor rearranges the samples and removes any distortions added by the frontend to yield the binary data that is the digital representation of the original analog signal.

Commercial analog-to-digital converters

These are usually integrated circuits.

Most converters sample with 6 to 24 bits of resolution, and produce fewer than 1 megasample per second. Thermal noise generated by passive components such as resistors masks the measurement when higher resolution is desired. For audio applications and in room temperatures, such noise is usually a little less than 1 μV (microvolt) of white noise. If the Most Significant Bit corresponds to a standard 2 volts of output signal, this translates to a noise-limited performance that is less than 20~21 bits, and obviates the need for any dithering. Mega- and gigasample per second converters are available, though (Feb 2002). Megasample converters are required in digital video cameras, video capture cards, and TV tuner cards to convert full-speed analog video to digital video files. Commercial converters usually have ±0.5 to ±1.5 LSB error in their output.

In many cases the most expensive part of an integrated circuit is the pins, because they make the package larger, and each pin has to be connected to the integrated circuit's silicon. To save pins, it is common for slow ADCs to send their data one bit at a time over a serial interface to the computer, with the next bit coming out when a clock signal changes state, say from zero to 5 V. This saves quite a few pins on the ADC package, and in many cases, does not make the overall design any more complex (even microprocessors which use memory-mapped I/O only need a few bits of a port to implement a serial bus to an ADC).

Commercial ADCs often have several inputs that feed the same converter, usually through an analog multiplexer. Different models of ADC may include sample and hold circuits, instrumentation amplifiers or differential inputs, where the quantity measured is the difference between two voltages.

Applications

Application to music recording

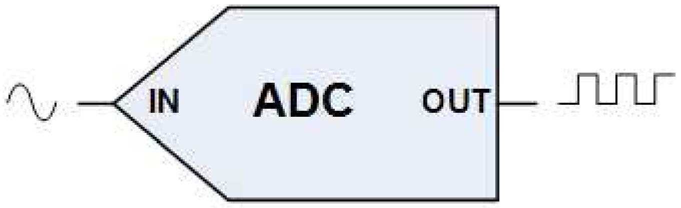
ADCs are integral to current music reproduction technology. Since much music production is done on computers, when an analog recording is used, an ADC is needed to create the PCM data stream that goes onto a compact disc or digital music file.

The current crop of AD converters utilized in music can sample at rates up to 192 kilohertz. High bandwidth headroom allows the use of cheaper or faster anti-aliasing filters of less severe filtering slopes. The proponents of oversampling assert that such shallower anti-aliasing filters produce less deleterious effects on sound quality, exactly because of their gentler slopes. Others prefer entirely filterless AD conversion, arguing that aliasing is less detrimental to sound perception than pre-conversion brickwall filtering. Considerable literature exists on these matters, but commercial considerations often play a significant role. Most high-profile recording studios record in 24-bit/192-176.4 kHz PCM or in DSD formats, and then downsample or decimate the signal for Red-Book CD production (44.1 kHz or at 48 kHz for commonly used for radio/TV broadcast applications).

Digital Signal Processing

AD converters are used virtually everywhere where an analog signal has to be processed, stored, or transported in digital form. Fast video ADCs are used, for example, in TV tuner cards. Slow on-chip 8, 10, 12, or 16 bit ADCs are common in microcontrollers. Very fast ADCs are needed in digital oscilloscopes, and are crucial for new applications like software defined radio.

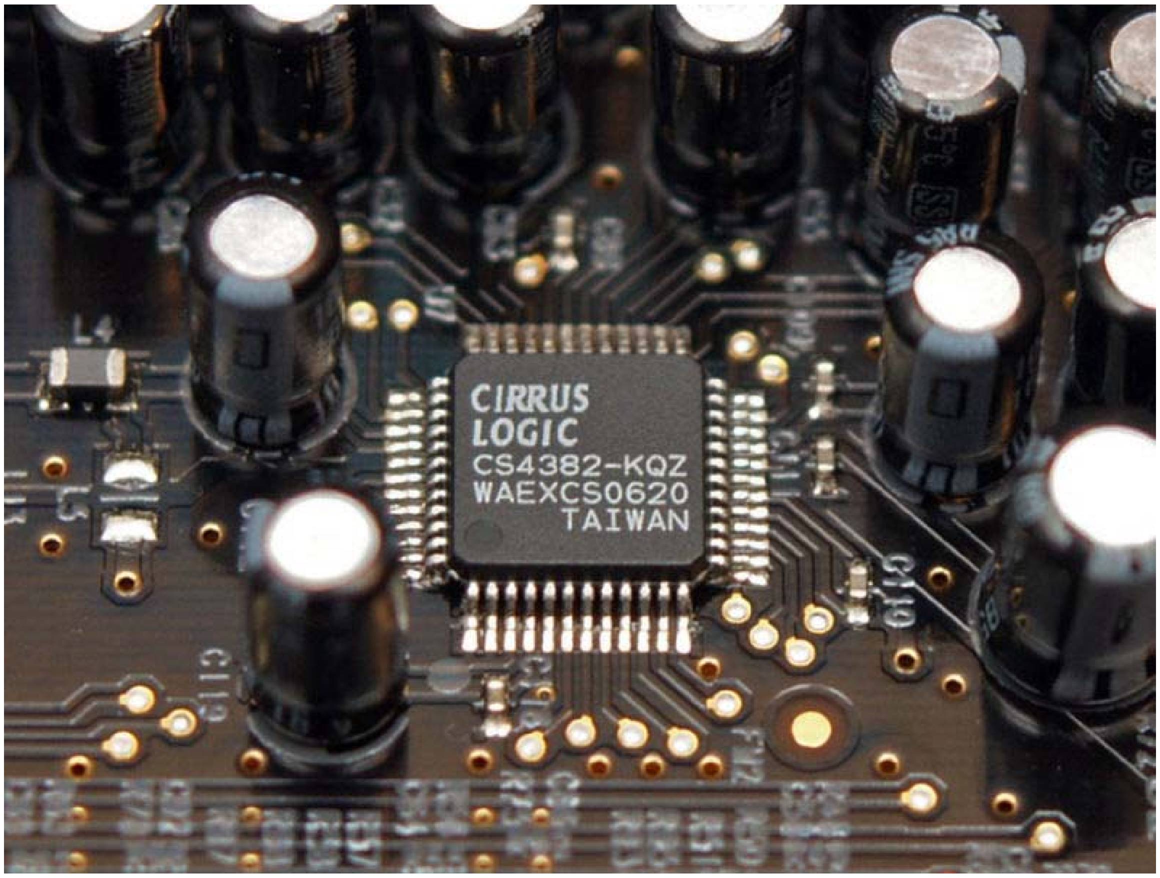
Electrical Symbol



ELECTRICAL SYMBOL FOR ANALOG TO DIGITAL CONVERTER (ADC)

Chapter 8

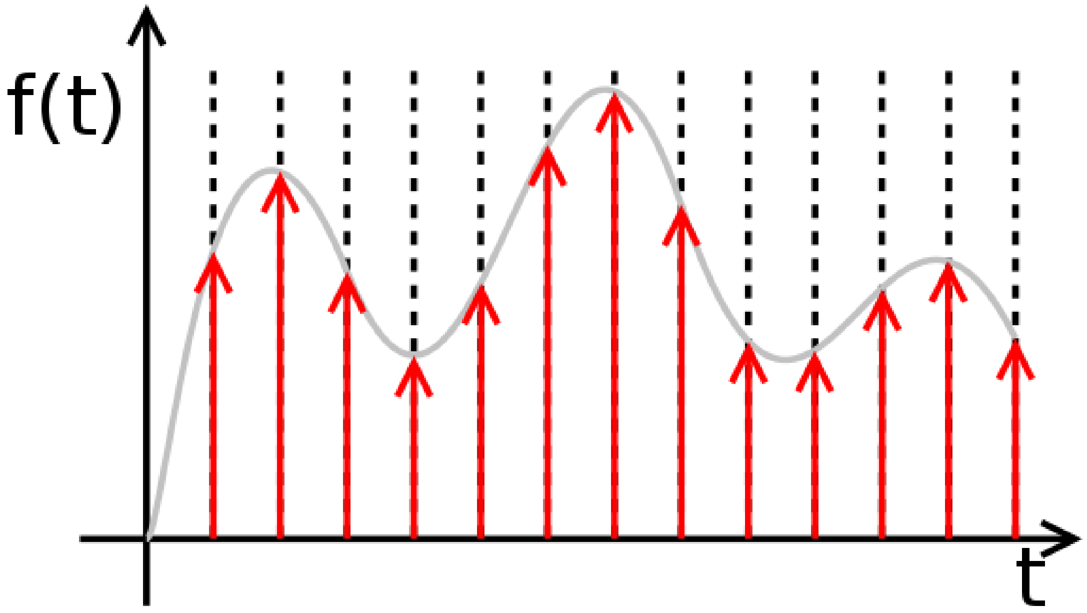
Digital-to-Analog Converter



8-channel digital-to-analog converter Cirrus Logic CS4382 as used in a soundcard.

In electronics, a **digital-to-analog converter** (**DAC** or **D-to-A**) is a device that converts a digital (usually binary) code to an analog signal (current, voltage, or electric charge). An analog-to-digital converter (ADC) performs the reverse operation.

Basic ideal operation



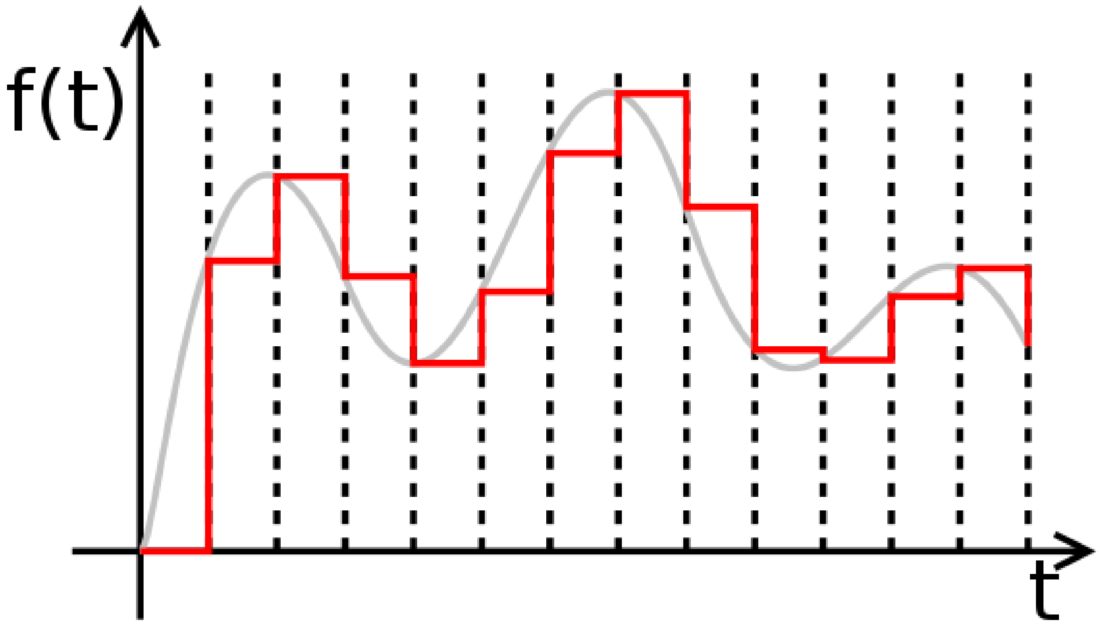
Ideally sampled signal

A DAC converts an abstract finite-precision number (usually a fixed-point binary number) into a concrete physical quantity (e.g., a voltage or a pressure). In particular, DACs are often used to convert finite-precision time series data to a continually varying physical signal.

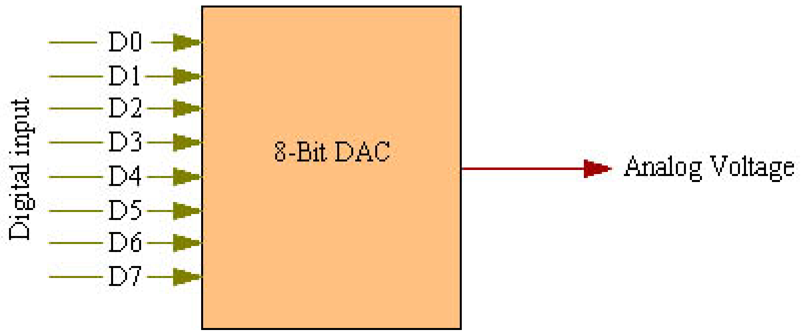
A typical DAC converts the abstract numbers into a concrete sequence of impulses that are then processed by a reconstruction filter using some form of interpolation to fill in data between the impulses. Other DAC methods (e.g., methods based on Delta-sigma modulation) produce a pulse-density modulated signal that can then be filtered in a similar way to produce a smoothly varying signal.

By the Nyquist–Shannon sampling theorem, sampled data can be reconstructed perfectly provided that its bandwidth meets certain requirements (e.g., a baseband signal with bandwidth less than the Nyquist frequency; BUT requires an infinite number of samples. The finite number used in real life cause other problems especially with the D/A reconstruction of the original signal. However, even with an ideal reconstruction filter, digital sampling introduces quantization error that makes perfect reconstruction practically impossible. Increasing the digital resolution (i.e., increasing the number of bits used in each sample) or introducing sampling dither can reduce this error.

Practical operation



Piecewise constant output of a conventional practical DAC.



A simplified functional diagram of an 8-bit DAC

Instead of impulses, usually the sequence of numbers update the analogue voltage at uniform sampling intervals.

These numbers are written to the DAC, typically with a clock signal that causes each number to be latched in sequence, at which time the DAC output voltage changes rapidly from the previous value to the value represented by the currently latched number. The effect of this is that the output voltage is *held* in time at the current value until the next input number is latched resulting in a piecewise constant or 'staircase' shaped output. This is equivalent to a zero-order hold operation and has an effect on the frequency response of the reconstructed signal.

The fact that DACs output a sequence of piecewise constant values (known as zero-order hold in sample data textbooks) or rectangular pulses causes multiple harmonics above the

Nyquist frequency. Usually, these are removed with a low pass filter acting as a reconstruction filter in applications that require it.

Applications

Audio

Top-loading CD player and external digital-to-analog converter.

Most modern audio signals are stored in digital form (for example MP3s and CDs) and in order to be heard through speakers they must be converted into an analog signal. DACs are therefore found in CD players, digital music players, and PC sound cards.

Specialist standalone DACs can also be found in high-end hi-fi systems. These normally take the digital output of a compatible CD player or dedicated transport and convert the signal into an analog line-level output that can then be fed into an amplifier to drive speakers.

Similar digital-to-analog converters can be found in digital speakers such as USB speakers, and in sound cards.

VOIP (Voice over IP) Phone, Data transmission over the Internet is done digitally so in order for voice to be transmitted it must be converted to digital using an Analog-to-Digital Converter and be converted into analog again using a DAC so the voice it can be heard on the other end.

Video

Video signals from a digital source, such as a computer, must be converted to analog form if they are to be displayed on an analog monitor. As of 2007, analog inputs are more commonly used than digital, but this may change as flat panel displays with DVI and/or HDMI connections become more widespread. A video DAC is, however, incorporated in any digital video player with analog outputs. The DAC is usually integrated with some memory (RAM), which contains conversion tables for gamma correction, contrast and brightness, to make a device called a RAMDAC.

A device that is distantly related to the DAC is the digitally controlled potentiometer, used to control an analog signal digitally.

Mechanical

An unusual application of digital-to-analog conversion was the whiffletree electromechanical digital-to-analog convertor linkage in the IBM Selectric typewriter.

DAC types

The most common types of electronic DACs are:

- The pulse-width modulator, the simplest DAC type. A stable current or voltage is switched into a low-pass analog filter with a duration determined by the digital input code. This technique is often used for electric motor speed control, but has many other applications as well.
- Oversampling DACs or interpolating DACs such as the delta-sigma DAC, use a pulse density conversion technique. The oversampling technique allows for the use of a lower resolution DAC internally. A simple 1-bit DAC is often chosen because the oversampled result is inherently linear. The DAC is driven with a pulse-density modulated signal, created with the use of a low-pass filter, step nonlinearity (the actual 1-bit DAC), and negative feedback loop, in a technique called delta-sigma modulation. This results in an effective high-pass filter acting on the quantization (signal processing) noise, thus steering this noise out of the low frequencies of interest into the megahertz frequencies of little interest, which is called noise shaping. The quantization noise at these high frequencies is removed or greatly attenuated by use of an analog low-pass filter at the output (sometimes a simple RC low-pass circuit is sufficient). Most very high resolution DACs (greater than 16 bits) are of this type due to its high linearity and low cost. Higher oversampling rates can relax the specifications of the output low-pass filter and enable further suppression of quantization noise. Speeds of greater than

100 thousand samples per second (for example, 192 kHz) and resolutions of 24 bits are attainable with delta-sigma DACs. A short comparison with pulse-width modulation shows that a 1-bit DAC with a simple first-order integrator would have to run at 3 THz (which is physically unrealizable) to achieve 24 meaningful bits of resolution, requiring a higher-order low-pass filter in the noise-shaping loop. A single integrator is a low-pass filter with a frequency response inversely proportional to frequency and using one such integrator in the noise-shaping loop is a first order delta-sigma modulator. Multiple higher order topologies (such as MASH) are used to achieve higher degrees of noise-shaping with a stable topology.

- The binary-weighted DAC, which contains one resistor or current source for each bit of the DAC connected to a summing point. These precise voltages or currents sum to the correct output value. This is one of the fastest conversion methods but suffers from poor accuracy because of the high precision required for each individual voltage or current. Such high-precision resistors and current sources are expensive, so this type of converter is usually limited to 8-bit resolution or less.
- The R-2R ladder DAC which is a binary-weighted DAC that uses a repeating cascaded structure of resistor values R and 2R. This improves the precision due to the relative ease of producing equal valued-matched resistors (or current sources). However, wide converters perform slowly due to increasingly large RC-constants for each added R-2R link.
- The thermometer-coded DAC, which contains an equal resistor or current-source segment for each possible value of DAC output. An 8-bit thermometer DAC would have 255 segments, and a 16-bit thermometer DAC would have 65,535 segments. This is perhaps the fastest and highest precision DAC architecture but at the expense of high cost. Conversion speeds of >1 billion samples per second have been reached with this type of DAC.
- Hybrid DACs, which use a combination of the above techniques in a single converter. Most DAC integrated circuits are of this type due to the difficulty of getting low cost, high speed and high precision in one device.
    - The segmented DAC, which combines the thermometer-coded principle for the most significant bits and the binary-weighted principle for the least significant bits. In this way, a compromise is obtained between precision (by the use of the thermometer-coded principle) and number of resistors or current sources (by the use of the binary-weighted principle). The full binary-weighted design means 0% segmentation, the full thermometer-coded design means 100% segmentation.

DAC performance

DACs are very important to system performance. The most important characteristics of these devices are:

- **Resolution**: This is the number of possible output levels the DAC is designed to reproduce. This is usually stated as the number of bits it uses, which is the base two logarithm of the number of levels. For instance a 1 bit DAC is designed to reproduce 2 (2^1) levels while an 8 bit DAC is designed for 256 (2^8) levels. Resolution is related to the **effective number of bits** (ENOB) which is a measurement of the actual resolution attained by the DAC.

- **Maximum sampling frequency**: This is a measurement of the maximum speed at which the DACs circuitry can operate and still produce the correct output. As stated in the Nyquist–Shannon sampling theorem, a signal must be sampled at over twice the frequency of the desired signal. For instance, to reproduce signals in all the audible spectrum, which includes frequencies of up to 20 kHz, it is necessary to use DACs that operate at over 40 kHz. The CD standard samples audio at 44.1 kHz, thus DACs of this frequency are often used. A common frequency in cheap computer sound cards is 48 kHz — many work at only this frequency, offering the use of other sample rates only through (often poor) internal resampling.
- **Monotonicity**: This refers to the ability of a DAC's analog output to move only in the direction that the digital input moves (i.e., if the input increases, the output doesn't dip before asserting the correct output.) This characteristic is very important for DACs used as a low frequency signal source or as a digitally programmable trim element.
- **THD+N**: This is a measurement of the distortion and noise introduced to the signal by the DAC. It is expressed as a percentage of the total power of unwanted harmonic distortion and noise that accompany the desired signal. This is a very important DAC characteristic for dynamic and small signal DAC applications.
- **Dynamic range**: This is a measurement of the difference between the largest and smallest signals the DAC can reproduce expressed in decibels. This is usually related to DAC resolution and noise floor.

Other measurements, such as phase distortion and jitter, can also be very important for some applications.

Bits	Color limit	Frequency	Examples
10	1.024 colors	54 MHz	
12		54 MHz	Sony NS-575p
12	4.096 colors	108 MHz	
12		150 MHz	NeoDigits Helios X5000
12		216 MHz	Philips BDP9000 (Blu-ray)
12		297 MHz	Toshiba HD-XE1
12		216 MHz	Samsung BD-P1200 (Blu-ray)
14	16.384 colors	108 MHz	Pioneer Elite, Black Finish, DV79AVI
14		216 MHz	Marantz DV9600, Sony DVPNS9100ES
16		149 MHz	NeuNeo HVD108

DAC figures of merit

- Static performance:
    - Differential nonlinearity (DNL) shows how much two adjacent code analog values deviate from the ideal 1LSB step
    - Integral nonlinearity (INL) shows how much the DAC transfer characteristic deviates from an ideal one. That is, the ideal characteristic is usually a straight line; INL shows how much the actual voltage at a given code value differs from that line, in LSBs (1LSB steps).
    - Gain
    - Offset
    - Noise is ultimately limited by the thermal noise generated by passive components such as resistors. For audio applications and in room temperatures, such noise is usually a little less than 1 μV (microvolt) of white noise. This limits performance to less than 20~21 bits even in 24-bit DACs.
- Frequency domain performance
    - Spurious-free dynamic range (SFDR) indicates in dB the ratio between the powers of the converted main signal and the greatest undesired spur
    - Signal to noise and distortion ratio (SNDR) indicates in dB the ratio between the powers of the converted main signal and the sum of the noise and the generated harmonic spurs
    - i-th harmonic distortion (HDi) indicates the power of the i-th harmonic of the converted main signal
    - Total harmonic distortion (THD) is the sum of the powers of all HDi
    - If the maximum DNL error is less than 1 LSB, then D/A converter is guaranteed to be monotonic.

However, many monotonic converters may have a maximum DNL greater than 1 LSB.

- Time domain performance:
    - Glitch energy
    - Response uncertainty
    - Time nonlinearity (TNL)

Chapter 9

Control Register

A **control register** is a processor register which changes or controls the general behavior of a CPU or other digital device. Common tasks performed by control registers include interrupt control, switching the addressing mode, paging control, and coprocessor control.

Control registers in x86 series

CR0

The CR0 register is 32 bits long on the 386 and higher processors. On x86-64 processors in long mode, it (and the other control registers) are 64 bits long. CR0 has various control flags that modify the basic operation of the processor.

Bit	Name	Full Name	Description
31	PG	Paging	If 1, enable paging and use the CR3 register, else disable paging
30	CD	Cache disable	Globally enables/disable the memory cache
29	NW	Not-write through	Globally enables/disable write-back caching
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
16	WP	Write protect	Determines whether the CPU can write to pages marked read-only
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
3	TS	Task switched	Allows saving x87 task context only after x87 instruction used after task switch
2	EM	Emulation	If set, no x87 floating point unit present, if clear, x87 FPU present
1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0

0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode

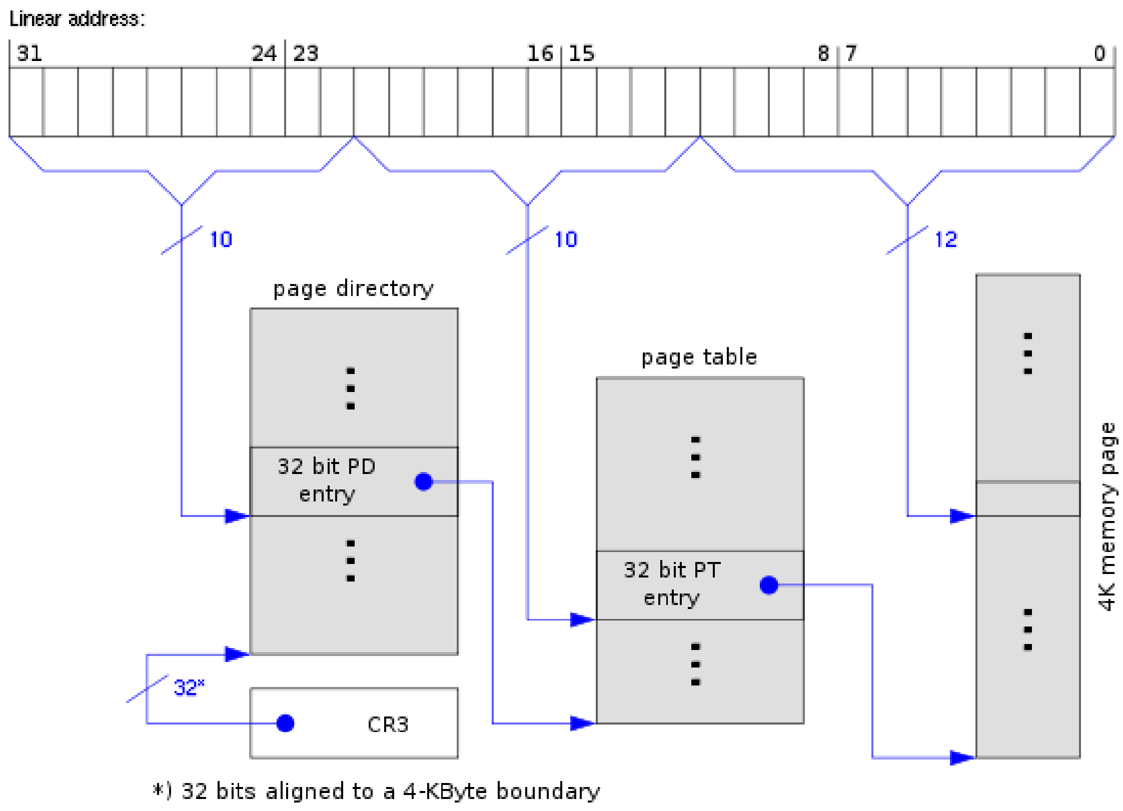
CR1

Reserved

CR2

Contains a value called Page Fault Linear Address (PFLA). When a page fault occurs, the address the program attempted to access is stored in the CR2 register.

CR3



Typical use of CR3 in address translation with 4 KiB pages.

Used when virtual addressing is enabled, hence when the PG bit is set in CR0. CR3 enables the processor to translate virtual addresses into physical addresses by locating the page directory and page tables for the current task. Typically, the upper 20 bits of CR3 become the *page directory base register* (PDBR).

CR4

Used in protected mode to control operations such as virtual-8086 support, enabling I/O breakpoints, page size extension and machine check exceptions.

Bit	Name	Full Name	Description
18	OSXSAVE	XSAVE and Processor Extended States Enable	
17	PCIDE	PCID Enable	If set, enables process-context identifiers (PCIDs).
14	SMXE	SMX Enable	
13	VMXE	VMX Enable	
10	OSXMMEXCPT	Operating System Support for Unmasked SIMD Floating-Point Exceptions	If set, enables unmasked SSE exceptions.
9	OSFXSR	Operating system support for FXSAVE and FXSTOR instructions	If set, enables SSE instructions and fast FPU save & restore
8	PCE	Performance-Monitoring Counter enable	If set, RDPMC can be executed at any privilege level, else RDPMC can only be used in ring 0.
7	PGE	Page Global Enabled	If set, address translations (PDE or PTE records) may be shared between address spaces.
6	MCE	Machine Check Exception	If set, enables machine check interrupts to occur.
5	PAE	Physical Address Extension	If set, changes page table layout to translate 32-bit virtual addresses into extended 36-bit physical addresses.
4	PSE	Page Size Extensions	If unset, page size is 4 KB, else page size is increased to 4 MB (ignored with PAE set).
3	DE	Debugging Extensions	
2	TSD	Time Stamp Disable	If set, RDTSC instruction can only be executed when in ring 0, otherwise RDTSC can be used at any privilege level.
1	PVI	Protected-mode Virtual Interrupts	If set, enables support for the virtual interrupt flag (VIF) in protected mode.

0	VME	Virtual 8086 Mode Extensions	If set, enables support for the virtual interrupt flag (VIF) in virtual-8086 mode.

Additional Control registers in x86-64 series

EFER

Extended Feature Enable Register (EFER) is a register added in the AMD K6 processor, to allow enabling the SYSCALL/SYSRET instruction, and later for entering and exiting long mode.

Bit	Purpose
63:12	Reserved
11	Execute-disable bit enable (NXE)
10	IA-32e mode active (LMA)
9	Reserved
8	IA-32e mode enable (LME)
7:1	Reserved
0	SysCall enable (SCE)

Chapter 10

Hardware Register and Processor Register

Hardware register

In digital electronics, especially computing, a **hardware register** stores bits of information, in a way that all the bits can be written to or read out simultaneously. The hardware registers inside a central processing unit (CPU) are called processor registers. Signals from a state machine to the register control when registers transmit to or accept information from other registers. Sometimes the state machine routes information from one register through a functional transform, such as an adder unit, and then to another register that stores the results.

Typical uses of hardware registers include *configuration* and start-up of certain features, especially during initialization, *buffer storage* e.g. video memory for graphics cards, input/output (I/O) of different kinds, and *status reporting* such as whether a certain event has occurred in the hardware unit.

Reading a hardware register in "peripheral units" -- computer hardware outside the CPU—involves accessing its memory-mapped I/O address or port-mapped I/O address with a "load" or "store" instruction, issued by the processor. Hardware registers are addressed in words, but sometimes only use a few bits of the word read in to, or written out to the register.

Strobe registers have the same interface as normal hardware registers, but instead of storing data, they trigger an action each time they are written to (or, in rare cases, read from). They are a means of signaling.

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". Registers can be implemented in a wide variety of ways, including register files, standard SRAM, individual flip-flops, or high speed core memory.

In addition to the "programmer-visible" registers that can be read and written with software, many chips have internal registers that are used for state machines and pipelining; for example, registered memory.

Commercial design tools such as Socrates Bitwise by Duolog Technologies, simplify and automate memory-mapped register specification and code generation for hardware, firmware, hardware verification, testing and documentation.,,

Using IP-XACT IEEE 1685, commercial design tools, such as MRV Magillem Register View by MAGILLEM, provide a real synchronization between the register description and the RTL hardware platform description, then collaborative work in the design flow can be addressed. This hardware registers alignment becomes critical when multiple levels of abstraction are used when switching from TLM to RTL during IP integration.

SPIRIT IP-XACT and DITA SIDSC XML define standard XML formats for memory-mapped registers.,,

Because write-only registers make debugging almost impossible, lead to the read-modify-write problem, and also make it unnecessarily difficult for the Advanced Configuration and Power Interface to determine the device's state when entering sleep mode in order to restore that state when exiting sleep mode, many programmers tell hardware designers to make sure that all writable registers are also readable. However, there are some cases when reading certain types of hardware registers is useless. For example, a strobe register bit that generates a one cycle pulse into specialized hardware will always read logic 0.

Processor register

In computer architecture, a **processor register** (or **general purpose register**) is a small amount of storage available on the CPU whose contents can be accessed more quickly than storage available elsewhere. Typically, this specialized storage is not considered part of the normal memory range for the machine. Most, but not all, modern computers adopt the so-called load-store architecture. Under this paradigm, data is loaded from some larger memory — be it cache or RAM — into registers, manipulated or tested in some way (using machine instructions for arithmetic/logic/comparison) and then stored back into memory, possibly at some different location. A common property of computer programs is locality of reference: the same values are often accessed repeatedly; and holding these frequently used values in registers improves program execution performance.

Processor registers are at the top of the memory hierarchy, and provide the fastest way for a CPU to access data. The term is often used to refer only to the group of registers that are directly encoded as part of an instruction, as defined by the instruction set. More properly, these are called the "architectural registers". For instance, the IA-32 instruction set defines a set of 32-bit registers, but a CPU that implements the x86 instruction set will often contain many more registers than just these registers.

Allocating frequently used variables to registers can be critical to a program's performance. This action (register allocation) is performed by a compiler in the code generation phase.

Categories of registers

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". A processor often contains several kinds of registers, that can be classified accordingly to their content or instructions that operate on them:

- **User-accessible Registers** - The most common division of user-accessible registers is into data registers and address registers.
- **Data registers** are used to hold numeric values such as integer and floating-point values. In some older and low end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.
- **Address registers** hold addresses and are used by instructions that indirectly access memory.
    - Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address); others allow registers to hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist.
    - A stack pointer, sometimes called a stack register, is the name given to a register that can be used by some instructions to maintain a stack.
- **Conditional registers** hold truth values often used to determine whether some instruction should or should not be executed.
- **General purpose registers** (**GPR**s) can store both data and addresses, i.e., they are combined Data/Address registers.
- **Floating point registers** (**FPR**s) store floating point numbers in many architectures.
- **Constant registers** hold read-only values such as zero, one, or pi.
- **Vector registers** hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- **Special purpose registers (SPR)** hold program state; they usually include the program counter (aka instruction pointer), stack pointer, and status register (aka processor status word). In embedded microprocessors, they can also correspond to specialized hardware elements.
    - **Instruction registers** store the instruction currently being executed.
- In some architectures, **model-specific registers** (also called *machine-specific registers*) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.

- **Control and status registers** - It has three types. Program counter, instruction registers, Program status word (PSW).
- Registers related to fetching information from RAM, a collection of storage registers located on separate chips from the CPU (unlike most of the above, these are generally not *architectural* registers):
    - Memory buffer register
    - Memory data register
    - Memory address register
    - Memory Type Range Registers (MTRR)

Hardware registers are similar, but occur outside CPUs.

Some examples

Architecture	**Integer registers**	**Double FP registers**
x86	8	8
x86-64	16	16
IBM/360	16	4
Z/Architecture	16	16
Itanium	128	128
UltraSPARC	32	32
POWER	32	32
Alpha	32	32
6502	3	0
PIC microcontroller	1	0
AVR microcontroller	32	0
ARM	16	16

The table shows the number of registers of several mainstream architectures. Note that the stack pointer (ESP) is counted as an integer register on x86-compatible processors, even though there are a limited number of instructions that may be used to operate on its contents. Similar caveats apply to most architectures.

Register usage

The number of registers available on a processor and the operations that can be performed using those registers has a significant impact on the efficiency of code generated by optimizing compilers. The Strahler number defines the minimum number of registers required to evaluate an expression tree.

Chapter 11

Flag Word and FLAGS Register (Computing)

Flag word

A **flag word** is a generic term for a word (value) used to indicate conditions within a binary computer. In particular, the byte can be used to show up to 8 discrete conditions. Essentially, each bit represents one flag, capable of showing two states.

A flag word can be seen as a bit array with a short, constant length.

Examples

Processor status register

Taking the example of a 6502 processor's status register, the following information was held within 8 bits:

- Bit 7. Negative flag
- Bit 6. Overflow flag
- Bit 5. Unused
- Bit 4. Break flag
- Bit 3. Decimal flag
- Bit 2. Interrupt-disable flag
- Bit 1. Carry flag
- Bit 0. Zero flag

As you can see, a lot of information about the state of the processor can be packed into 8 bits.

Unix process exit code

A more general example would be the use of a Unix exit status as a flag word. In this case, the exit code is by the programmer to pass status information to another process. An imaginary program which returns the status of 8 burglar alarm switches connected to the printer port could set each of the bits in the exit code in turn, depending on whether the switches are closed or open.

Extracting bits from flag words

To read a status byte, assuming your programming language does not offer this facility by default, is quite easy. You simply need to AND the status byte with a mask byte. The mask byte should have only the bit corresponding to the flag you want to read set, as in the example below.

Suppose that status byte 103 (decimal) is returned, and that we want to check flag bit 5.

The flag we want to read is number 5 (counting from zero) - so the mask byte will be 2^5 = 32. ANDing 32 with 103 gives 32, which means the flag bit is set. If the flag bit was not set, the result would have been 0.

In modern computing, the shift operator (<<) can be used to quickly perform the power-of-two. In general, a mask with the Nth bit set can be computed as

```
(1 << n)
```

Thus to check the Nth bit from a variable **v**, we can perform the operation

```
bool nth_is_set = (v & (1 << n)) != 0
```

Changing bits in flag words

Writing, reading or toggling bits in flags can be done only using the OR, AND and NOT operations - operations which can be performed quickly in the processor.

To set a bit, OR the status byte with a mask byte. Any bits set in the mask byte or the status byte will be set in the result.

```
int setBit(int val, int bit_position)
{
  return val | (1 << bit_position);
}
```

To clear a bit, perform a NOT operation on the mask byte, then AND it with the status byte. The result will have the appropriate flag cleared (set to 0).

```
int clearBit(int val, int bit_position)
{
  return val & ~(1 << bit_position);
}
```

To toggle a bit, XOR the status byte and the mask byte. This will set a bit if it is cleared or clear a bit if it is set.

```
int toggleBit(int val, int bit_position)
{
  return val ^ (1 << bit_position);
}
```

FLAGS register (computing)

The **FLAGS** register is the status register in Intel x86 microprocessors that contains the current state of the processor. This register is 16-bits wide. Its successors, the **EFLAGS** and **RFLAGS** registers are 32-bits and 64-bits wide, respectively. The wider registers retain compatibility with their smaller predecessors.

Intel x86 FLAGS Register			
Bit #	**Abbreviation**	**Description**	**Category**
		FLAGS	
0	CF	Carry flag	S
1	1	Reserved	
2	PF	Parity flag	S
3	0	Reserved	
4	AF	Adjust flag	S
5	0	Reserved	
6	ZF	Zero flag	S
7	SF	Sign flag	S
8	TF	Trap flag (single step)	X
9	IF	Interrupt enable flag	X
10	DF	Direction flag	C
11	OF	Overflow flag	S
12, 13	IOPL	I/O privilege level (286+ only)	X
14	NT	Nested task flag (286+ only)	X
15	0	Reserved	
		EFLAGS	
16	RF	Resume flag (386+ only)	X
17	VM	Virtual 8086 mode flag (386+ only)	X
18	AC	Alignment check (486SX+ only)	X
19	VIF	Virtual interrupt flag (Pentium+)	X
20	VIP	Virtual interrupt pending (Pentium+)	X
21	ID	Able to use CPUID instruction (Pentium+)	X
22	0	Reserved	
23	0	Reserved	

24	0	Reserved
25	0	Reserved
26	0	Reserved
27	0	Reserved
28	0	Reserved
29	0	Reserved
30	0	Reserved
31	0	Reserved
	RFLAGS	
32-63	0	Reserved

Examples

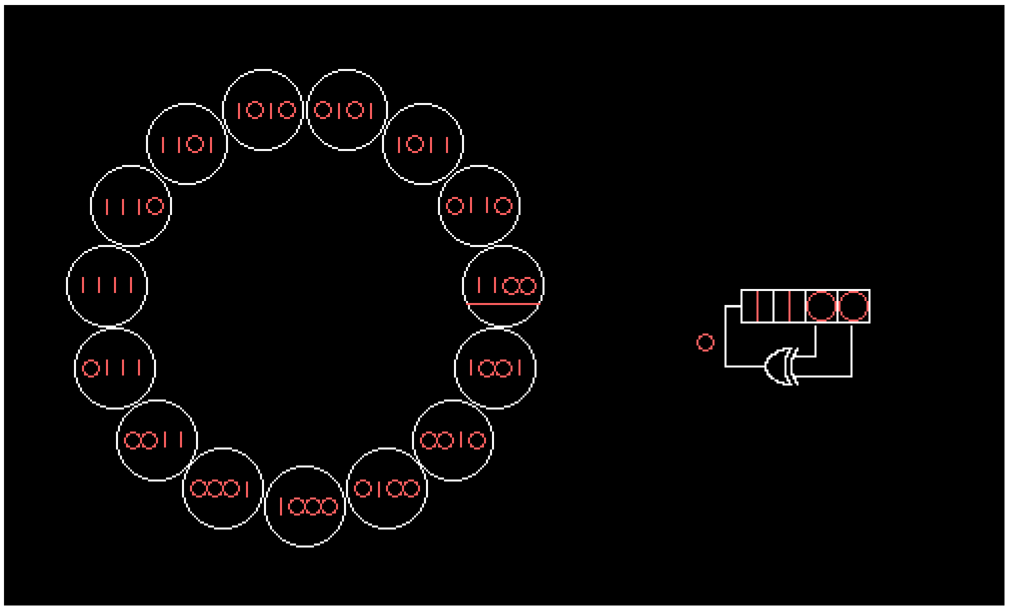
Below is an example for changing the flag in DF (direction flag)

```
mov bx, 400h ; Set the DF flag
pushf ; Pushes the current flags onto the stack
pop ax ; Pop the flags from the stack into ax register
push ax ; Push them back onto the stack for storage
xor ax, bx ; XOR dest, src | Used for toggling the DF flag only, keep
the rest of the flags
push ax ; Push again to add the new value to the stack
popf ; Pop the newly pushed into the FLAGS register
; ... Code here ...
popf ; Pop the old FLAGS back into place

;In practical software, the cld and std mnemonics are used to clear and
set the direction flag, respectively.
```

Chapter 12

Linear Feedback Shift Register



A 4-bit Fibonacci LFSR with its state diagram. The XOR gate provides feedback to the register that shifts bits from left to right. The maximal sequence consists of every possible state except the "0000" state.

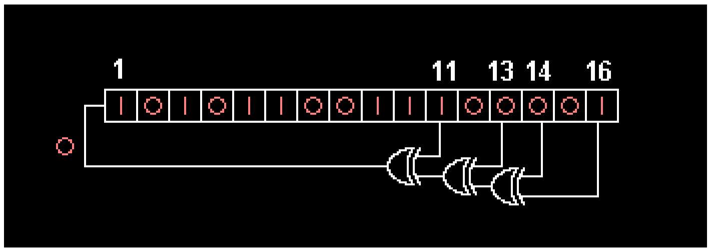
A **linear feedback shift register** (LFSR) is a shift register whose input bit is a linear function of its previous state.

The only linear function of single bits is xor, thus it is a shift register whose input bit is driven by the exclusive-or (xor) of some bits of the overall shift register value.

The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle.

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common.

Fibonacci LFSRs



A 16-bit Fibonacci LFSR. The feedback tap numbers in white correspond to a primitive polynomial in the table so the register cycles through the maximum number of 65535 states excluding the all-zeroes state. The state ACE1 hex shown will be followed by 5670 hex.

The bit positions that affect the next state are called the taps. In the diagram the taps are [16,14,13,11]. The rightmost bit of the LFSR is called the output bit. The taps are XOR'd sequentially with the output bit and then fed back into the leftmost bit. The sequence of bits in the rightmost position is called the output stream.

- The bits in the LFSR state which influence the input are called *taps* (white in the diagram).
- A maximum-length LFSR produces an m-sequence (i.e. it cycles through all possible $2^n - 1$ states within the shift register except the state where all bits are zero), unless it contains all zeros, in which case it will never change.
- As an alternative to the XOR based feedback in an LFSR, one can also use XNOR. This function is not linear, but it results in an equivalent polynomial counter whose state of this counter is the complement of the state of an LFSR. A state with all ones is illegal when using an XNOR feedback, in the same way as a state with all zeroes is illegal when using XOR. This state is considered illegal because the counter would remain "locked-up" in this state.

The sequence of numbers generated by an LFSR or its XNOR counterpart can be considered a binary numeral system just as valid as Gray code or the natural binary code.

The arrangement of taps for feedback in an LFSR can be expressed in finite field arithmetic as a polynomial mod 2. This means that the coefficients of the polynomial must be 1's or 0's. This is called the feedback polynomial or characteristic polynomial. For example, if the taps are at the 16th, 14th, 13th and 11th bits (as shown), the feedback polynomial is

$$x^{16} + x^{14} + x^{13} + x^{11} + 1.$$

The 'one' in the polynomial does not correspond to a tap — it corresponds to the input to the first bit (i.e. x^0, which is equivalent to 1). The powers of the terms represent the tapped bits, counting from the left. The first and last bits are always connected as an input and tap respectively.

Tables of primitive polynomials from which maximum-length LFSRs can be constructed are given below and in the references.

- The LFSR will only be maximum-length if the number of taps is even; just 2 or 4 taps can suffice even for extremely long sequences.
- The set of taps must be relatively prime, and share no common divisor to all taps.
- There can be more than one maximum-length tap sequence for a given LFSR length
- Once one maximum-length tap sequence has been found, another automatically follows. If the tap sequence, in an n-bit LFSR, is $[n, A, B, C, 0]$, where the 0 corresponds to the $x^0 = 1$ term, then the corresponding 'mirror' sequence is $[n, n - C, n - B, n - A, 0]$. So the tap sequence [32, 7, 3, 2, 0] has as its counterpart [32, 30, 29, 25, 0]. Both give a maximum-length sequence.

Some example C/C++ code is below (assuming 16-bit `short`s):

```
unsigned short lfsr = 0xACE1u;
unsigned bit;
unsigned period = 0;
do
{
  /* taps: 16 14 13 11; characteristic polynomial: x^16 + x^14 + x^13 +
x^11 + 1 */
  bit  = (((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1)
^ 1;
  lfsr =  (lfsr >> 1) | (bit << 15);
  ++period;
} while(lfsr != 0xACE1u);
```

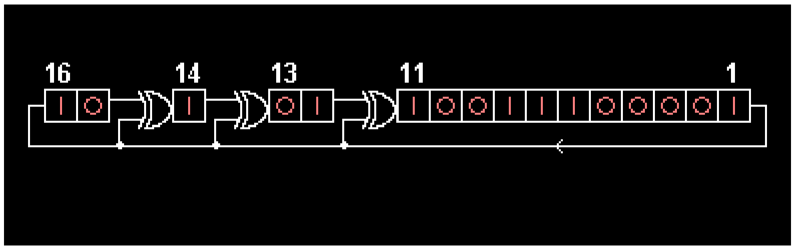
The above code assumes the most significant bit of `lfsr` is bit 1, and the least significant bit is bit 16.

As well as *Fibonacci*, this LFSR configuration is also known as **standard**, **many-to-one** or **external XOR gates**. LFSR has an alternative configuration.

Galois LFSRs

Named after the French mathematician Évariste Galois, an LFSR in Galois configuration, which is also known as **modular**, **internal XORs** as well as **one-to-many LFSR**, is an alternate structure that can generate the same output stream as a conventional LFSR. In the Galois configuration, when the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XOR'd with the output bit before they are stored in the next position. The new output bit is the next input bit.

The effect of this is that when the output bit is zero all the bits in the register shift to the right unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.



A 16-bit Galois LFSR. The register numbers in white correspond to the same primitive polynomial as the Fibonacci example but are counted in reverse to the shifting direction. This register also cycles through the maximal number of 65535 states excluding the all-zeroes state. The state ACE1 hex shown will be followed by E270 hex.

To generate the same output stream, the order of the taps is the *counterpart* of the order for the conventional LFSR, otherwise the stream will be in reverse. Note that the internal state of the LFSR is not necessarily the same. The Galois register shown has the same output stream as the Fibonacci register in the first section.

- Galois LFSRs do not concatenate every tap to produce the new input (the XOR'ing is done within the LFSR and no XOR gates are run in serial, therefore the propagation times are reduced to that of one XOR rather than a whole chain), thus it is possible for each tap to be computed in parallel, increasing the speed of execution.
- In a software implementation of an LFSR, the Galois form is more efficient as the XOR operations can be implemented a word at a time: only the output bit must be examined individually.

Below is a code example of a 32-bit maximal period Galois LFSR that is valid in C and C++, (assuming that `unsigned int` has 32 bit precision):

```
unsigned lfsr = 1;
unsigned period = 0;

do {
  /* taps: 32 31 29 1; characteristic polynomial: x^32 + x^31 + x^29 +
x + 1 */
  lfsr = (lfsr >> 1) ^ (unsigned int)(0 - (lfsr & 1u) & 0xd0000001u);
  ++period;
} while(lfsr != 1u);
```

And here is the code for the 16 bit example in the figure (Assuming 16-bit `short`s)

```
unsigned short lfsr = 0xACE1u;
unsigned period = 0;
```

```
do {
  /* taps: 16 14 13 11; characteristic polynomial: x^16 + x^14 + x^13 +
x^11 + 1 */
  lfsr = (lfsr >> 1) ^ (-(lfsr & 1u) & 0xB400u);
  ++period;
} while(lfsr != 0xACE1u);
```

These code examples create a toggle mask to apply to the shifted value using the XOR operator. The mask is created by first removing all but the least significant bit (the output bit) of the current value. This value is then negated (two's complement negation), which creates a value of either all 0s or all 1s, if the output bit is 0 or 1, respectively. By ANDing the result with the tap-value (e.g., 0xB400 in the second example) before applying it as the toggle mask, it acts functionally as a conditional to either apply or not apply the toggle mask based on the output bit. A more explicit but significantly less efficient code example is shown below.

```
unsigned short lfsr = 0xACE1u;
unsigned period = 0;

do {
  unsigned lsb = lfsr & 1;  /* Get lsb (i.e., the output bit). */
  lfsr >>= 1;               /* Shift register */
  if (lsb == 1)             /* Only apply toggle mask if output bit is
1. */
    lfsr ^= 0xB400u;        /* Apply toggle mask, value has 1 at bits
corresponding
                             * to taps, 0 elsewhere. */
  ++period;
} while(lfsr != 0xACE1u);
```

Non-binary Galois LFSR

Binary Galois LFSRs like the ones shown above can be generalized to any q-ary alphabet $\{0, 1, ... , q - 1\}$ (e.g., for binary, q is equal to two, and the alphabet is simply $\{0, 1\}$). In this case, the exclusive-or component is generalized to addition modulo-q (note that XOR is addition modulo 2), and the feedback bit (output bit) is multiplied (modulo-q) by a q-ary value which is constant for each specific tap point. Note that this is also a generalization of the binary case, where the feedback is multiplied by either 0 (no feedback, i.e., no tap) or 1 (feedback is present). Given an appropriate tap configuration, such LFSRs can be used to generate Galois fields for arbitrary values of q.

Some polynomials for maximal LFSRs

The following table lists maximal-length polynomials for shift-register lengths up to 19. Note that more than one maximal-length polynomial may exist for any given shift-register length.

Bits	**Feedback polynomial**	**Period**
n		**$2^n - 1$**
2	$x^2 + x + 1$	3
3	$x^3 + x^2 + 1$	7
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^3 + 1$	31
6	$x^6 + x^5 + 1$	63
7	$x^7 + x^6 + 1$	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	255
9	$x^9 + x^5 + 1$	511
10	$x^{10} + x^7 + 1$	1023
11	$x^{11} + x^9 + 1$	2047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	4095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	8191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	16383
15	$x^{15} + x^{14} + 1$	32767
16	$x^{16} + x^{14} + x^{13} + x^{11} + 1$	65535
17	$x^{17} + x^{14} + 1$	131071
18	$x^{18} + x^{11} + 1$	262143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	524287
20 to 168		

Output-stream properties

- Ones and zeroes occur in 'runs'. The output stream 0110100, for example consists of five runs of lengths 1,2,1,1,2, in order. In one period of a maximal LFSR, 2^{n-1} runs occur (for example, a six bit LFSR will have 32 runs). Exactly 1/2 of these runs will be one bit long, 1/4 will be two bits long, up to a single run of zeroes $n - 1$ bits long, and a single run of ones n bits long. This distribution almost equals the statistical expectation value for a truly random sequence. However, the probability of finding exactly this distribution in a sample of a truly random sequence is rather low.
- LFSR output streams are deterministic. If you know the present state, you can predict the next state. This is not possible with truly random events.
- The output stream is reversible; an LFSR with mirrored taps will cycle through the output sequence in reverse order.

Applications

LFSRs can be implemented in hardware, and this makes them useful in applications that require very fast generation of a pseudo-random sequence, such as direct-sequence spread spectrum radio. LFSRs have also been used for generating an approximation of white noise in various programmable sound generators.

The Global Positioning System uses an LFSR to rapidly transmit a sequence that indicates high-precision relative time offsets.

Uses as counters

The repeating sequence of states of an LFSR allows it to be used as a clock divider, or as a counter when a non-binary sequence is acceptable as is often the case where computer index or framing locations need to be machine-readable. LFSR counters have simpler feedback logic than natural binary counters or Gray code counters, and therefore can operate at higher clock rates. However it is necessary to ensure that the LFSR never enters an all-zeros state, for example by presetting it at start-up to any other state in the sequence. The table of primitive polynomials shows how LFSRs can be arranged in Fibonacci or Galois form to give maximal periods. One can obtain any other period by adding to an LFSR that has a longer period some logic that shortens the sequence by skipping some states.

Uses in cryptography

LFSRs have long been used as pseudo-random number generators for use in stream ciphers (especially in military cryptography), due to the ease of construction from simple electromechanical or electronic circuits, long periods, and very uniformly distributed output streams. However, an LFSR is a linear system, leading to fairly easy cryptanalysis. For example, given a stretch of known plaintext and corresponding ciphertext, an attacker can intercept and recover a stretch of LFSR output stream used in the system described, and from that stretch of the output stream can construct an LFSR of minimal size that simulates the intended receiver by using the Berlekamp-Massey algorithm. This LFSR can then be fed the intercepted stretch of output stream to recover the remaining plaintext.

Three general methods are employed to reduce this problem in LFSR-based stream ciphers:

- Non-linear combination of several bits from the LFSR state;
- Non-linear combination of the output bits of two or more LFSRs
- Irregular clocking of the LFSR, as in the alternating step generator.

Important LFSR-based stream ciphers include A5/1 and A5/2, used in GSM cell phones, E0, used in Bluetooth, and the shrinking generator. The A5/2 cipher has been broken and both A5/1 and E0 have serious weaknesses.

Uses in digital broadcasting and communications

To prevent short repeating sequences (e.g., runs of 0's or 1's) from forming spectral lines that may complicate symbol tracking at the receiver or interfere with other transmissions, linear feedback registers are often used to "randomize" the transmitted bitstream. This randomization is removed at the receiver after demodulation. When the LFSR runs at the same rate as the transmitted symbol stream, this technique is referred to as scrambling. When the LFSR runs considerably faster than the symbol stream, expanding the bandwidth of the transmitted signal, this is direct-sequence spread spectrum.

Neither scheme should be confused with encryption or encipherment; scrambling and spreading with LFSRs do *not* protect the information from eavesdropping. They are instead used to produce equivalent streams that possess convenient engineering properties to allow for robust and efficient modulation and demodulation.

Digital broadcasting systems that use linear feedback registers:

- ATSC Standards (digital TV transmission system – North America)
- DAB (Digital Audio Broadcasting system – for radio)
- DVB-T (digital TV transmission system – Europe, Australia, parts of Asia)
- NICAM (digital audio system for television)

Other digital communications systems using LFSRs:

- IBS (INTELSAT business service)
- IDR (Intermediate Data Rate service)
- SDI (Serial Digital Interface transmission)
- Data transfer over PSTN (according to the ITU-T V-series recommendations)
- CDMA (Code Division Multiple Access) cellular telephony
- 100BASE-T2 "fast" Ethernet scrambles bits using an LFSR
- 1000BASE-T Ethernet, the most common form of Gigabit Ethernet, scrambles bits using an LFSR
- PCI Express 3.0
- USB 3.0
- IEEE 802.11a scrambles bits using an LFSR

The mathematics of a cyclic redundancy check, used to provide a quick check against transmission errors, are closely related to those of an LFSR.

Chapter 13

Program Counter and Register File

Program counter

The **program counter**, or PC (also called the **instruction pointer** to a seminal Intel instruction set, such as the 8080 or 4004, or **instruction address register**, or just part of the **instruction sequencer** in some computers) is a processor register that indicates where the computer is in its instruction sequence. Depending on the details of the particular computer, the PC holds either the **address of the instruction being executed**, or **the address of the next instruction to be executed**.

In most processors, the program counter is incremented automatically after fetching a program instruction, so that instructions are normally retrieved sequentially from memory, with certain instructions, such as branches, jumps and subroutine calls and returns, interrupting the sequence by placing a new value in the program counter.

Such jump instructions allow a new address to be chosen as the start of the next part of the flow of instructions from the memory. They allow new values to be loaded (written) into the program counter register. A subroutine call is achieved simply by reading the old contents of the program counter, before they are overwritten by a new value, and saving them somewhere in memory or in another register. A subroutine return is then achieved by writing the saved value back in to the program counter again.

Working of a simple program counter

The central processing unit (CPU) of a simple computer contains the hardware (control unit and ALU) that executes the instructions, as they are fetched from the memory unit. Most instruction cycles consist of the CPU sending an **address**, on the address bus, to the **memory unit**, which then responds by sending the contents of that memory location as **data**, on the data bus. (This is tied up with the idea of the stored-program computer in which executable instructions are stored alongside ordinary data in the **memory unit**, and handled identically by it). The Program Counter (PC) is just one of the many registers in the hardware of the CPU. It, like each of the other registers, consists of a bank of binary latches (a binary latch is also known as a flip-flop), with one flip-flop per bit in the integer that is to be stored (32 for a 32-bit CPU, for example). In the case of the PC, the integer represents the address in the **memory unit** that is to be fetched next.

Once the data (the instruction) has been received on the **data bus**, the PC is incremented. In some CPUs this is achieved by adding 000..001 to its contents, and latching the result into the register to be its new contents; on most CPUs, though, the PC is implemented as a register that is internally wired so that it counts up to the next value when a specific signal is applied to it externally. Such a register, in electronics, is referred to as a binary counter, and hence the origin of the term **program counter**.

The all-pervading nature of the program counter

The presence of the program counter in the CPU has far reaching consequences on our way of thinking when we program computers. Indeed, the program counter (or any equivalent block of hardware that serves the same purpose) is very much central to the von Neumann architecture.

The PC imposes a strict sequential ordering on the fetching of instructions from the memory unit (the flow of control), even where no sequentiality is implied by the algorithm itself (the von Neumann bottleneck). This is why research into possible models for parallel computing considered, at one point, other **non von Neumann** or dataflow models that did not use a program counter. For example, functional programming languages offered much hope at the high level, with combinatory logic at the assembler level. Even then, most of the researchers emulated this in the microcode of conventional computers (hence still involving a program counter in the hardware); but, in fact, combinators are so simple, they could, in principle, be implemented directly in the hardware without recourse to microcode or program counters at all.

In the end, though, the results of that research fed back, instead, into ways of improving the execution speed of conventional processors. Ways were found for organising out-of-order execution so as to extract the sequencing information that is implicit in the data. Also, the pipeline and very long instruction word organisations allowed the compiler to arrange for multiple calculations to be set off in parallel. At the start of each instruction execution, though, the instruction needs to be fetched from memory, and this is initiated by an **instruction fetch** cycle that gets the instructions, one at a time, as directed by the program counter.

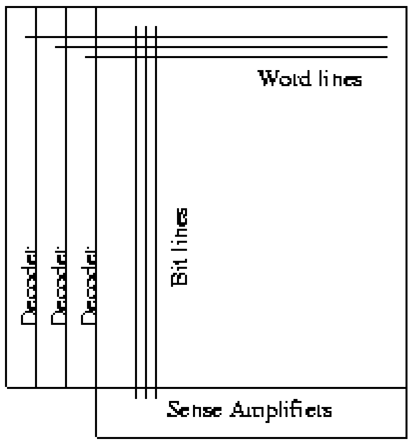
Even high level programming languages have the program-counter concept engrained deep down in their behavior. You need only to watch how a programmer develops or debugs a computer program to see evidence of this, with the programmer using a finger to point to successive lines in the program to model the steps of its execution. Indeed, a high level programming language is no less than the assembler language of a high level virtual machine -- a computer that would be too complex to be cost-effective to build directly in hardware, so is implemented instead using multiple shells of emulation (with the compiler or interpreter providing the higher levels, and the microcode providing the lower levels).

Register file

A **register file** is an array of processor registers in a central processing unit (CPU). Modern integrated circuit-based register files are usually implemented by way of fast static RAMs with multiple ports. Such RAMs are distinguished by having dedicated read and write ports, whereas ordinary multiported SRAMs will usually read and write through the same ports.

The instruction set architecture of a CPU will almost always define a set of registers which are used to stage data between memory and the functional units on the chip. In simpler CPUs, these *architectural registers* correspond one-for-one to the entries in a physical register file within the CPU. More complicated CPUs use register renaming, so that the mapping of which physical entry stores a particular architectural register changes dynamically during execution. The register file is part of the architecture and visible to the programmer, as opposed to the concept of transparent caches.

Implementation



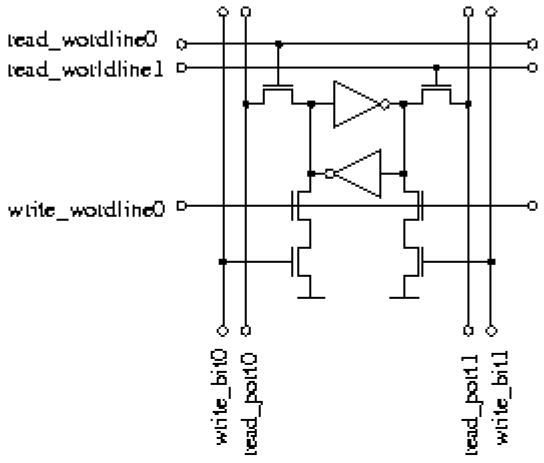
The usual layout convention is that a simple array is read out vertically. That is, a single word line, which runs horizontally, causes a row of bit cells to put their data on bit lines, which run vertically. Sense amps, which convert low-swing read bitlines into full-swing logic levels, are usually at the bottom (by convention). Larger register files are then sometimes constructed by tiling mirrored and rotated simple arrays.

Register files have one word line per entry per port, one bit line per bit of width per read port, and two bit lines per bit of width per write port. Each bit cell also has a Vdd and Vss. Therefore, the wire pitch area increases as the square of the number of ports, and the transistor area increases linearly. At some point, it may be smaller and/or faster to have multiple redundant register files, with smaller numbers of read ports, than a single register file with all the read ports. The MIPS R8000's integer unit, for example, had a 9 read 4 write port 32 entry 64-bit register file implemented in a 0.7 µm process, which could be seen when looking at the chip from arm's length.

Decoder

- The decoder is often broken into predecoder and decoder proper.
- The decoder is a series of AND gates that drive word lines.
- There is one decoder per read or write port. If the array has four read and two write ports, for example, it has 6 word lines per bit cell in the array, and six AND gates per row in the decoder. Note that the decoder has to be pitch matched to the array, which forces those AND gates to be wide and short

Array



A typical register file -- "triple-ported", able to read from 2 registers and write to 1 register simultaneously -- is made of bit cells like this one.

The basic scheme for a bit cell:

- State is stored in pair of inverters
- Data is read out by nmos transistor to a bit line.
- Data is written by shorting one side or the other to ground through a two-nmos stack.
- So: read ports take one transistor per bit cell, write ports take four!

Many optimizations are possible:

- Sharing lines between cells, for example, Vdd and Vss.
- Read bit lines are often precharged to something between Vdd and Vss.
- Read bit lines often swing only a fraction of the way to Vdd or Vss. A sense amplifier converts this small-swing signal into a full logic level. Small swing signals are faster because the bit line has little drive but a great deal of parasitic capacitance.
- Write bit lines may be braided, so that they couple equally to the nearby read bitlines. Because write bitlines are full swing, they can cause significant disturbances on read bitlines.

- If Vdd is a horizontal line, it can be switched off, by yet another decoder, if any of the write ports are writing that line during that cycle. This optimization increases the speed of the write.
- Techniques that reduce the energy used by register files are useful in low-power electronics

Microarchitecture

Most register files make no special provision to prevent multiple write ports from writing the same entry simultaneously. Instead, the instruction scheduling hardware ensures that only one instruction in any particular cycle writes a particular entry. If multiple instructions targeting the same register are issued, all but one have their write enables turned off.

The crossed inverters take some finite time to settle after a write operation, during which a read operation will either take longer or return garbage. It is common to have bypass multiplexors that bypass written data to the read ports when a simultaneous read and write to the same entry is commanded. These bypass multiplexors are often just part of a larger bypass network that forwards results that have not yet been committed between functional units.

The register file is usually pitch matched to the datapath that it serves. Pitch matching avoids having the many busses passing over the datapath turn corners, which would use a lot of area. But since every unit must have the same bit pitch, every unit in the datapath ends up with the bit pitch forced by the widest unit, which can waste area in the other units. Register files, because they have two wires per bit per write port, and because all the bit lines must contact the silicon at every bit cell, can often set the pitch of a datapath.

Area can sometimes be saved, on machines with multiple units in a datapath, by having two datapaths side-by-side, each of which has smaller bit pitch than a single datapath would have. This case usually forces multiple copies of a register file, one for each datapath.

The Alpha 21264 (EV6), for instance, had two copies of the integer register file, and took an extra cycle to propagate data between the two. The issue logic tried to reduce the number of operations forwarding data between the two. The MIPS R8000 floating-point unit had two copies of the floating-point register file, each with four write and four read ports, and wrote both copies at the same time.

Processors that do register renaming can arrange for each functional unit to write to a subset of the physical register file. This arrangement can eliminate the need for multiple write ports per bit cell, for large savings in area. The resulting register file, effectively a stack of register files with single write ports, then benefits from replication and subsetting the read ports. At the limit, this technique would place a stack of 1-write, 2-read regfiles at the inputs to each functional unit. Since regfiles with a small number of ports are often dominated by transistor area, it is best not to push this technique to this limit, but it is useful all the same.

The SPARC ISA defines register windows, in which the 5-bit architectural names of the registers actually point into a window on a much larger register file, with hundreds of entries. Implementing multiported register files with hundreds of entries requires a lot of area. The register window slides by 16 registers when moved, so that each architectural register name can refer to only a small number of registers in the larger array, e.g. architectural register r20 can only refer to physical registers #20, #36, #52, #68, #84, #100, #116, if there are just seven windows in the physical file.

To save area, some SPARC implementations implement a 32-entry register file, in which each cell has seven "bits". Only one is read and writeable through the external ports, but the contents of the bits can be rotated. A rotation accomplishes in a single cycle a movement of the register window. Because most of the wires accomplishing the state movement are local, tremendous bandwidth is possible with little power.

This same technique is used in the R10000 register renaming mapping file, which stores a 6-bit virtual register number for each of the physical registers. In the renaming file, the renaming state is checkpointed whenever a branch is taken, so that when a branch is detected to be mispredicted, the old renaming state can be recovered in a single cycle.

Chapter 14

Register Renaming

In computer architecture, **register renaming** refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations.

Problem definition

Programs are composed of instructions which operate on values. The instructions must name these values in order to distinguish them from one another. A typical instruction might say, add X and Y and put the result in Z. In this instruction, X, Y, and Z are the names of storage locations.

In order to have a compact instruction encoding, most processor instruction sets have a small set of special locations which can be directly named. For example, the x86 instruction set architecture has 8 integer registers, x86-64 has 16, many RISCs have 32, and IA-64 has 128. In smaller processors, the names of these locations correspond directly to elements of a register file.

Different instructions may take different amounts of time (e.g., CISC architecture). For instance, a processor may be able to execute hundreds of instructions while a single load from main memory is in progress. Shorter instructions executed while the load is outstanding will finish first, thus the instructions are finishing out of the original program order. Out-of-order execution has been used in most recent high-performance CPUs to achieve some of their speed gains.

Consider this piece of code running on an out-of-order CPU:

1. R1=M[1024]
2. R1=R1+2
3. M[1032]=R1
4. R1=M[2048]
5. R1=R1+4
6. M[2056]=R1

Instructions 4, 5, and 6 are independent of instructions 1, 2, and 3, but the processor cannot finish 4 until 3 is done, because 3 would then write the wrong value.

We can eliminate this restriction by changing the names of some of the registers:

1. R1=M[1024]	4. R2=M[2048]
2. R1=R1+2	5. R2=R2+4
3. M[1032]=R1	6. M[2056]=R2

Now instructions 4, 5, and 6 can be executed in parallel with instructions 1, 2, and 3, so that the program can be executed faster.

When possible, the compiler performs this renaming. The compiler is constrained in many ways, primarily by the finite number of register names in the instruction set. Many high performance CPUs have more physical registers than may be named directly in the instruction set, so they rename registers in hardware to achieve additional parallelism.

Data hazards

When more than one instruction references a particular location for an operand, either reading it (as an input) or writing it (as an output), executing those instructions in an order different from the original program order can lead to three kinds of data hazards:

Read-after-write (RAW)

A read from a register or memory location must return the value placed there by the last write in program order, not some other write. This is referred to as a **true dependency** or **flow dependency**, and requires the instructions to execute in program order.

Write-after-write (WAW)

Successive writes to a particular register or memory location must leave that location containing the result of the second write. This can be resolved by **squashing** (synonyms: cancelling, annulling, mooting) the first write if necessary. WAW dependencies are also known as **output dependencies**.

Write-after-read (WAR)

A read from a register or memory location must return the last prior value written to that location, and not one written programmatically after the read. This is the sort of **false dependency** that can be resolved by renaming. WAR dependencies are also known as **anti-dependencies**.

Instead of delaying the write until all reads are completed, two copies of the location can be maintained, the old value and the new value. Reads that precede, in program order, the write of the new value can be provided with the old value, even while other reads that follow the write are provided with the new value. The false dependency is broken and additional opportunities for out-of-order execution are created. When all reads needing the old value have been satisfied, it can be discarded. This is the essential concept behind register renaming.

Anything that is read and written can be renamed. While the general-purpose and floating-point registers are discussed the most, flag and status registers or even individual status bits are commonly renamed as well.

Memory locations can also be renamed, although it is not commonly done to the extent practised in register renaming. The Transmeta Crusoe processor's gated store buffer is a form of memory renaming.

If programs refrained from reusing registers immediately, there would be no need for register renaming. Some instruction sets (e.g., IA-64) specify very large numbers of registers for specifically this reason. There are limitations to this approach:

- It is very difficult for the compiler to avoid reusing registers without large code size increases. In loops, for instance, successive iterations would have to use different registers, which requires replicating the code in a process called loop unrolling
- Large numbers of registers require lots of bits to specify those registers, making the code size increase.
- Many instruction sets historically specified smaller numbers of registers and cannot be changed now.

Code size increases are important because when the program code is larger, the instruction cache misses more often and the processor stalls waiting for new instructions.

Architectural vs physical registers

Machine language programs specify reads and writes to a limited set of registers specified by the instruction set architecture (ISA). For instance, the Alpha ISA specifies 32 integer registers, each 64 bits wide, and 32 floating-point registers, each 64 bits wide. These are the **architectural** registers. Programs written for processors running the Alpha instruction set will specify operations reading and writing those 64 registers. If a programmer stops the program in a debugger, she or he can observe the contents of these 64 registers (and a few status registers) to determine the progress of the machine.

One particular processor which implements this ISA, the Alpha 21264, has 80 integer and 72 floating-point **physical** registers. There are, on an Alpha 21264 chip, 80 physically separate locations which can store the results of integer operations, and 72 locations which can store the results of floating point operations. (In fact, there are even more locations than that, but those extra locations are not germane to the register renaming operation.)

Below are described two styles of register renaming, distinguished by the circuit which holds data ready for an execution unit.

In all renaming schemes, the machine converts the architectural registers referenced in the instruction stream into tags. Where the architectural registers might be specified by 3 to 5 bits, the tags are usually a 6 to 8 bit number. The rename file must have a read port for every input of every instruction renamed every cycle, and a write port for every output of every instruction renamed every cycle. Because the size of a register file generally grows as the square of the number of ports, the rename file is usually physically large and consumes significant power.

In the **tag-indexed register file** style, there is one large register file for data values, containing one register for every tag. For example, if the machine has 80 physical registers, then it would use 7 bit tags. 48 of the possible tag values in this case are unused.

In this style, when an instruction is issued to an execution unit, the tags of the source registers are sent to the physical register file, where the values corresponding to those tags are read and sent to the execution unit.

In the **reservation station** style, there are many small associative register files, usually one at the inputs to each execution unit. Each operand of each instruction in an issue queue has a place for a value in one of these register files.

In this style, when an instruction is issued to an execution unit, the register file entries corresponding to the issue queue entry are read and forwarded to the execution unit.

Architectural Register File or Retirement Register File (RRF)
: The committed register state of the machine. RAM indexed by logical register number. Typically written into as results are retired or committed out of a reorder buffer.

Future File
: The most speculative register state of the machine. RAM indexed by logical register number.

Active Register File
: The Intel P6 group's term for Future File.

History Buffer
: Typically used in combination with a future file. Contains the "old" values of registers that have been overwritten. If the producer is still in flight it may be RAM indexed by history buffer number. After a branch misprediction must use results from the history buffer—either they are copied, or the future file lookup is disabled and the history buffer is CAM indexed by logical register number.

Reorder Buffer (ROB)
: A structure that is sequentially (circularly) indexed on a per-operation basis, for instructions in flight. It differs from a history buffer because the reorder buffer typically comes after the future file (if it exists) and before the architectural register file.

Reorder buffers come in data-less and data-ful versions.

In Willamette's ROB, the ROB entries point to registers in the physical register file (PRF), and also contain other book keeping. This was also the first Out of Order design done by Andy Glew, at Illinois with HaRRM.

P6's ROB, the ROB entries contain data; there is no separate PRF. Data values from the ROB are copied from the ROB to the RRF at retirement.

One small detail: if there is temporal locality in ROB entries (i.e., if instructions close together in the Von Neuman instruction sequence write back close together in time, it may be possible to perform write combining on ROB entries and so have fewer ports than a separate ROB/PRF would). It is not clear if it makes a difference, since a PRF should be banked.

ROBs usually don't have associative logic, and certainly none of the ROBs designed by Andy Glew have CAMs. Keith Diefendorff insisted that ROBs have complex associative logic for many years. The first ROB proposal may have had CAMs.

Details: tag-indexed register file

This is the renaming style used in the MIPS R10000, the Alpha 21264, and in the FP section of the AMD Athlon.

In the renaming stage, every architectural register referenced (for read or write) is looked up in an architecturally-indexed **remap file**. This file returns a tag and a ready bit. The tag is non-ready if there is a queued instruction which will write to it that has not yet executed. For read operands, this tag takes the place of the architectural register in the instruction. For every register write, a new tag is pulled from a **free tag FIFO**, and a new mapping is written into the remap file, so that future instructions reading the architectural register will refer to this new tag. The tag is marked as unready, because the instruction has not yet executed. The previous physical register allocated for that architectural register is saved with the instruction in the **reorder buffer**, which is a FIFO that holds the instructions in program order between the decode and graduation stages.

The instructions are then placed in various **issue queues**.

As instructions are executed, the tags for their results are broadcast, and the issue queues match these tags against the tags of their non-ready source operands. A match means that the operand is ready. The remap file also matches these tags, so that it can mark the corresponding physical registers as ready.

When all the operands to an instruction in an issue queue are ready, that instruction is ready to issue. The issue queues pick ready instructions to send to the various functional units each cycle. Non-ready instructions stay in the issue queues. This unordered removal of instructions from the issue queues is one of the things that makes them large and use lots of power.

Issued instructions read from a tag-indexed physical register file (bypassing just-broadcast operands), then execute.

Execution results are written to tag-indexed physical register file, as well as broadcast to the bypass network preceding each functional unit.

Graduation puts the previous tag for the written architectural register into the free queue so that it can be reused for a newly decoded instruction.

An exception or branch misprediction causes the remap file to back up to the remap state at last valid instruction via combination of state snapshots and cycling through the previous tags in the in-order pre-graduation queue. Since this mechanism is required, and since it can recover any remap state (not just the state before the instruction currently being graduated), branch mispredictions can be handled before the branch reaches graduation, potentially hiding the branch misprediction latency.

Details: reservation stations

This is the style used in the integer section of the AMD K7 and K8 designs.

In the renaming stage, every architectural register referenced for reads is looked up in both the architecturally-indexed **future file** and the rename file. The future file read gives the value of that register, if there is no outstanding instruction yet to write to it (i.e., it's ready). When the instruction is placed in an issue queue, the values read from the future file are written into the corresponding entries in the reservation stations. Register writes in the instruction cause a new, non-ready tag to be written into the rename file. The tag number is usually serially allocated in instruction order—no free tag FIFO is necessary.

Just as with the tag-indexed scheme, the issue queues wait for non-ready operands to see matching tag broadcasts. Unlike the tag-indexed scheme, matching tags cause the corresponding broadcast value to be written into the issue queue entry's reservation station.

Issued instructions read their arguments from the reservation station, bypass just-broadcast operands, and then execute. As mentioned earlier, the reservation station register files are usually small, with perhaps eight entries.

Execution results are written to the reorder buffer, to the reservation stations (if the issue queue entry has a matching tag), and to the future file if this is the last instruction to target that architectural register (in which case register is marked ready).

Graduation copies the value from the reorder buffer into the architectural register file. The sole use of the architectural register file is to recover from exceptions and branch mispredictions.

Exceptions and branch mispredictions, recognised at graduation, cause the architectural file to be copied to the future file, and all registers marked as ready in the rename file. There is usually no way to reconstruct the state of the future file for some instruction intermediate between decode and graduation, so there is usually no way to do early recovery from branch mispredictions.

Comparison between the schemes

In both schemes, instructions are inserted in-order into the issue queues, but are removed out-of-order. If the queues do not collapse empty slots, then they will either have many unused entries, or require some sort of variable priority encoding for when multiple instructions are simultaneously ready to go. Queues that collapse holes have simpler priority encoding, but require simple but large circuitry to advance instructions through the queue.

Reservation stations have better latency from rename to execute, because the rename stage finds the register values directly, rather than finding the physical register number, and then using that to find the value. This latency shows up as a component of the branch mispredict latency.

Reservation stations also have better latency from instruction issue to execution, because each local register file is smaller than the large central file of the tag-indexed scheme. Tag generation and exception processing are also simpler in the reservation station scheme, as discussed below.

The physical register files used by reservation stations usually collapse unused entries in parallel with the issue queue they serve, which makes these register files larger in aggregate, and burn more power, and more complicated than the simpler register files used in a tag-indexed scheme. Worse yet, every entry in each reservation station can be written by every result bus, so that a reservation-station machine with, e.g., 8 issue queue entries per functional unit will typically have 9 times as many bypass networks as an equivalent tag-indexed machine. Result forwarding thus takes much more power and area than in a tag-indexed design.

Furthermore, the reservation station scheme has four places (Future File, Reservation Station, Reorder Buffer and Architectural File) where a result value can be stored, where the tag-indexed scheme has just one (the physical register file). Because the results from the functional units, broadcast to all these storage locations, must reach a much larger number of locations in the machine than in the tag-indexed scheme, this function consumes more power, area, and time. Still, in machines equipped with very accurate branch prediction schemes and if execute latencies are a major concern, reservation stations can work remarkably well.

History

The IBM System/360 Model 91 was an early machine that supported out-of-order execution of instructions; it used the Tomasulo algorithm, which uses register renaming.

The POWER1 is the first microprocessor that used register renaming and out-of-order execution in 1990.

The original R10000 design had neither collapsing issue queues nor variable priority encoding, and suffered starvation problems as a result—the oldest instruction in the queue would sometimes not be issued until both instruction decode stopped completely for lack of rename registers, and every other instruction had been issued. Later revisions of the design starting with the R12000 used a partially variable priority encoder to mitigate this problem.

Early out-of-order machines did not separate the renaming and ROB/PRF storage functions. For that matter, some of the earliest, such as Sohi's RUU or the Metaflow DCAF, combined scheduling, renaming, and storage all in the same structure.

Most modern machines do renaming by RAM indexing a map table with the logical register number. E.g., P6 did this; future files do this, and have data storage in the same structure.

However, earlier machines used content-addressable memory (a type of hardware which provides the functionality of an associative array) in the renamer. E.g., the HPSM RAT, or Register Alias Table, essentially used a CAM on the logical register number in combination with different versions of the register.

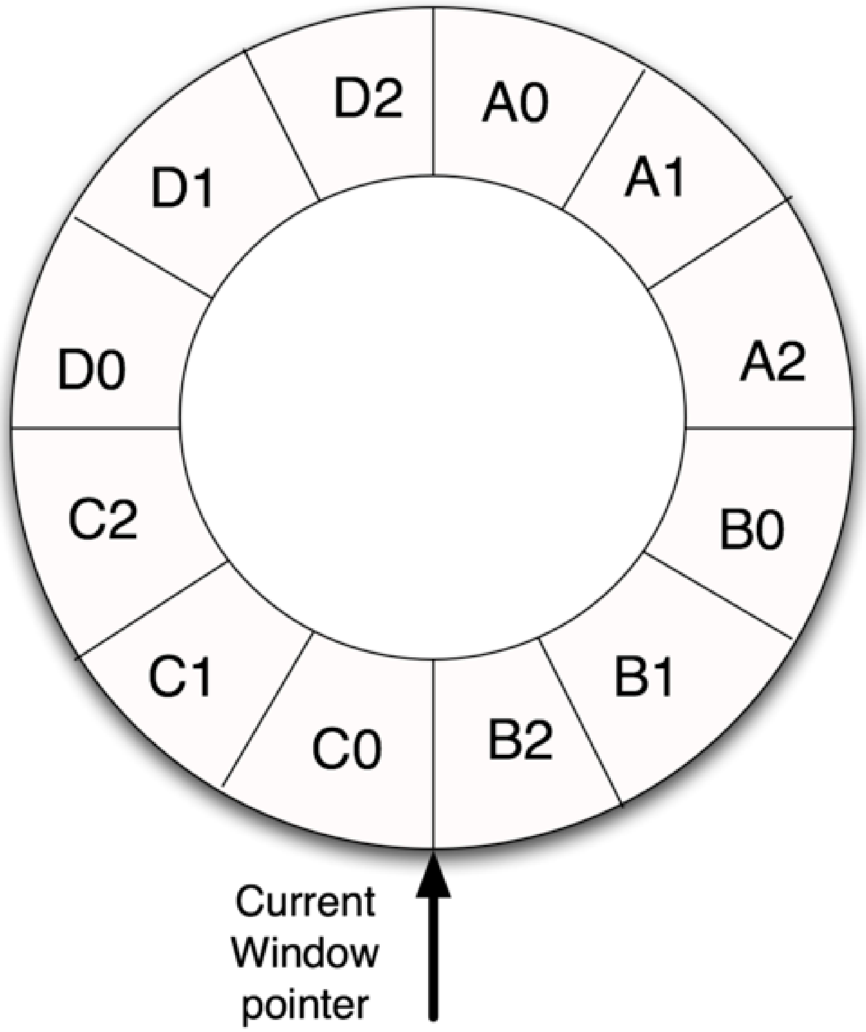
In many ways, the story of out-of-order microarchitecture has been how these CAMs have been progressively eliminated. Small CAMs are useful; large CAMs are impractical.

The P6 microarchitecture was the first Intel based processor that implemented both out-of-order execution and register renaming. The P6 microarchitecture manifested in Pentium Pro, Pentium II, Pentium III, Pentium M, Core, and Core 2 microprocessors.

Chapter 15

Register Window and Shift Register

Register window



Example of a 4-window register window system

In computer engineering, the use of **register windows** is a technique to improve the performance of a particularly common operation, the procedure call. This was one of the main design features of the original Berkeley RISC design, which would later be commercialized as the SPARC, AMD 29000, and Intel i960.

Most CPU designs include a small amount of very high-speed memory known as registers. Registers are used by the CPU in order to hold temporary values while working on longer strings of instructions. Considerable performance can be added to a design with more registers, however, since the registers are a visible piece of the CPU's instruction set, the number cannot typically be changed after the design has been released.

While registers are almost a universal solution to performance, they do have a drawback. Different parts of a computer program all use their own temporary values, and therefore compete for the use of the registers. Since a good understanding of the nature of program flow at runtime is very difficult, there is no easy way for the developer to know in advance how many registers they should use, and how many to leave aside for other parts of the program. In general these sorts of considerations are ignored, and the developers, and more likely, the compilers they use, attempt to use all the registers visible to them. (In the case of processors with very few registers to begin with, this is also the only reasonable course of action.)

This is where register windows become useful. Since every part of a program wants registers for its own use, it makes sense to provide several sets of registers for the different parts of the program. Of course if these registers were visible, there would simply be more registers to compete over, the "trick" is to make them invisible. This is actually somewhat simpler than it might sound; the movement from one part of the program to another during a procedure call is easily "seen", it is accomplished by one of a small number of instructions and ends with one of a similarly small set. In the Berkeley design, these calls would cause a new set of registers to be "swapped in" at that point, or marked as "dead" (or "reusable") when the call ends.

In the Berkeley RISC design, only eight registers were visible to the programs, out of a total of 64. The complete set of registers was known as the register file, and any particular set of eight as a **window**. The file allowed up to eight procedure calls to have their own register sets. As long as the program did not call down chains longer than eight calls deep, the registers never had to be *spilled* (saved out to main memory or cache), a terribly slow process compared to register access. For many programs a chain of six is as deep as the program will go.

By comparison the Sun Microsystems SPARC architecture provides simultaneous visibility into four sets of eight registers each. Three sets of eight registers each are "windowed". Eight registers (i0 through i7) form the input registers to the current procedure level. Eight registers (L0 through L7) are local to the current procedure level, and eight registers (o0 through o7) are the outputs from the current procedure level to the next level called. When a procedure is called, the register window shifts by sixteen registers, hiding the old input registers and old local registers and making the old output registers the new input registers. The common registers (old output registers and new input registers) are used for parameter passing. Finally, eight registers (g0 through g7) are globally visible to all procedure levels.

The AMD 29000 improved the design by allowing the windows to be of variable size, which helps utilization in the common case where fewer than eight registers are needed for a call. It also separated the registers into a global set of 64, and an additional 128 for the windows.

Register windows also provide an easy upgrade path. Since the additional registers are invisible to the programs, additional windows can be added at any time. For instance, the use of object-oriented programming often results in a greater number of "smaller" calls, which can be accommodated by increasing the windows from eight to sixteen for instance. This was the approach used in the SPARC, which has included more register windows with newer generations of the architecture. The end result is fewer slow register window *spill* and *fill* operations because the register windows overflow less often.

Register windows are not the only way to improve register performance. The group at Stanford University designing the MIPS architecture saw the Berkeley work and decided that the problem was not a shortage of registers, but poor utilization of the existing ones. They instead invested more time in their compiler's register allocation, making sure it wisely used the larger set available in the MIPS instruction set. This resulted in reduced complexity of the chip, with one half the total number of registers, while offering potentially higher performance in those cases where a single procedure could make use of the larger visible register space. In the end, with modern compilers, the MIPS design makes better use of its register space even during procedure calls.

Shift register

In digital circuits, a **shift register** is a cascade of flip flops, sharing the same clock, which has the output of any one but the last flip-flop connected to the "data" input of the next one in the chain, resulting in a circuit that shifts by one position the one-dimensional "bit array" stored in it, *shifting in* the data present at its input and *shifting out* the last bit in the array, when enabled to do so by a transition of the clock input. More generally, a **shift register** may be multidimensional, such that its "data in" input and stage outputs are themselves bit arrays: this is implemented simply by running several shift registers of the same bit-length in parallel.

Shift registers can have both parallel and serial inputs and outputs. These are often configured as **serial-in, parallel-out** (SIPO) or as **parallel-in, serial-out** (PISO). There are also types that have both serial and parallel input and types with serial and parallel output. There are also **bi-directional** shift registers which allow shifting in both directions: L→R or R→L. The serial input and last output of a shift register can also be connected together to create a **circular shift register**.

Serial-in, serial-out (SISO)

Destructive readout

0	0	0	0
1	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1
0	1	1	0
0	0	1	1
0	0	0	1
0	0	0	0

These are the simplest kind of shift registers. The data string is presented at 'Data In', and is shifted right one stage each time 'Data Advance' is brought high. At each advance, the bit on the far left (i.e. 'Data In') is shifted into the first flip-flop's output. The bit on the far right (i.e. 'Data Out') is shifted out and lost.

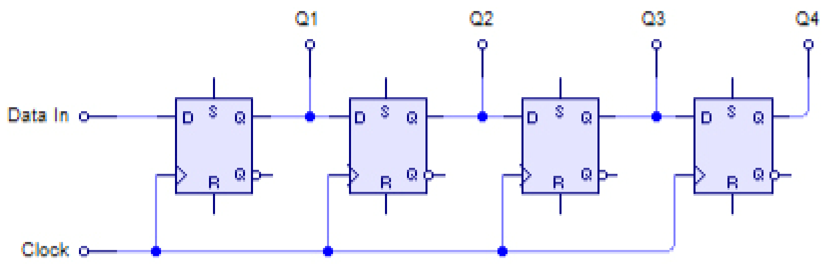
The data are stored after each flip-flop on the 'Q' output, so there are four storage 'slots' available in this arrangement, hence it is a 4-Bit Register. To give an idea of the shifting pattern, imagine that the register holds 0000 (so all storage slots are empty). As 'Data In' presents 1,0,1,1,0,0,0,0 (in that order, with a pulse at 'Data Advance' each time — this is called clocking or strobing) to the register, this is the result. The left hand column corresponds to the left-most flip-flop's output pin, and so on.

So the serial output of the entire register is 10110000 . As you can see if we were to continue to input data, we would get exactly what was put in, but offset by four 'Data Advance' cycles. This arrangement is the hardware equivalent of a queue. Also, at any time, the whole register can be set to zero by bringing the reset (R) pins high.

This arrangement performs *destructive readout* - each datum is lost once it has been shifted out of the right-most bit.

Serial-in, parallel-out (SIPO)

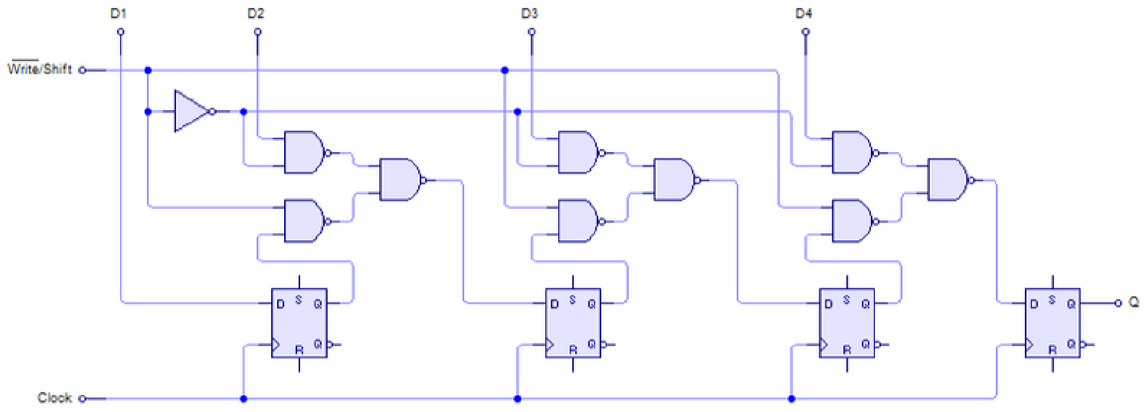
This configuration allows conversion from serial to parallel format. Data is input serially, as described in the SISO section above. Once the data has been input, it may be either read off at each output simultaneously, or it can be shifted out and replaced.



4-Bit SIPO Shift Register

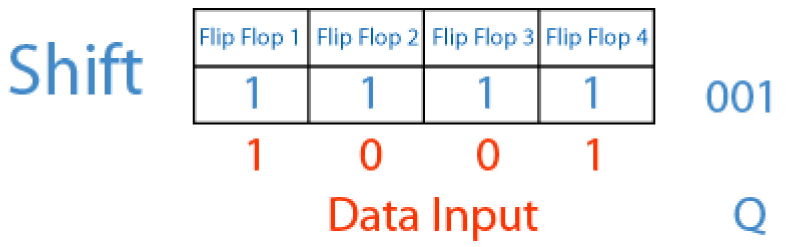
Parallel-in, serial-out (PISO)

This configuration has the data input on lines D1 through D4 in parallel format. To write the data to the register, the Write/Shift control line must be held LOW. To shift the data, the W/S control line is brought HIGH and the registers are clocked. The arrangement now acts as a PISO shift register, with D1 as the Data Input. However, as long as the number of clock cycles is not more than the length of the data-string, the Data Output, Q, will be the parallel data read off in order.



4-Bit PISO Shift Register

The image below shows the write/shift sequence, including the internal state of the shift register.



Flip Flop 1	Flip Flop 2	Flip Flop 3	Flip Flop 4
1	1	1	1

Shift
001
1 0 0 1
Data Input
Q

Uses

One of the most common uses of a shift register is to convert between serial and parallel interfaces. This is useful as many circuits work on groups of bits in parallel, but serial interfaces are simpler to construct. Shift registers can be used as simple delay circuits. Several bidirectional shift registers could also be connected in parallel for a hardware implementation of a stack.

SIPO registers are commonly attached to the output of microprocessors when more output pins are required than are available. This allows several binary devices to be controlled using only two or three pins - the devices in question are attached to the parallel outputs of the shift register, then the desired state of all those devices can be sent out of the microprocessor using a single serial connection. Similarly, PISO configurations are commonly used to add more binary inputs to a microprocessor than are available - each binary input (i.e. a switch or button, or more complicated circuitry designed to output high when active) is attached to a parallel input of the shift register, then the data is sent back via serial to the microprocessor using several fewer lines than originally required.

Shift registers can be used also as pulse extenders. Compared to monostable multivibrators, the timing has no dependency on component values, however it requires external clock and the timing accuracy is limited by a granularity of this clock. Example: Ronja Twister, where five 74164 shift registers create the core of the timing logic this way (schematic).

In early computers, shift registers were used to handle data processing: two numbers to be added were stored in two shift registers and clocked out into an arithmetic and logic unit (ALU) with the result being fed back to the input of one of the shift registers (the accumulator) which was one bit longer since binary addition can only result in an answer that is the same size or one bit longer.

Many computer languages include instructions to 'shift right' and 'shift left' the data in a register, effectively dividing by two or multiplying by two for each place shifted.

Very large serial-in serial-out shift registers (thousands of bits in size) were used in a similar manner to the earlier delay line memory in some devices built in the early 1970s. Such memories were sometimes called *circulating memory*. For example, the DataPoint 3300 terminal stored its display of 25 rows of 72 columns of upper-case characters using fifty-four 200-bit shift registers, arranged in six tracks of nine packs each, providing storage for 1800 six-bit characters. The shift register design meant that scrolling the terminal display could be accomplished by simply pausing the display output to skip one line of characters.

History

One of the first known examples of a shift register was in the Colossus, a code-breaking machine of the 1940s. It was a five-stage device built of vacuum tubes and thyratrons.

Chapter 16

Word (Computing)

In computing, **word** is a term for the natural unit of data used by a particular computer design. A word is simply a fixed sized group of bits that are handled together by the system. The number of bits in a word (the **word size** or **word length**) is an important characteristic of computer architecture.

The size of a word is reflected in many aspects of a computer's structure and operation; the majority of the registers in the computer are usually word sized and the amount of data transferred between the processing part computer and the memory system, in a single operation, is most often a word. The largest possible address size, used to designate a location in memory, is typically a hardware word (in other words, the full-sized natural word of the processor, as opposed to any other definition used on the platform).

Modern computers usually have a word size of 16, 32 or 64 bits but many other sizes have been used, including 8, 9, 12, 18, 24, 36, 39, 40, 48 and 60 bits. The slab is an example of a system with an earlier word size. Several of the earliest computers used the decimal base rather than binary, typically having a word size of 10 or 12 decimal digits, and some early computers had no fixed word length at all.

The size of a word can sometimes differ from the expected due to backward compatibility with earlier computers. If multiple compatible variations or a family of processors share a common architecture and instruction set but differ in their word sizes, their documentation and software may become notationally complex to accommodate the difference.

Uses of words

Depending on how a computer is organized, units of the word size may be used for:

- **Integer numbers**: Holders for integer numerical values may be available in one or in several different sizes, but one of the sizes available will almost always be the word. The other sizes, if any, are likely to be multiples or fractions of the word size. The smaller sizes are normally used only for efficient use of memory; when loaded into the processor, their values usually go into a larger, word sized holder.

- **Floating point numbers**: Holders for floating point numerical values are typically either a word or a multiple of a word.

- **Addresses**: Holders for memory addresses must be of a size capable of expressing the needed range of values but not be excessively large, so often the size used is the word though it can also be a multiple or fraction of the word size.

- **Registers**: Processor registers are designed with a size appropriate for the type of data they hold, e.g. integers, floating point numbers or addresses. Many computer architectures use "general purpose" registers that can hold any of several types of data, these registers must be sized to hold the largest of the types, historically this is the word size of the architecture though increasingly special purpose, larger, registers have been added to deal with newer types.

- **Memory-processor transfer**: When the processor reads from the memory subsystem into a register or writes a register's value to memory, the amount of data transferred is often a word. In simple memory subsystems, the word is transferred over the memory data bus, which typically has a width of a word or half word. In memory subsystems that use caches, the word-sized transfer is the one between the processor and the first level of cache; at lower levels of the memory hierarchy larger transfers (which are a multiple of the word size) are normally used.

- **Unit of address resolution**: In a given architecture, successive address values designate successive units of memory; this unit is the unit of address resolution. In most computers, the unit is either a character (e.g. a byte) or a word. (A few computers have used bit resolution.) If the unit is a word, then a larger amount of memory can be accessed using an address of a given size. On the other hand, if the unit is a byte, then individual characters can be addressed (i.e. selected during the memory operation).

- **Instructions**: Machine instructions are normally fractions or multiples of the architecture's word size. This is a natural choice since instructions and data usually share the same memory subsystem. In Harvard architectures the word sizes of instructions and data need not be related.

Word size choice

When a computer architecture is designed, the choice of a word size is of substantial importance. There are design considerations which encourage particular bit-group sizes for particular uses (e.g. for addresses), and these considerations point to different sizes for different uses. However, considerations of economy in design strongly push for one size, or a very few sizes related by multiples or fractions (submultiples) to a primary size. That preferred size becomes the word size of the architecture.

Character size is one of the influences on a choice of word size. Before the mid-1960s, characters were most often stored in six bits; this allowed no more than 64 characters, so alphabetics were limited to upper case. Since it is efficient in time and space to have the word size be a multiple of the character size, word sizes in this period were usually multiples of 6 bits (in binary machines). A common choice then was the 36-bit word, which is also a good size for the numeric properties of a floating point format.

After the introduction of the IBM System/360 design which used eight-bit characters and supported lower-case letters, the standard size of a character (or more accurately, a byte) became eight bits. Word sizes thereafter were naturally multiples of eight bits, with 16, 32, and 64 bits being commonly used.

Variable word architectures

Early machine designs included some that used what is often termed a *variable word length*. In this type of organization, a numeric operand had no fixed length but rather its end was detected when a character with a special marking was encountered. Such machines often used binary coded decimal for numbers. This class of machines included the IBM 702, IBM 705, IBM 7080, IBM 7010, UNIVAC 1050, IBM 1401, and IBM 1620.

Most of these machines work on one unit of memory at a time and since each instruction or datum is several units long, each instruction takes several cycles just to access memory. These machines are often quite slow because of this. For example, instruction fetches on an IBM 1620 Model I take 8 cycles just to read the 12 digits of the instruction (the Model II reduced this to 6 cycles, but reduced the fetch times to 4 cycles if both address fields were not needed by the instruction). Instruction execution took a completely variable number of cycles, depending on the size of the operands.

Word and byte addressing

The memory model of an architecture is strongly influenced by the word size. In particular, the resolution of a memory address, that is, the smallest unit that can be designated by an address, has often been chosen to be the word. In this approach, address values which differ by one designate adjacent memory words. This is natural in machines which deal almost always in word (or multiple-word) units, and has the advantage of allowing instructions to use minimally-sized fields to contain addresses, which can permit a smaller instruction size or a larger variety of instructions.

When byte processing is to be a significant part of the workload, it is usually more advantageous to use the byte, rather than the word, as the unit of address resolution. This allows an arbitrary character within a character string to be addressed straightforwardly. A word can still be addressed, but the address to be used requires a few more bits than the word-resolution alternative. The word size needs to be an integral multiple of the character size in this organization. This addressing approach was used in the IBM 360, and has been the most common approach in machines designed since then.

The power of two

Different amounts of memory are used to store data values with different degrees of precision. The commonly used sizes are usually a power of two multiple of the unit of address resolution (byte or word). Converting the index of an item in an array into the address of the item then requires only a shift operation rather than a multiplication. In some cases this relationship can also avoid the use of division operations. As a result, most modern computer designs have word sizes (and other operand sizes) that are a power of two times the size of a byte.

Size families

As computer designs have grown more complex, the central importance of a single word size to an architecture has decreased. Although more capable hardware can use a wider variety of sizes of data, market forces exert pressure to maintain backward compatibility while extending processor capability. As a result, what might have been the central word size in a fresh design has to coexist as an alternative size to the original word size in a backward compatible design. The original word size remains available in future designs, forming the basis of a size family.

In the mid-1970s, DEC designed the VAX to be a successor of the PDP-11. They used "word" for a 16-bit quantity while they used the term "longword" to refer to a 32-bit quantity. This is in contrast to earlier machines, where the natural unit of addressing memory would be called a *word*, while a quantity that is one half a word would be called, if anything, a *halfword*. In fitting with this scheme, a VAX "quadword" is 64 bits.

Another example is the x86 family, of which processors of three different word lengths (16-bit, later 32- and 64-bit) have been released. This leads to a common source of confusion because software is routinely ported from one word-length to the next. As a result, some APIs and documentation define or refer to an older (and thus shorter) word-length than software may be compiled for. For example, Microsoft's Windows API maintains the programming language definition of **WORD** as 16 bits, despite the fact that the API may be used on a 32- or 64-bit x86 processor, where the word size would be 32 or 64 bits, respectively. Data structures containing such 32- or 64-bit words are referred to as **DWORD** and **QWORD** respectively. A similar phenomenon has developed in Intel's x86 assembly language – because of backward compatibility in the instruction set, some instruction mnemonics carry "d" or "q" identifiers denoting "double-", "quad-" or "double-quad-", which are in terms of the architecture's original 16-bit word size.

Table of word sizes

key: **b: bits, d: decimal digits, *w*: word size of architecture, *n*: variable size**

Year	Computer Architecture	Word Size *w*	Integer Sizes	Floating Point Sizes	Instruction Sizes	Unit of Address Resolution	Char Size
1837	Babbage Analytical engine	50 d	*w*	—	5 different cards were used for different functions, exact size of cards not known	*w*	—
1941	Zuse Z3	22 b	—	*w*	8 b	*w*	—
1942	ABC	50 b	*w*	—	—	—	—
1944	Harvard Mark I	23 d	*w*	—	24 b	—	—
1946 (1948) {1953}	ENIAC (w/ Panel #16) {w/ Panel #26}	10 d	*w*, 2*w* (*w*) {*w*}	—	— (2*d*, 4*d*, 6*d*, 8*d*) {2*d*, 4*d*, 6*d*, 8*d*}	— — {*w*}	—
1951	UNIVAC I	12 d	*w*	—	½*w*	*w*	1 d
1952	IAS machine	40 b	*w*	—	½*w*	*w*	5 b
1952	Fast Universal Digital Computer M-2	34 b	*w?*	*w*	34 b = 4 b opcode plus 3× 10b address	10 b	—
1952	IBM 701	36 b	½*w*, *w*	—	½*w*	½*w*, *w*	6 b
1952	UNIVAC 60	*n* d	1*d*, ... 10*d*	—	—	—	2*d*, 3*d*
1953	IBM 702	*n* d	0*d*, ... 511*d*	—	5*d*	*d*	1 d
1953	UNIVAC 120	*n* d	1*d*, ... 10*d*	—	—	—	2*d*, 3*d*
1954 (1955)	IBM 650 (w/IBM 653)	10 d	*w*	— (*w*)	*w*	*w*	2 d
1954	IBM 704	36 b	*w*	*w*	*w*	*w*	6 b
1954	IBM 705	*n* d	0*d*, ... 255*d*	—	5*d*	*d*	1 d
1954	IBM NORC	16 d	*w*	*w*, 2*w*	*w*	*w*	—
1956	IBM 305	*n* d	1*d*, ... 100*d*	—	10*d*	*d*	1 d
1957	Autonetics Recomp I	40 b	*w*, 79 b, 8*d*, 15*d*	—	½*w*	½*w*, *w*	5 b

1958	UNIVAC II	12 d	*w*	—	½*w*	*w*	1 d
1958	SAGE	32 b	½*w*	—	*w*	*w*	6 b
1958	Autonetics Recomp II	40 b	*w*, 79 b, 8*d*, 15*d*	2*w*	½*w*	½*w*, *w*	5 b
1959	IBM 1401	*n* d	1*d*, ...	—	*d*, 2*d*, 4*d*, 5*d*, 7*d*, 8*d*	*d*	1 d
1959 (TBD)	IBM 1620	*n* d	2*d*, ...	— (4*d*, ... 102*d*)	12*d*	*d*	2 d
1960	LARC	12 d	*w*, 2*w*	*w*, 2*w*	*w*	*w*	2 d
1960	CDC 1604	48 b	*w*	*w*	½*w*	*w*	6 b
1960	IBM 1410	*n* d	1*d*, ...	—	*d*, 2*d*, 6*d*, 7*d*, 11*d*, 12*d*	*d*	1 d
1960	IBM 7070	10 d	*w*	*w*	*w*	*w*, *d*	2 d
1960	PDP-1	18 b	*w*	—	*w*	*w*	6 b
1961	IBM 7030 (Stretch)	64 b	1*b*, ... 64*b*, 1*d*, ... 16*d*	*w*	½*w*, *w*	b, ½*w*, *w*	1 b, ... 8 b
1961	IBM 7080	*n* d	0*d*, ... 255*d*	—	5*d*	*d*	1 d
1962	UNIVAC III	25 b, 6 d	*w*, 2*w*, 3*w*, 4*w*	—	*w*	*w*	6 b
1962	Autonetics D-17B Minuteman I Guidance Computer	27 b	11 b, 24 b	—	24 b	*w*	—
1962	UNIVAC 1107	36 b	$^1/_6$*w*, ⅓*w*, ½*w*, *w*	*w*	*w*	*w*	6 b
1962	IBM 7010	*n* d	1*d*, ...	—	*d*, 2*d*, 6*d*, 7*d*, 11*d*, 12*d*	*d*	1 d
1962	IBM 7094	36 b	*w*	*w*, 2*w*	*w*	*w*	6 b
1963	Gemini Guidance Computer	39 b	26 b	—	13 b	13 b, 26 b	—
1963 (1966)	Apollo Guidance Computer	15 b	*w*	—	*w*, 2*w*	*w*	—
1963	Saturn Launch Vehicle	26 b	*w*	—	13 b	*w*	—

	Digital Computer						
1964	CDC 6600	60 b	*w*	*w*	¼*w*, ½*w*	*w*	6 b
1964	Autonetics D-37C Minuteman II Guidance Computer	27 b	11 b, 24 b	—	24 b	*w*	4 b, 5 b
1965	IBM 360	32 b	½*w*, *w*, 1*d*, ... 16*d*	*w*, 2*w*	½*w*, *w*, 1½*w*	8 b	8 b
1965	UNIVAC 1108	36 b	⅙*w*, ¼*w*, ⅓*w*, ½*w*, *w*, 2*w*	*w*, 2*w*	*w*	*w*	6 b, 9 b
1965	PDP-8	12 b	*w*	—	*w*	*w*	8 b
1970	PDP-11	16 b	*w*	2*w*, 4*w*	*w*, 2*w*, 3*w*	8 b	8 b
1971	Intel 4004	4 b	*w*, *d*	—	2*w*, 4*w*	*w*	—
1972	Intel 8008	8 b	*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1972	Calcomp 900	9 b	*w*	—	*w*, 2*w*	*w*	8 b
1974	Intel 8080	8 b	*w*, 2*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1975	ILLIAC IV	64 b	*w*	*w*, ½*w*	*w*	*w*	—
1975	Motorola 6800	8 b	*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1975	MOS Tech. 6501 MOS Tech. 6502	8 b	*w*, 2*d*	—	*w*, 2*w*, 3*w*	*w*	8 b
1976	Cray-1	64 b	24 b, *w*	*w*	¼*w*, ½*w*	*w*	8 b
1976	Zilog Z80	8 b	*w*, 2*w*, 2*d*	—	*w*, 2*w*, 3*w*, 4*w*, 5*w*	*w*	8 b
1978 (1980)	Intel 8086 (w/Intel 8087)	16 b	½*w*, *w*, 2*d* (*w*, 2*w*, 4*w*)	— (2*w*, 4*w*, 5*w*, 17*d*)	½*w*, *w*, ... 7*w*	8 b	8 b
1978	VAX-11/780	32 b	¼*w*, ½*w*, *w*, 1*d*, ... 31*d*, 1*b*, ... 32*b*	*w*, 2*w*	¼*w*, ... 14¼*w*	8 b	8 b
1979	Motorola 68000	32 b	¼*w*, ½*w*, *w*, 2*d*	—	½*w*, *w*, ... 7½*w*	8 b	8 b
1982	Motorola	32 b	¼*w*, ½*w*,	—	½*w*, *w*, ... 7½*w*	8 b	8 b

(1983)	68020 (w/Motorola 68881)		*w*, 2*d*	(*w*, 2*w*, 2½*w*)			
1985	ARM1	32 b	*w*	—	*w*	8 b	8 b
1985	MIPS	32 b	¼*w*, ½*w*, *w*	*w*, 2*w*	*w*	8 b	8 b
1989	Intel 80486	16 b	½*w*, *w*, 2*d* *w*, 2*w*, 4*w*	2*w*, 4*w*, 5*w*, 17*d*	½*w*, *w*, ... 7*w*	8 b	8 b
1989	Motorola 68040	32 b	¼*w*, ½*w*, *w*, 2*d*	*w*, 2*w*, 2½*w*	½*w*, *w*, ... 7½*w*	8 b	8 b
1991	Alpha	64 b	¼*w*, ½*w*, *w*	*w*, 2*w*	½*w*	8 b	8 b
1991	Cray C90	64 b	32 b, *w*	*w*	¼*w*, ½*w*, 48b	*w*	8 b
1991	PowerPC	32–64 b	¼*w*, ½*w*, *w*	*w*, 2*w*	*w*	8 b	8 b
2000	IA-64	64 b	8 b, ¼*w*, ½*w*, *w*	½*w*, *w*	41 b	8 b	8 b
2002	XScale	32 b	*w*	*w*, 2*w*	½*w*, *w*	8 b	8 b

key: **b: bits, d: decimal digits, *w*: word size of architecture, *n*: variable size**

Chapter 17

Byte

The **byte**, is a unit of digital information in computing and telecommunications, that most commonly consists of eight bits. Historically, a byte was the number of bits used to encode a single character of text in a computer and it is for this reason the basic addressable element in many computer architectures.

The size of the byte has historically been hardware dependent and no definitive standards exist that mandate the size. The *de facto* standard of eight bits is a convenient power of two permitting the values 0 through 255 for one byte. Many types of applications use variables representable in eight or fewer bits, and processor designers optimize for this common usage. The popularity of major commercial computing architectures have aided in the ubiquitous acceptance of the 8-bit size. The term octet was defined to explicitly denote a sequence of 8 bits because of the ambiguity associated with the term byte.

History

The term *byte* was coined by Dr. Werner Buchholz in July 1956, during the early design phase for the IBM Stretch computer. It is a respelling of *bite* to avoid accidental mutation to *bit*.

Early computers were designed for 4-bit BCD code (binary coded decimal) or 6-bit code for printable "graphic set", which included 26 alphabetic characters (only uppercase), 10 Numerical digits, and from 11 to 25 special graphic symbols. To include the control characters and allow digital devices to communicate with each other and to process, store, and communicate character-oriented information such as written language, and lowercase characters, a 7-bit ASCII code was introduced. Since with just only one bit more an eight bits allows two four-bit patterns to efficiently encode two digits with binary coded decimal, the eight-bit EBCDIC character encoding was later adopted and promulgated as a standard by the IBM in the System/360, the preset byte.

The size of a byte was at first selected to be a multiple of existing teletypewriter codes, particularly the 6-bit codes used by the U.S. Army (Fieldata) and Navy.

In 1963, to end the use of incompatible teleprinter codes by different branches of the U.S. government, ASCII, a 7-bit code, was adopted as a Federal Information Processing Standard, making 6-bit bytes commercially obsolete. In the early 1960s, AT&T

introduced digital telephony first on long-distance trunk lines. These used the 8-bit μ-law encoding. This large investment promised to reduce transmission costs for 8-bit data. The use of 8-bit codes for digital telephony also caused 8-bit data "octets" to be adopted as the basic data unit of the early Internet.

In the late 1970s, microprocessors such as the Intel 8008 (the direct predecessor of the 8080, and then the 8086 used in early PCs) could perform a small number of operations on four bits, such as the DAA (decimal adjust) instruction, and the *half carry* flag, which were used to implement decimal arithmetic routines. These four-bit quantities were called nibbles, in homage to the then-common 8-bit bytes.

Reasons for the ubiquity of the eight-bit byte include the popularity of the IBM System/360 architecture, introduced in the 1960s, and the 8-bit microprocessors, introduced in the 1970s.

The term octet is used to unambiguously specify a size of eight bits, and is used extensively in protocol definitions, for example.

Unit symbol

Prefixes for bit and byte multiples

Decimal			Binary				
Value	SI		Value	IEC		JEDEC	
1000	k	kilo	1024	Ki	kibi	K	kilo
1000^2	M	mega	1024^2	Mi	mebi	M	mega
1000^3	G	giga	1024^3	Gi	gibi	G	giga
1000^4	T	tera	1024^4	Ti	tebi		
1000^5	P	peta	1024^5	Pi	pebi		
1000^6	E	exa	1024^6	Ei	exbi		
1000^7	Z	zetta	1024^7	Zi	zebi		
1000^8	Y	yotta	1024^8	Yi	yobi		

The unit symbol for the byte is specified in IEEE 1541 and the Metric Interchange Format as the upper-case character B, while other standards, such as the International Electrotechnical Commission (IEC) standard IEC 60027, appear silent on the subject.

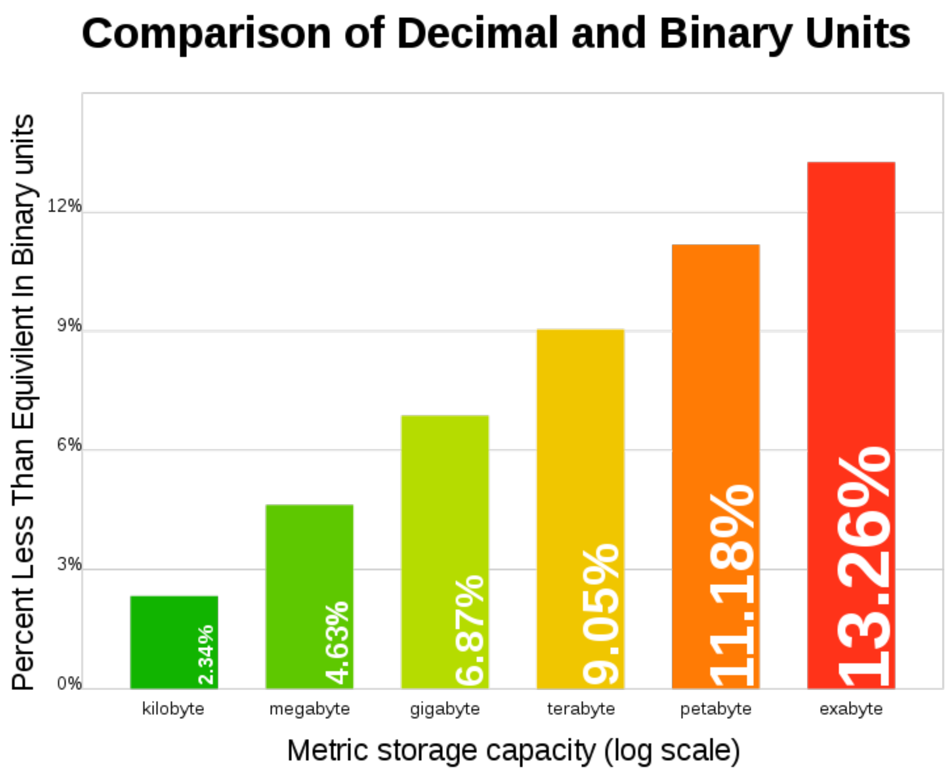
In the International System of Units (SI), B is the symbol of the bel, a unit of logarithmic power ratios named after Alexander Graham Bell. The usage of B for byte therefore conflicts with this definition. It is also not consistent with the SI convention that only units named after persons should be capitalized. However, there is little danger of confusion because the bel is a rarely used unit. It is used primarily in its decadic fraction, the decibel (dB), for signal strength and sound pressure level measurements, while a unit for one tenth of a byte, i.e. the decibyte, is never used.

The unit symbol kB is commonly used for kilobyte, but may be confused with the common meaning of kb for kilobit. IEEE 1541 specifies the lower case character b as the symbol for bit; however, the IEC 60027 and Metric-Interchange-Format specify bit (e.g., Mbit for megabit) for the symbol, a sufficient disambiguation from byte.

The lowercase letter o for octet is a commonly used symbol in several non-English languages (e.g., French and Romanian), and is also used with metric prefixes (for example, *ko* and *Mo*)

Today the harmonized ISO/IEC 80000-13:2008 – Quantities and units — Part 13: Information science and technology standard cancels and replaces subclauses 3.8 and 3.9 of IEC 60027-2:2005, namely those related to Information theory and Prefixes for binary multiples.

Unit multiples



Percentage difference between decimal and binary interpretations of the unit prefixes grows with increasing storage size.

There has been considerable confusion about the meanings of SI (or metric) prefixes used with the unit byte, especially concerning prefixes such as kilo (k or K) and mega (M) as shown in the chart *Prefixes for bit and byte*. Since computer memory is designed with binary logic, multiples are expressed in powers of 2, rather than 10. The software and computer industries often use binary estimates of the SI-prefixed quantities, while producers of computer storage devices prefer the SI values. This is the reason for specifying computer hard drive capacities of, say, 100 GB, when it contains 93 GiB of storage space.

While the numerical difference between the decimal and binary interpretations is small for the prefixes kilo and mega, it grows to over 20% for prefix yotta, illustrated in the linear-log graph (at right) of difference versus storage size.

Common uses

The byte is also defined as a data type in certain programming languages. The C and C++ programming languages, for example, define *byte* as an "*addressable unit of data large enough to hold any member of the basic character set of the execution environment*" (clause 3.6 of the C standard). The C standard requires that the `char` integral data type is capable of holding at least 255 different values, and is represented by at least 8 bits (clause 5.2.4.2.1). Various implementations of C and C++ reserve 8, 9, 16, 32, or 36 bits for the storage of a byte. The actual number of bits in a particular implementation is documented as `CHAR_BIT` as implemented in the `limits.h` file. Java's primitive `byte` data type is always defined as consisting of 8 bits and being a signed data type, holding values from −128 to 127.

In data transmission systems, a contiguous sequence of binary bits in a serial data stream, such as in modem or satellite communications, which is the smallest meaningful unit of data. These bytes might include start bits, stop bits, or parity bits, and thus could vary from 7 to 12 bits to contain a single 7-bit ASCII code.