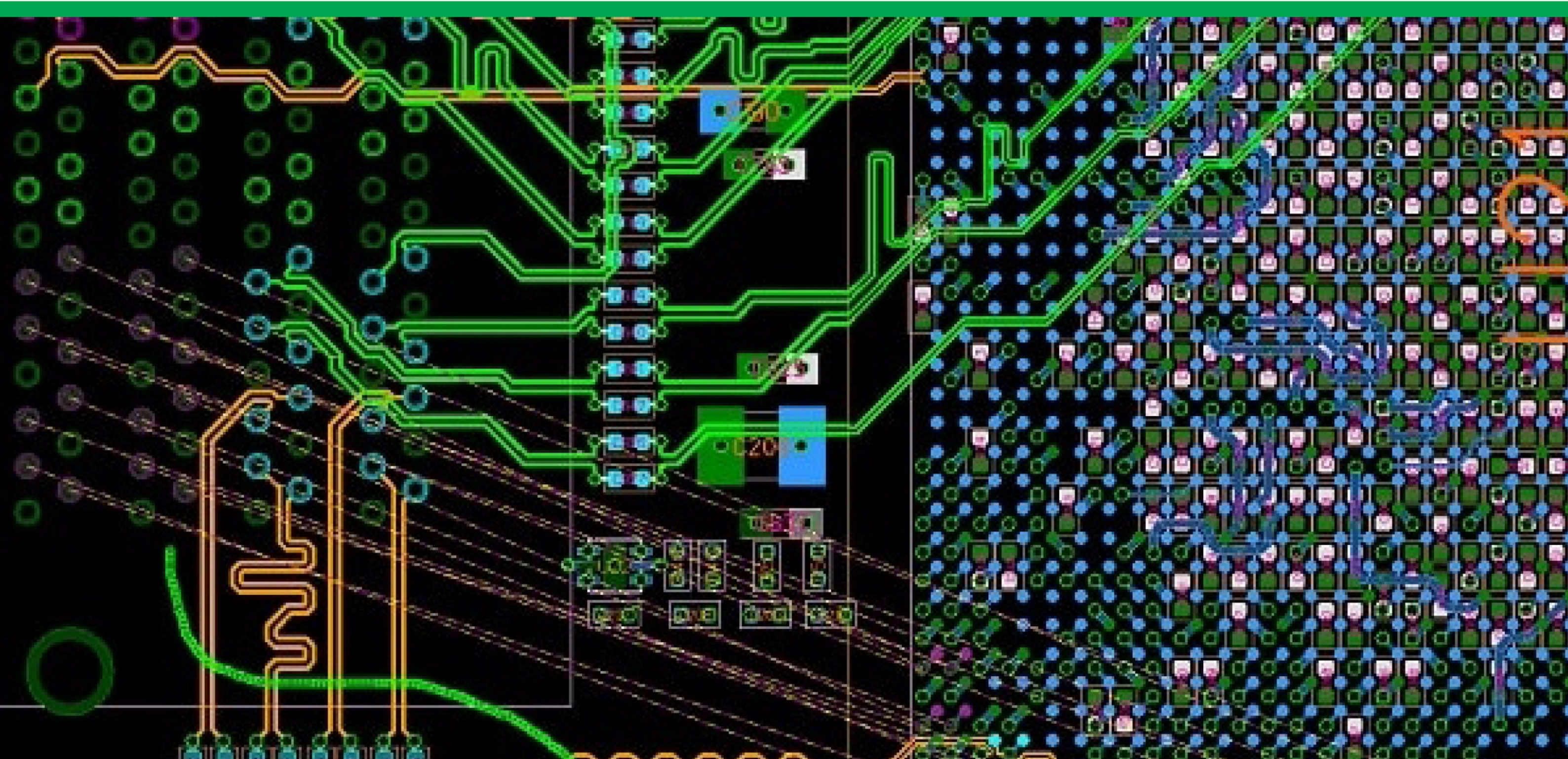
Electronic Design Automation

Alfred Talley



ELECTRONIC DESIGN AUTOMATION

ELECTRONIC DESIGN AUTOMATION

Alfred Talley

Electronic Design Automation
by Alfred Talley

Copyright© 2022 BIBLIOTEX

www.bibliotex.com

All rights reserved. No part of this book may be reproduced or used in any manner without the prior written permission of the copyright owner, except for the use brief quotations in a book review.

To request permissions, contact the publisher at info@bibliotex.com

Ebook ISBN: 9781984663955

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

Chapter 1	Introduction to Electronic Design Automation	1
Chapter 2	Compiler	28
Chapter 3	Scanning Electron Microscopy	56
Chapter 4	Data Analysis and Design	87
Chapter 5	Combinational Logic Design	112
Chapter 6	Conceptual Model	132

1

Introduction to Electronic Design Automation

Electronic design automation (EDA or ECAD) is a category of software tools for designing electronic systems such as printed circuit boards and integrated circuits. The tools work together in a design flow that chip designers use to design and analyze entire semiconductor chips.

HISTORY

Early Days

Before EDA, integrated circuits were designed by hand, and manually laid out. Some advanced shops used geometric software to generate the tapes for the Gerber photoplotter, but even those copied digital recordings of mechanically-drawn components. The process was fundamentally graphic, with the translation from electronics to graphics done manually. The best known company from this era was

Calma, whose GDSII format survives. By the mid-70s, developers started to automate the design, and not just the drafting. The first placement and routing (Place and route) tools were developed. The proceedings of the Design Automation Conference cover much of this era. The next era began about the time of the publication of “Introduction to VLSI Systems” by Carver Mead and Lynn Conway in 1980. This ground breaking text advocated chip design with programming languages that compiled to silicon.

The immediate result was a considerable increase in the complexity of the chips that could be designed, with improved access to design verification tools that used logic simulation. Often the chips were easier to lay out and more likely to function correctly, since their designs could be simulated more thoroughly prior to construction. Although the languages and tools have evolved, this general approach of specifying the desired behavior in a textual programming language and letting the tools derive the detailed physical design remains the basis of digital IC design today. The earliest EDA tools were produced academically. One of the most famous was the “Berkeley VLSI Tools Tarball”, a set of UNIX utilities used to design early VLSI systems. Still widely used is the Espresso heuristic logic minimizer and Magic. Another crucial development was the formation of MOSIS, a consortium of universities and fabricators that developed an inexpensive way to train student chip designers by producing real integrated circuits. The basic concept was to use reliable, low-cost, relatively low-technology IC processes, and pack a large number of projects per wafer, with just a few copies of each projects’ chips. Cooperating

fabricators either donated the processed wafers, or sold them at cost, seeing the programme as helpful to their own long-term growth.

Birth of Commercial EDA

1981 marks the beginning of EDA as an industry. For many years, the larger electronic companies, such as Hewlett Packard, Tektronix, and Intel, had pursued EDA internally. In 1981, managers and developers spun out of these companies to concentrate on EDA as a business. Daisy Systems, Mentor Graphics, and Valid Logic Systems were all founded around this time, and collectively referred to as DMV. Within a few years there were many companies specializing in EDA, each with a slightly different emphasis. The first trade show for EDA was held at the Design Automation Conference in 1984. In 1986, Verilog, a popular high-level design language, was first introduced as a hardware description language by Gateway Design Automation. In 1987, the U.S. Department of Defense funded creation of VHDL as a specification language. Simulators quickly followed these introductions, permitting direct simulation of chip designs: executable specifications. In a few more years, back-ends were developed to perform logic synthesis.

Current Status

Current digital flows are extremely modular (see Integrated circuit design, Design closure, and Design flow (EDA)). The front ends produce standardized design descriptions that compile into invocations of "cells,", without regard to the cell technology. Cells implement logic or other electronic

functions using a particular integrated circuit technology. Fabricators generally provide libraries of components for their production processes, with simulation models that fit standard simulation tools. Analog EDA tools are far less modular, since many more functions are required, they interact more strongly, and the components are (in general) less ideal. EDA for electronics has rapidly increased in importance with the continuous scaling of semiconductor technology. Some users are foundry operators, who operate the semiconductor fabrication facilities, or "fabs", and design-service companies who use EDA software to evaluate an incoming design for manufacturing readiness. EDA tools are also used for programming design functionality into FPGAs.

Software Focuses

Design

- High-level synthesis(syn. behavioural synthesis, algorithmic synthesis) For digital chips
- Logic synthesis translation of abstract, logical language such as Verilog or VHDL into a discrete netlist of logic-gates
- Schematic Capture For standard cell digital, analog, rf like Capture CIS in Orcad by CADENCE and ISIS in Proteus
- Layout like Layout in Orcad by Cadence, ARES in Proteus

Design Flows

Design flows are the explicit combination of electronic design automation tools to accomplish the design of an

integrated circuit. Moore's law has driven the entire IC implementation RTL to GDSII design flows from one which uses primarily standalone synthesis, placement, and routing algorithms to an integrated construction and analysis flows for design closure. The challenges of rising interconnect delay led to a new way of thinking about and integrating design closure tools. New scaling challenges such as leakage power, variability, and reliability will keep on challenging the current state of the art in design closure. The RTL to GDSII flow underwent significant changes from 1980 through 2005. The continued scaling of CMOS technologies significantly changed the objectives of the various design steps.

The lack of good predictors for delay has led to significant changes in recent design flows. Challenges like leakage power, variability, and reliability will continue to require significant changes to the design closure process in the future. Many factors describe what drove the design flow from a set of separate design steps to a fully integrated approach, and what further changes are coming to address the latest challenges. In his keynote at the 40th Design Automation Conference entitled The Tides of EDA, Alberto Sangiovanni-Vincentelli distinguished three periods of EDA: The Age of the Gods, The Age of the Heroes, and The Age of the Men. These eras were characterized respectively by senses, imagination, and reason. When we limit ourselves to the RTL to GDSII flow of the CAD area, we can distinguish three main eras in its development: the Age of Invention, the Age of Implementation, and the Age of Integration.

- The Age of Invention: During the invention era, routing, placement, static timing analysis and logic synthesis were invented.
- The Age of Implementation: In the age of implementation, these steps were drastically improved by designing sophisticated data structures and advanced algorithms. This allowed the tools in each of these design steps to keep pace with the rapidly increasing design sizes. However, due to the lack of good predictive cost functions, it became impossible to execute a design flow by a set of discrete steps, no matter how efficiently each of the steps was implemented.
- The Age of Integration: This led to the age of integration where most of the design steps are performed in an integrated environment, driven by a set of incremental cost analyzers.

Simulation

- Transistor simulation – low-level transistor-simulation of a schematic/layout's behavior, accurate at device-level.
- Logic simulation – digital-simulation of an RTL or gate-netlist's digital (boolean 0/1) behavior, accurate at boolean-level.
- Behavioral Simulation – high-level simulation of a design's architectural operation, accurate at cycle-level or interface-level.
- Hardware emulation – Use of special purpose hardware to emulate the logic of a proposed design.

Can sometimes be plugged into a system in place of a yet-to-be-built chip; this is called in-circuit emulation.

- Technology CAD simulate and analyze the underlying process technology. Electrical properties of devices are derived directly from device physics.
- Electromagnetic field solvers, or just field solvers, solve Maxwell's equations directly for cases of interest in IC and PCB design. They are known for being slower but more accurate than the layout extraction above.

Electronic Circuit Simulation

Electronic circuit simulation uses mathematical models to replicate the behavior of an actual electronic device or circuit. Simulation software allows for modeling of circuit operation and is an invaluable analysis tool. Due to its highly accurate modeling capability, many Colleges and Universities use this type of software for the teaching of electronics technician and electronics engineering programmes.

Electronics simulation software engages the user by integrating them into the learning experience. These kinds of interactions actively engage learners to analyze, synthesize, organize, and evaluate content and result in learners constructing their own knowledge. Simulating a circuit's behavior before actually building it can greatly improve design efficiency by making faulty designs known as such, and providing insight into the behavior of electronics circuit designs.

In particular, for integrated circuits, the tooling (photomasks) is expensive, breadboards are impractical, and probing the behavior of internal signals is extremely difficult. Therefore almost all IC design relies heavily on simulation. The most well known analog simulator is SPICE. Probably the best known digital simulators are those based on Verilog and VHDL. Some electronics simulators integrate a schematic editor, a simulation engine, and on-screen waveforms, and make "what-if" scenarios easy and instant. They also typically contain extensive model and device libraries. These models typically include IC specific transistor models such as BSIM, generic components such as resistors, capacitors, inductors and transformers, user defined models (such as controlled current and voltage sources, or models in Verilog-A or VHDL-AMS). Printed circuit board (PCB) design requires specific models as well, such as transmission lines for the traces and IBIS models for driving and receiving electronics.

Types

While there are strictly analog electronics circuit simulators, popular simulators often include both analog and event-driven digital simulation capabilities, and are known as mixed-mode simulators. This means that any simulation may contain components that are analog, event driven (digital or sampled-data), or a combination of both. An entire mixed signal analysis can be driven from one integrated schematic. All the digital models in mixed-mode simulators provide accurate specification of propagation time and rise/fall time delays.

The event driven algorithm provided by mixed-mode simulators is general purpose and supports non-digital types of data. For example, elements can use real or integer values to simulate DSP functions or sampled data filters. Because the event driven algorithm is faster than the standard SPICE matrix solution, simulation time is greatly reduced for circuits that use event driven models in place of analog models.

Mixed-mode simulation is handled on three levels; (a) with primitive digital elements that use timing models and the built-in 12 or 16 state digital logic simulator, (b) with subcircuit models that use the actual transistor topology of the integrated circuit, and finally, (c) with In-line Boolean logic expressions.

Exact representations are used mainly in the analysis of transmission line and signal integrity problems where a close inspection of an IC's I/O characteristics is needed. Boolean logic expressions are delay-less functions that are used to provide efficient logic signal processing in an analog environment. These two modeling techniques use SPICE to solve a problem while the third method, digital primitives, use mixed mode capability. Each of these methods has its merits and target applications. In fact, many simulations (particularly those which use A/D technology) call for the combination of all three approaches. No one approach alone is sufficient. Another type of simulation used mainly for power electronics represent piecewise linear algorithms. These algorithms use an analog (linear) simulation until a power electronic switch changes its state. At this time a new

analog model is calculated to be used for the next simulation period. This methodology both enhances simulation speed and stability significantly.

Complexities

Often circuit simulators do not take into account the process variations that occur when the design is fabricated into silicon. These variations can be small, but taken together can change the output of a chip significantly. Process variations occur in the manufacture of circuits in silicon. Temperature variation can also be modeled to simulate the circuit's performance through temperature ranges.

Analysis and Verification

- Functional verification
- Clock Domain Crossing Verification (CDC check): Similar to linting, but these checks/tools specialize in detecting and reporting potential issues like data loss, meta-stability due to use of multiple clock domains in the design.
- Formal verification, also model checking: Attempts to prove, by mathematical methods, that the system has certain desired properties, and that certain undesired effects (such as deadlock) cannot occur.
- Equivalence checking: algorithmic comparison between a chip's RTL-description and synthesized gate-netlist, to ensure functional equivalence at the *logical* level.
- Static timing analysis: Analysis of the timing of a circuit in an input-independent manner, hence finding a worst case over all possible inputs.

- Physical verification, PV: checking if a design is physically manufacturable, and that the resulting chips will not have any function-preventing physical defects, and will meet original specifications.

Manufacturing preparation

- Mask data preparation, MDP: generation of actual lithography photomask used to physically manufacture the chip.
    - o Resolution enhancement techniques, RET – methods of increasing of quality of final photomask.
    - o Optical proximity correction, OPC – up-front compensation for diffraction and interference effects occurring later when chip is manufactured using this mask.
    - o Mask generation – generation of flat mask image from hierarchical design.
    - o Automatic test pattern generation, ATPG – generates pattern-data to systematically exercise as many logic-gates, and other components, as possible.
    - o Built-in self-test, or BIST – installs self-contained test-controllers to automatically test a logic (or memory) structure in the design

Companies

For more details on this topic, see List of EDA companies.

Top Companies

- $3.73 billion - Synopsys
- $2.06 billion - Cadence
- $1.18 billion - Mentor Graphics

- $233 million - Magma Design Automation
- $157 million - Zuken Inc.

Note: Market caps current as of October, 2010. EEsof should likely be on this list, but does not have a market cap as it is the EDA division of Agilent.

Acquisitions

Many of the EDA companies acquire small companies with software or other technology that can be adapted to their core business. Most of the market leaders are rather incestuous amalgamations of many smaller companies. This trend is helped by the tendency of software companies to design tools as accessories that fit naturally into a larger vendor's suite of programmes (on digital circuitry, many new tools incorporate analog design, and mixed systems. This is happening because there is now a trend to place entire electronic systems on a single chip.

COMPUTER GRAPHICS

The development of computer graphics has made computers easier to interact with, and better for understanding and interpreting many types of data. Developments in computer graphics have had a profound impact on many types of media and have revolutionized animation, movies and the video game industry. The term computer graphics has been used in a broad sense to describe "almost everything on computers that is not text or sound". Typically, the term *computer graphics* refers to several different things:

- the representation and manipulation of image data by a computer

- the various technologies used to create and manipulate images
- the images so produced, and
- the sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content, see study of computer graphics

Today, computers and computer-generated images touch many aspects of daily life. Computer imagery is found on television, in newspapers, for example in weather reports, or for example in all kinds of medical investigation and surgical procedures. A well-constructed graph can present complex statistics in a form that is easier to understand and interpret. In the media "such graphs are used to illustrate papers, reports, thesis", and other presentation material. Many powerful tools have been developed to visualize data. Computer generated imagery can be categorized into several different types: 2D, 3D, 4D, 7D, and animated graphics. As technology has improved, 3D computer graphics have become more common, but 2D computer graphics are still widely used.

Computer graphics has emerged as a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Over the past decade, other specialized fields have been developed like information visualization, and scientific visualization more concerned with "the visualization of three dimensional phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth,

perhaps with a dynamic (time) component". The advance in computer graphics was to come from Ivan Sutherland. In 1961 Sutherland created another computer drawing programme called Sketchpad. Using a light pen, Sketchpad allowed one to draw simple shapes on the computer screen, save them and even recall them later. The light pen itself had a small photoelectric cell in its tip. This cell emitted an electronic pulse whenever it was placed in front of a computer screen and the screen's electron gun fired directly at it. By simply timing the electronic pulse with the current location of the electron gun, it was easy to pinpoint exactly where the pen was on the screen at any given moment. Once that was determined, the computer could then draw a cursor at that location. Sutherland seemed to find the perfect solution for many of the graphics problems he faced.

Even today, many standards of computer graphics interfaces got their start with this early Sketchpad programme. One example of this is in drawing constraints. If one wants to draw a square for example, s/he doesn't have to worry about drawing four lines perfectly to form the edges of the box. One can simply specify that s/he wants to draw a box, and then specify the location and size of the box. The software will then construct a perfect box, with the right dimensions and at the right location. Another example is that Sutherland's software modeled objects - not just a picture of objects. In other words, with a model of a car, one could change the size of the tires without affecting the rest of the car. It could stretch the body of the car without deforming the tires. These early computer graphics were Vector graphics, composed of thin lines whereas modern

day graphics are Raster based using pixels. The difference between vector graphics and raster graphics can be illustrated with a shipwrecked sailor.

He creates an SOS sign in the sand by arranging rocks in the shape of the letters "SOS." He also has some brightly colored rope, with which he makes a second "SOS" sign by arranging the rope in the shapes of the letters. The rock SOS sign is similar to raster graphics. Every pixel has to be individually accounted for. The rope SOS sign is equivalent to vector graphics. The computer simply sets the starting point and ending point for the line and perhaps bend it a little between the two end points. The disadvantages to vector files are that they cannot represent continuous tone images and they are limited in the number of colors available. Raster formats on the other hand work well for continuous tone images and can reproduce as many colors as needed. Also in 1961 another student at MIT, Steve Russell, created the first video game, Spacewar. Written for the DEC PDP-1, Spacewar was an instant success and copies started flowing to other PDP-1 owners and eventually even DEC got a copy. The engineers at DEC used it as a diagnostic programme on every new PDP-1 before shipping it. The sales force picked up on this quickly enough and when installing new units, would run the world's first video game for their new customers.

E. E. Zajac, a scientist at Bell Telephone Laboratory (BTL), created a film called "Simulation of a two-giro gravity attitude control system" in 1963. In this computer generated film, Zajac showed how the attitude of a satellite could be altered as it orbits the Earth. He created the animation on

an IBM 7090 mainframe computer. Also at BTL, Ken Knowlton, Frank Sindon and Michael Noll started working in the computer graphics field. Sindon created a film called Force, Mass and Motion illustrating Newton's laws of motion in operation.

Around the same time, other scientists were creating computer graphics to illustrate their research. At Lawrence Radiation Laboratory, Nelson Max created the films, "Flow of a Viscous Fluid" and "Propagation of Shock Waves in a Solid Form." Boeing Aircraft created a film called "Vibration of an Aircraft." It wasn't long before major corporations started taking an interest in computer graphics. TRW, Lockheed-Georgia, General Electric and Sperry Rand are among the many companies that were getting started in computer graphics by the mid 1960's. IBM was quick to respond to this interest by releasing the IBM 2250 graphics terminal, the first commercially available graphics computer. Ralph Baer, a supervising engineer at Sanders Associates, came up with a home video game in 1966 that was later licensed to Magnavox and called the Odyssey. While very simplistic, and requiring fairly inexpensive electronic parts, it allowed the player to move points of light around on a screen. It was the first consumer computer graphics product.

Also in 1966, Sutherland at MIT invented the first computer controlled head-mounted display (HMD). Called the Sword of Damocles because of the hardware required for support, it displayed two separate wireframe images, one for each eye. This allowed the viewer to see the computer scene in stereoscopic 3D. After receiving his Ph.D. from

MIT, Sutherland became Director of Information Processing at ARPA (Advanced Research Projects Agency), and later became a professor at Harvard. Dave Evans was director of engineering at Bendix Corporation's computer division from 1953 to 1962, after which he worked for the next five years as a visiting professor at Berkeley. There he continued his interest in computers and how they interfaced with people. In 1968 the University of Utah recruited Evans to form a computer science programme, and computer graphics quickly became his primary interest. This new department would become the world's primary research center for computer graphics. In 1967 Sutherland was recruited by Evans to join the computer science programme at the University of Utah. There he perfected his HMD. Twenty years later, NASA would re-discover his techniques in their virtual reality research.

At Utah, Sutherland and Evans were highly sought after consultants by large companies but they were frustrated at the lack of graphics hardware available at the time so they started formulating a plan to start their own company. A student by the name of Edwin Catmull started at the University of Utah in 1970 and signed up for Sutherland's computer graphics class. Catmull had just come from The Boeing Company and had been working on his degree in physics. Growing up on Disney, Catmull loved animation yet quickly discovered that he didn't have the talent for drawing. Now Catmull (along with many others) saw computers as the natural progression of animation and they wanted to be part of the revolution. The first animation that Catmull saw was his own. He created an animation

of his hand opening and closing. It became one of his goals to produce a feature length motion picture using computer graphics. In the same class, Fred Parke created an animation of his wife's face.

Because of Evan's and Sutherland's presence, UU was gaining quite a reputation as the place to be for computer graphics research so Catmull went there to learn 3D animation. As the UU computer graphics laboratory was attracting people from all over, John Warnock was one of those early pioneers; he would later found Adobe Systems and create a revolution in the publishing world with his PostScript page description language. Tom Stockham led the image processing group at UU which worked closely with the computer graphics lab. Jim Clark was also there; he would later found Silicon Graphics, Inc. The first major advance in 3D computer graphics was created at UU by these early pioneers, the hidden-surface algorithm. In order to draw a representation of a 3D object on the screen, the computer must determine which surfaces are "behind" the object from the viewer's perspective, and thus should be "hidden" when the computer creates (or renders) the image.

2D Computer Graphics

2D computer graphics are the computer-based generation of digital images—mostly from two-dimensional models, such as 2D geometric models, text, and digital images, and by techniques specific to them. 2D computer graphics are mainly used in applications that were originally developed upon traditional printing and drawing technologies, such as typography, cartography, technical drawing, advertising, etc..

In those applications, the two-dimensional image is not just a representation of a real-world object, but an independent artifact with added semantic value; two-dimensional models are therefore preferred, because they give more direct control of the image than 3D computer graphics, whose approach is more akin to photography than to typography.

Pixel Art

Pixel art is a form of digital art, created through the use of raster graphics software, where images are edited on the pixel level. Graphics in most old (or relatively limited) computer and video games, graphing calculator games, and many mobile phone games are mostly pixel art.

Vector Graphics

Vector graphics formats are complementary to raster graphics, which is the representation of images as an array of pixels, as it is typically used for the representation of photographic images Vector graphics consists in encoding information about shapes and colors that comprise the image, which can allow for more flexibility in rendering. There are instances when working with vector tools and formats is best practice, and instances when working with raster tools and formats is best practice. There are times when both formats come together. An understanding of the advantages and limitations of each technology and the relationship between them is most likely to result in efficient and effective use of tools.

3D Computer Graphics

3D computer graphics in contrast to 2D computer graphics are graphics that use a three-dimensional

representation of geometric data that is stored in the computer for the purposes of performing calculations and rendering 2D images. Such images may be for later display or for real-time viewing. Despite these differences, 3D computer graphics rely on many of the same algorithms as 2D computer vector graphics in the wire frame model and 2D computer raster graphics in the final rendered display. In computer graphics software, the distinction between 2D and 3D is occasionally blurred; 2D applications may use 3D techniques to achieve effects such as lighting, and primarily 3D may use 2D rendering techniques. 3D computer graphics are often referred to as 3D models. Apart from the rendered graphic, the model is contained within the graphical data file. However, there are differences. A 3D model is the mathematical representation of any three-dimensional object. A model is not technically a graphic until it is visually displayed. Due to 3D printing, 3D models are not confined to virtual space. A model can be displayed visually as a two-dimensional image through a process called *3D rendering*, or used in non-graphical computer simulations and calculations. There are some 3D computer graphics software for users to create 3D images.

Computer Animation

Computer animation is the art of creating moving images via the use of computers. It is a subfield of computer graphics and animation. Increasingly it is created by means of 3D computer graphics, though 2D computer graphics are still widely used for stylistic, low bandwidth, and faster real-time rendering needs. Sometimes the target of the animation

is the computer itself, but sometimes the target is another medium, such as film. It is also referred to as CGI (Computer-generated imagery or computer-generated imaging), especially when used in films. Virtual entities may contain and be controlled by assorted attributes, such as transform values (location, orientation, and scale) stored in an object's transformation matrix. Animation is the change of an attribute over time. Multiple methods of achieving animation exist; the rudimentary form is based on the creation and editing of keyframes, each storing a value at a given time, per attribute to be animated. The 2D/3D graphics software will interpolate between keyframes, creating an editable curve of a value mapped over time, resulting in animation.

Other methods of animation include procedural and expression-based techniques: the former consolidates related elements of animated entities into sets of attributes, useful for creating particle effects and crowd simulations; the latter allows an evaluated result returned from a user-defined logical expression, coupled with mathematics, to automate animation in a predictable way (convenient for controlling bone behavior beyond what a hierarchy offers in skeletal system set up). To create the illusion of movement, an image is displayed on the computer screen then quickly replaced by a new image that is similar to the previous image, but shifted slightly. This technique is identical to the illusion of movement in television and motion pictures.

Concepts and Principles

Images are typically produced by optical devices;such as cameras, mirrors, lenses, telescopes, microscopes, etc. and

natural objects and phenomena, such as the human eye or water surfaces. A digital image is a representation of a two-dimensional image in binary format as a sequence of ones and zeros. Digital images include both vector images and raster images, but raster images are more commonly used.

Pixel

In digital imaging, a pixel (or picture element) is a single point in a raster image. Pixels are normally arranged in a regular 2-dimensional grid, and are often represented using dots or squares. Each pixel is a sample of an original image, where more samples typically provide a more accurate representation of the original. The intensity of each pixel is variable; in color systems, each pixel has typically three components such as red, green, and blue.

Graphics

Graphics are visual presentations on some surface, such as a wall, canvas, computer screen, paper, or stone to brand, inform, illustrate, or entertain. Examples are photographs, drawings, line art, graphs, diagrams, typography, numbers, symbols, geometric designs, maps, engineering drawings, or other images. Graphics often combine text, illustration, and color. Graphic design may consist of the deliberate selection, creation, or arrangement of typography alone, as in a brochure, flier, poster, web site, or book without any other element. Clarity or effective communication may be the objective, association with other cultural elements may be sought, or merely, the creation of a distinctive style.

Rendering

Rendering is the process of generating an image from a model (or models in what collectively could be called a *scene* file), by means of computer programmes. A scene file contains objects in a strictly defined language or data structure; it would contain geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The data contained in the scene file is then passed to a rendering programme to be processed and output to a digital image or raster graphics image file. The rendering programme is usually built into the computer graphics software, though others are available as plug-ins or entirely separate programmes. The term "rendering" may be by analogy with an "artist's rendering" of a scene. Though the technical details of rendering methods vary, the general challenges to overcome in producing a 2D image from a 3D representation stored in a scene file are outlined as the graphics pipeline along a rendering device, such as a GPU. A GPU is a purpose-built device able to assist a CPU in performing complex rendering calculations. If a scene is to look relatively realistic and predictable under virtual lighting, the rendering software should solve the rendering equation. The rendering equation doesn't account for all lighting phenomena, but is a general lighting model for computer-generated imagery. 'Rendering' is also used to describe the process of calculating effects in a video editing file to produce final video output.

3D Projection

3D projection is a method of mapping three dimensional

points to a two dimensional plane. As most current methods for displaying graphical data are based on planar two dimensional media, the use of this type of projection is widespread, especially in computer graphics, engineering and drafting.

Ray Tracing

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane. The technique is capable of producing a very high degree of photorealism; usually higher than that of typical scanline rendering methods, but at a greater computational cost.

Shading

Shading refers to depicting depth in 3D models or illustrations by varying levels of darkness. It is a process used in drawing for depicting levels of darkness on paper by applying media more densely or with a darker shade for darker areas, and less densely or with a lighter shade for lighter areas. There are various techniques of shading including cross hatching where perpendicular lines of varying closeness are drawn in a grid pattern to shade an area. The closer the lines are together, the darker the area appears. Likewise, the farther apart the lines are, the lighter the area appears. The term has been recently generalized to mean that shaders are applied.

Texture Mapping

Texture mapping is a method for adding detail, surface texture, or colour to a computer-generated graphic or 3D model. Its application to 3D graphics was pioneered by Dr

Edwin Catmull in 1974. A texture map is applied (mapped) to the surface of a shape, or polygon. This process is akin to applying patterned paper to a plain white box. Multitexturing is the use of more than one texture at a time on a polygon. Procedural textures (created from adjusting parameters of an underlying algorithm that produces an output texture), and bitmap textures (created in an image editing application) are, generally speaking, common methods of implementing texture definition from a 3D animation programme, while intended placement of textures onto a model's surface often requires a technique known as UV mapping.

Anti-aliasing

Rendering resolution-independent entities (such as 3D models) for viewing on a raster (pixel-based) device such as a LCD display or CRT television inevitably causes aliasing artifacts mostly along geometric edges and the boundaries of texture details; these artifacts are informally called "jaggies". Anti-aliasing methods rectify such problems, resulting in imagery more pleasing to the viewer, but can be somewhat computationally expensive. Various anti-aliasing algorithms (such as supersampling) are able to be employed, then customized for the most efficient rendering performance versus quality of the resultant imagery; a graphics artist should consider this trade-off if anti-aliasing methods are to be used. A pre-anti-aliased bitmap texture being displayed on a screen (or screen location) at a resolution different than the resolution of the texture itself (such as a textured model in the distance from the virtual camera)

will exhibit aliasing artifacts, while any procedurally-defined texture will always show aliasing artifacts as they are resolution-independent; techniques such as mipmapping and texture filtering help to solve texture-related aliasing problems.

Volume Rendering

Volume rendering is a technique used to display a 2D projection of a 3D discretely sampled data set. A typical 3D data set is a group of 2D slice images acquired by a CT or MRI scanner. Usually these are acquired in a regular pattern (e.g., one slice every millimeter) and usually have a regular number of image pixels in a regular pattern. This is an example of a regular volumetric grid, with each volume element, or voxel represented by a single value that is obtained by sampling the immediate area surrounding the voxel.

3D Modeling

3D modeling is the process of developing a mathematical, wireframe representation of any three-dimensional object, called a “3D model”, via specialized software. Models may be created automatically or manually; the manual modeling process of preparing geometric data for 3D computer graphics is similar to plastic arts such as sculpting. 3D models may be created using multiple approaches: use of NURBS curves to generate accurate and smooth surface patches, polygonal mesh modeling (manipulation of faceted geometry), or polygonal mesh subdivision (advanced tessellation of polygons, resulting in smooth surfaces similar to NURBS models). A 3D model can be displayed as a two-dimensional

image through a process called *3D rendering*, used in a computer simulation of physical phenomena, or animated directly for other purposes. The model can also be physically created using 3D Printing devices.

2

Compiler

A compiler is a computer programme (or set of programmes) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable programme.

The name "compiler" is primarily used for programmes that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled programme can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A programme that translates from a low level language to a higher level one is a *decompiler*. A programme that translates between high-level languages is

usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a programme that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization. Programme faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest a lot of time ensuring the correctness of their software. The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

History

Software for early computers was primarily written in assembly language for many years. Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler. The very limited memory capacity of early computers also created many technical problems when implementing a compiler. Towards the end of the 1950s, machine-independent programming languages were first proposed. Subsequently, several experimental compilers were developed. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957.

COBOL was an early language to be compiled on multiple architectures, in 1960.

In many application domains the idea of using a higher level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers have become more and more complex.

Early compilers were written in assembly language. The first *self-hosting* compiler — capable of compiling its own source code in a high-level language — was created for Lisp by Tim Hart and Mike Levin at MIT in 1962. Since the 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been popular choices for implementation language. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must be compiled either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) compiled by running the compiler in an interpreter.

Compilers in Education

Compiler construction and compiler optimization are taught at universities and schools as part of the computer science curriculum. Such courses are usually supplemented with the implementation of a compiler for an educational programming language. A well-documented example is Niklaus Wirth's PL/0 compiler, which Wirth used to teach compiler construction in the 1970s. In spite of its simplicity, the PL/0 compiler introduced several influential concepts to the field:

1. Programme development by stepwise refinement (also the title of a 1971 paper by Wirth)
2. The use of a recursive descent parser
3. The use of EBNF to specify the syntax of a language
4. A code generator producing portable P-code
5. The use of T-diagrams in the formal description of the bootstrapping problem

Compilation

Compilers enabled the development of programmes that are machine-independent. Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programmes than machine code on the same architecture, it has to be modified or rewritten if the programme is to be executed on different hardware architecture. With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programmes. A compiler translates the high-level source programmes into target programmes in machine languages for the specific hardwares. Once the target programme is generated, the user can execute the programme.

Structure of a Compiler

Compilers bridge source programmes in high-level languages with the underlying hardware. A compiler requires 1) determining the correctness of the syntax of programmes, 2) generating correct and efficient object code, 3) run-time

organization, and 4) formatting output according to assembler and/or linker conventions. A compiler consists of three main parts: the frontend, the middle-end, and the backend.

The front end checks whether the programme is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programmes are recognized. Errors are reported, if any, in a useful way. Type checking is also performed by collecting type information. The frontend then generates an *intermediate representation* or *IR* of the source code for processing by the middle-end.

The middle end is where optimization takes place. Typical transformations for optimization are removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. The middle-end generates another IR for the following backend. Most optimization efforts are focused on this part.

The back end is responsible for translating the IR from the middle-end into assembly code. The target instruction(s) are chosen for each IR instruction. Variables are also selected for the registers. Backend utilizes the hardware by figuring out how to keep parallel FUs busy, filling delay slots, and so on. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.

Compiler Output

One classification of compilers is by the platform on which their generated code executes. This is known as the

target platform. A *native* or *hosted* compiler is one which output is intended to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment. The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

Compiled versus Interpreted Languages

Higher-level programming languages are generally divided for convenience into compiled languages and interpreted languages. However, in practice there is rarely anything about a language that *requires* it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters. Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. However,

there is nothing inherent in the definition of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programmes to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programmes must usually be shipped with a runtime library that includes a version of the compiler itself.

Hardware Compilation

The output of some compilers may target hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC). Such compilers are said to be *hardware compilers* or synthesis tools because the source code they compile effectively control the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

Compiler Construction

In the early days, the approach taken to compiler design used to be directly affected by the complexity of the processing, the experience of the person(s) designing it, and the resources available. A compiler for a relatively simple

language written by one person might be a single, monolithic piece of software. When the source language is large and complex, and high quality output is required, the design may be split into a number of relatively independent phases. Having separate phases means development can be parceled up into small parts and given to different people. It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations). The division of the compilation processes into phases was championed by the Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University. This project introduced the terms *front end*, *middle end*, and *back end*. All but the smallest of compilers have more than two phases.

However, these phases are usually regarded as being part of the front end or the back end. The point at which these two *ends* meet is open to debate. The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation (than source code). The middle end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors. The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS. This front-end/middle/back-end approach makes it possible to

combine front ends for different languages with back ends for different CPUs.

Practical examples of this approach are the GNU Compiler Collection, LLVM, and the Amsterdam Compiler Kit, which have multiple front-ends, shared analysis and multiple back-ends.

One-pass versus Multi-pass Compilers

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one programme that did all of this work. So compilers were split up into smaller programmes which each made a pass over the source (or some representation of it) performing some of the required analysis and translations. The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one pass compilers generally compile faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation

happening during a subsequent pass. The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programmes is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programmes often requires less effort than proving the correctness of a larger, single, equivalent programme. While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A “source-to-source compiler” is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language programme as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran’s DOALL statements).
- Stage compiler that compiles to assembly language of a theoretical machine, like some Prolog implementations
  - This Prolog machine is also known as the Warren Abstract Machine (or WAM).
  - Bytecode compilers for Java, Python, and many more are also a subtype of this.

- Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .NET's Common Intermediate Language (CIL)
  - o Applications are delivered in bytecode, which is compiled to native machine code just prior to execution.

Front End

The front end analyzes the source code to build an internal representation of the programme, called the intermediate representation or *IR*. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

1. Line reconstruction. Languages which strop their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a canonical form ready for the parser. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of ALGOL and Coral 66) are examples of stropped languages which compilers would have a *Line Reconstruction* phase.
2. Lexical analysis breaks the source code text into small pieces called *tokens*. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically

a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.

3. Preprocessing. Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.
4. Syntax analysis involves parsing the token sequence to identify the syntactic structure of the programme. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
5. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programmes

or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Back End

The term *back end* is sometimes confused with *code generator* because of the overlapped functionality of generating assembly code. Some literature uses *middle end* to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators. The main phases of the back end include the following:

1. Analysis: This is the gathering of programme information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.
2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization.

3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation. In addition, the scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/ function level, or even over the whole programme (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The open source GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect.

Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Compiler Correctness

Compiler correctness is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification. Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Related Techniques

Assembly language is a type of low-level language and a programme that compiles it is more commonly known as an *assembler*, with the inverse programme known as a *disassembler*.

A programme that translates from a low level language to a higher level one is a *decompiler*. A programme that translates between high-level languages is usually called a *language translator*, *source to source translator*, *language converter*, or *language rewriter*. The last term is usually applied to translations that do not involve a change of language.

International Conferences and Organizations

Every year, the European Joint Conferences on Theory and Practice of Software (ETAPS) sponsors the International Conference on Compiler Construction (CC), with papers from both the academic and industrial sectors.

DEBUGGER

A debugger or debugging tool is a computer programme that is used to test and debug other programmes (the "target" program). The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation - full or partial simulation, to limit this impact.

A "crash" happens when the programme cannot normally continue because of a programming bug. For example, the programme might have tried to use an instruction not available on the current version of the CPU or attempted to access unavailable or protected memory. When the programme "crashes" or reaches a preset condition, the debugger typically shows the position in the original code if it is a source-level debugger or symbolic debugger, commonly now seen in integrated development environments. If it is a low-level debugger or a machine-language debugger it shows the line in the disassembly (unless it also has online access to the original source code and can display the appropriate section of code from the assembly or compilation).

Typically, debuggers also offer more sophisticated functions such as running a programme step by step (single-stepping or programme animation), stopping (breaking) (pausing the programme to examine the current state) at

some event or specified instruction by means of a breakpoint, and tracking the values of some variables. Some debuggers have the ability to modify the state of the programme while it is running, rather than merely to observe it. It may also be possible to continue execution at a different location in the programme to bypass a crash or logical error.

The importance of a good debugger cannot be overstated. Indeed, the existence and quality of such a tool for a given language and platform can often be the deciding factor in its use, even if another language/platform is better-suited to the task.. The absence of a debugger, having once been accustomed to using one, has been said to "make you feel like a blind man in a dark room looking for a black cat that isn't there". However, software can (and often does) behave differently running under a debugger than normally, due to the inevitable changes the presence of a debugger will make to a software programme's internal timing. As a result, even with a good debugging tool, it is often very difficult to track down runtime problems in complex multi-threaded or distributed systems. The same functionality which makes a debugger useful for eliminating bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features. It often also makes it useful as a general testing verification tool test coverage and performance analyzer, especially if instruction path lengths are shown. Most current mainstream debugging engines, such as gdb and dbx provide console-based command line interfaces. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, programme animation, and visualization

features. Some early mainframe debuggers such as Oliver and SIMON provided this same functionality for the IBM System/360 and later operating systems, as long ago as the 1970s.

Language Dependency

Some debuggers operate on a single specific language while others can handle multiple languages transparently. For example if the main target programme is written in COBOL but CALLs Assembler subroutines and also PL/1 subroutines, the debugger may dynamically switch modes to accommodate the changes in language as they occur.

Memory Protection

Some debuggers also incorporate memory protection to avoid storage violations such as buffer overflow. This may be extremely important in transaction processing environments where memory is dynamically allocated from memory 'pools' on a task by task basis.

Hardware Support for Debugging

Most modern microprocessors have at least one of these features in their CPU design to make debugging easier:

- hardware support for single-stepping a programme, such as the trap flag.
- An instruction set that meets the Popek and Goldberg virtualization requirements makes it easier to write debugger software that runs on the same CPU as the software being debugged; such a CPU can execute the inner loops of the programme under test at full speed, and still remain under the control of the debugger.

- In-System Programming allows an external hardware debugger to re-programme a system under test (for example, adding or removing instruction breakpoints). Many systems with such ISP support also have other hardware debug support.
- Hardware support for code and data breakpoints, such as address comparators and data value comparators or, with considerably more work involved, page fault hardware.
- JTAG access to hardware debug interfaces such as those on ARM architecture processors or using the Nexus command set. Processors used in embedded systems typically have extensive JTAG debug support.
- Microcontrollers with as few as six pins need to use low pin-count substitutes for JTAG, such as BDM, Spy-Bi-Wire, or DebugWire on the Atmel AVR. DebugWire, for example, uses bidirectional signaling on the RESET pin.

List of Debuggers

- AppPuncher Debugger — for debugging Rich Internet Applications
- AQtime
- CA/EZTEST — was a CICS interactive test/debug software package
- CharmDebug — a Debugger for Charm++
- CodeView
- DBG — a PHP Debugger and Profiler
- dbx

- DDD (Data Display Debugger)
- Distributed Debugging Tool (Allinea DDT)
- DDTLite — Allinea DDTLite for Visual Studio 2008
- DEBUG — the built-in debugger of DOS and Microsoft Windows
- Debugger for MySQL
- Opera Dragonfly
- Dynamic debugging technique (DDT), and its octal counterpart Octal Debugging Technique
- Eclipse
- Embedded System Debug Plug-in for Eclipse
- FusionDebug
- gDEBugger OpenGL, OpenGL ES and OpenCL debugger and profiler
- GNU Debugger (GDB), GNU Binutils
- HyperDBG a kernel debugger that leverages hardware-assisted virtualization
- Intel Debugger (IDB)
- Insight
- Parasoft Insure++
- iSYSTEM — in-circuit debugger for embedded systems
- Interactive Disassembler (IDA Pro)
- Java Platform Debugger Architecture
- Jinx — a whole-system debugger for heisenbugs. It works transparently as a device driver.
- JSwat — open-source Java debugger
- LLDB
- MacsBug

- Nemiver — graphical C/C++ debugger for the GNOME desktop environment
- OLIVER (CICS interactive test/debug) - a GUI equipped *instruction set simulator* (ISS)
- OllyDbg
- Omniscient Debugger — Forward and backward debugger for Java
- pydbg
- IBM Rational Purify
- RealView Debugger — Commercial debugger produced for and designed by ARM
- sdb
- SIMMON (Simulation Monitor)
- SIMON (Batch Interactive test/debug) — a GUI equipped *instruction set simulator* (ISS) for batch
- SoftICE
- TimeMachine — Forward and backward debugger designed by Green Hills Software
- TotalView
- Lauterbach TRACE32 — in-circuit debugger for embedded Systems
- Turbo Debugger
- Ups — C, Fortran source level debugger
- Valgrind
- VB Watch Debugger — debugger for Visual Basic 6.0
- Microsoft Visual Studio Debugger
- WinDbg
- Xdebug — PHP debugger and profiler

Debugger Front-ends

Some of the most capable and popular debuggers only implement a simple command line interface (CLI) — often to maximize portability and minimize resource consumption. Debugging via a graphical user interface (GUI) can be considered easier and more productive though. This is the reason for GUI debugger front-ends, that allow users to monitor and control subservient CLI-only debuggers via graphical user interface. Some GUI debugger front-ends are designed to be compatible with a variety of CLI-only debuggers, while others are targeted at one specific debugger.

List of Debugger Front-ends

- Many Integrated development environments come with integrated debuggers (or front-ends to standard debuggers).
  - Many Eclipse perspectives, e.g. the Java Development Tools (JDT), provide a debugger front-end.
- DDD is the standard front-end from the GNU Project. It is a complex tool that works with most common debuggers (GDB, jdb, Python debugger, Perl debugger, Tcl, and others) natively or with some external programmes (for PHP).
- GDB (the GNU debugger) GUI
  - Emacs — Emacs editor with built in support for the GNU Debugger acts as the frontend.
  - KDbg — Part of the KDE development tools.
  - Nemiver — A GDB frontend that integrates well in the GNOME desktop environment.

- o xxgdb — X-window frontend for GDB and dbx debugger.
- o Qt Creator — multi-platform frontend for GDB (debugging example).
- o cgdb — ncurses terminal programme that mimics vim key mapping.
- o ccdebug— A graphical GDB frontend using the Qt toolkit.
- o Padb — has a parallel front-end to GDB allowing it to target parallel applications.
- o Allinea's DDT — a parallel and distributed front-end to a modified version of GDB.
- o Xcode — contains a GDB front-end as well.
- o SlickEdit — contains a GDB front-end as well.
- o Eclipse C/C++ Development Tools (CDT) — includes visual debugging tools based on GDB.

PROFILING

In software engineering, programme profiling, software profiling or simply profiling, a form of dynamic programme analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the programme executes. The usual purpose of this analysis is to determine which sections of a programme to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

- A (code) profiler is a performance analysis tool that, most commonly, measures only the frequency and duration of function calls, but there are other specific

types of profilers (e.g. memory profilers) in addition to more comprehensive profilers, capable of gathering extensive performance data.

- An instruction set simulator which is also — by necessity — a profiler, can measure the totality of a program's behaviour from invocation to termination.

Gathering Programme Events

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. The usage of profilers is 'called out' in the performance engineering process.

History

Performance analysis tools existed on IBM/360 and IBM/370 platforms from the early 1970s, usually based on timer interrupts which recorded the Programme status word (PSW) at set timer intervals to detect "hot spots" in executing code. This was an early example of sampling. In early 1974, Instruction Set Simulators permitted full trace and other performance monitoring features. Profiler-driven programme analysis on Unix dates back to at least 1979, when Unix systems included a basic tool "prof" that listed each function and how much of programme execution time it used. In 1982, gprof extended the concept to a complete call graph analysis. In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing ATOM. ATOM is a platform for converting a programme into its own profiler. That is, at compile time, it inserts code into

the programme to be analyzed. That inserted code outputs analysis data. This technique - modifying a programme to analyze itself - is known as "instrumentation". In 2004, both the gprof and ATOM papers appeared on the list of the 50 most influential PLDI papers of all time.

Profiler Types based on Output

Flat Profiler

Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.

Call-graph Profiler

Call graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. However context is not preserved.

Methods of Data Gathering

Event-based Profilers

The programming languages listed here have event-based profilers:

- Java: the JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface), provides hooks to profilers, for trapping events like calls, class-load, unload, thread enter leave.
- .NET: Can attach a profiling agent as a COM server to the CLR. Like Java, the runtime then provides various callbacks into the agent, for trapping events like method JIT / enter / leave, object creation, etc. Particularly powerful in that the profiling agent can

rewrite the target application's bytecode in arbitrary ways.

- Python: Python profiling includes the profile module, hotshot (which is call-graph based), and using the 'sys.setprofile' function to trap events like c_{call,return,exception},python_{call,return,exception}.
- Ruby: Ruby also uses a similar interface like Python for profiling. Flat-profiler in profile.rb, module, and ruby-prof a C-extension are present.

Statistical Profilers

Some profilers operate by sampling. A sampling profiler probes the target program's programme counter at regular intervals using operating system interrupts. Sampling profiles are typically less numerically accurate and specific, but allow the target programme to run at near full speed. The resulting data are not exact, but a statistical approximation. *The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods.* In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target programme, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines). Also since they don't affect the execution speed as much, they can detect issues that would otherwise be hidden. They are also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing.

Still, kernel code to handle the interrupts entails a minor loss of CPU cycles, diverted cache usage, and is unable to distinguish the various tasks occurring in uninterruptible kernel code (microsecond-range activity). Dedicated hardware can go beyond this: some recent MIPS processors JTAG interface have a PCSAMPLE register, which samples the programme counter in a truly undetectable manner. Some of the most commonly used statistical profilers are AMD CodeAnalyst, Apple Inc. Shark, gprof, Intel VTune and Parallel Amplifier (part of Intel Parallel Studio).

Instrumentation

- Manual: Performed by the programmer, e.g. by adding instructions to explicitly calculate runtimes, simply count events or calls to measurement APIs such as the Application Response Measurement standard.
- Automatic source level: instrumentation added to the source code by an automatic tool according to an instrumentation policy.
- Compiler assisted: Example: "gcc -pg ..." for gprof, "quantify g++ ..." for Quantify
- Binary translation: The tool adds instrumentation to a compiled binary. Example: ATOM
- Runtime instrumentation: Directly before execution the code is instrumented. The programme run is fully supervised and controlled by the tool. Examples: Pin, Valgrind
- Runtime injection: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions. Example: DynInst

Interpreter Instrumentation

- Interpreter debug options can enable the collection of performance metrics as the interpreter encounters each target statement. A bytecode, control table or JIT interpreters are three examples that usually have complete control over execution of the target code, thus enabling extremely comprehensive data collection opportunities.

Hypervisor/Simulator

- Hypervisor: Data are collected by running the (usually) unmodified programme under a hypervisor. Example: SIMMON
- Simulator and Hypervisor: Data collected interactively and selectively by running the unmodified programme under an Instruction Set Simulator. Examples: SIMON and OLIVER.

3

Scanning Electron Microscopy

SEM Provides Topographical and Elemental Information at Magnifications of 10x to 100,000x with Virtually Unlimited Depth of Field.

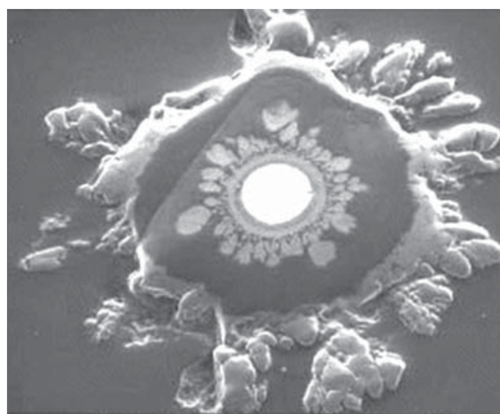
Applications Include

- **Materials evaluation**
  - Grain size
  - Surface roughness
  - Porosity
  - Particle size distributions
  - Material homogeneity
  - Intermetallic distribution and diffusion
- **Failure analysis**
  - Contamination location
  - Mechanical damage assessment
  - Electrostatic discharge effects
  - Micro-crack location

- **Quality Control screening**
  – "Good" to "bad" sample comparison
  – Film and coating thickness determination
  – Dimension verification
  – Gate width measurement
  – Mil Std. screening

Principle of Operation

A finely focused electron beam scanned across the surface of the sample generates secondary electrons, backscattered electrons, and characteristic X-rays. These signals are collected by detectors to form images of the sample displayed on a cathode ray tube screen. Features seen in the SEM image may then be immediately analyzed for elemental composition using *EDS* or *WDS*.



Secondary Electron Imaging: shows the topography of surface features a few nm across. Films and stains as thin as 20 nm produce adequate-contrast images. Materials are viewed at useful magnifications up to 100,000x without the need for extensive sample preparation and without damaging the sample. Even higher magnifications and resolution are routinely obtained by our *Field Emission SEM.*

Backscattered Electron Imaging: shows the spatial distribution of elements or compounds within the top micron of the sample. Features as small as 10 nm are resolved and composition variations of as little as as 0.2% determined.

Data Output: is generated in real time on the CRT monitor. Images and spectra can be printed here, recorded on CD ROM and/or emailed for insertion into your own reports.

Field Emission

Scanning Electron Microscopy (FESEM)

Principle of Operation

A field-emission cathode in the electron gun of a scanning electronmicroscope provides narrower probing beams at low as well as high electronenergy, resulting in both improved spatial resolution and minimized samplecharging and damage. For applications which demand the highest magnification possible.

Applications Include

- Semiconductor device cross section analyses for gate widths, gate oxides,film thicknesses, and construction details
- Advanced coating thickness and structure uniformity determination

- Small contamination feature geometry and elemental composition measurement

Why Field Emission SEM?

- FESEM produces clearer, less electrostatically distorted images withspatial resolution down to 1 1/2 nm. That's 3 to 6 times better than conventional SEM.
- Smaller-area contamination spots can be examined at electron acceleratingvoltages compatible with Energy Dispersive X-ray Spectroscopy.
- Reduced penetration of low kinetic energy electrons probes closer tothe immediate material surface.
- High quality, low voltage images are obtained with negligible electrical charging of samples. (Accelerating voltages range from 0.5 to 30 kV.)
- Need for placing conducting coatings on insulating materials is virtually eliminated.
- For ultra-high magnification imaging.

Cross-section of a Laser Window Showing Multiple thin Layers at 50,000x

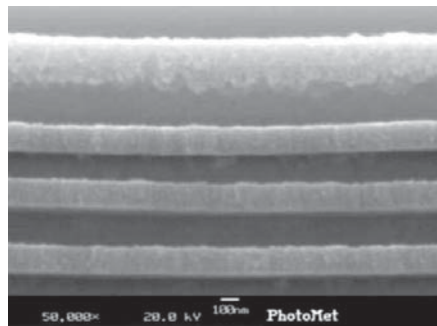


Fig. Cross Section of Contact on Silicon

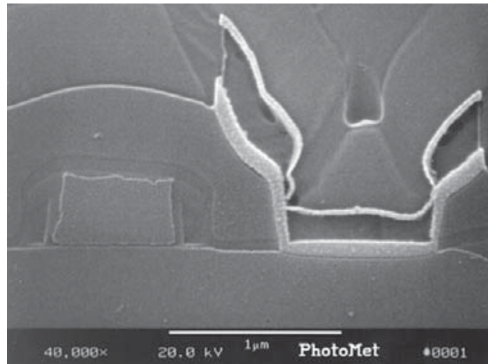
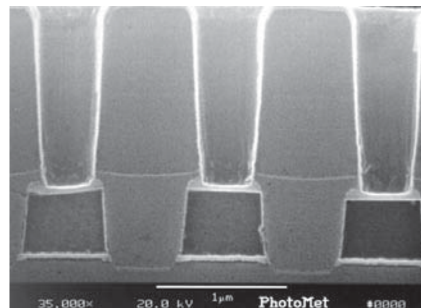


Fig. Cross Section of Via Openings



Energy Dispersive X-Ray Spectroscopy (EDS)

EDS identifies the elemental composition of materials imaged in a Scanning Electron Microscope *(SEM)* for all elements with an atomic number greater than boron. Most elements are detected at concentrations on the order of 0.1%.

Applications Include

- Materials evaluation and identification
  - Contaminants.
  - Elemental diffusion profiles.
  - Glassivation phosphorus content.
  - Multiple spot analysis of areas from 1 micron to 10 cm in diameter.
- Failure analysis
  - Contamination identification.
  - Unknowns identification.

  - Stringer location and identification.
- Quality control screening
  - Material verification.
  - Plating specification and certification.

Principle of Operation

As the electron beam of the SEM is scanned across the sample surface, it generates X-ray fluorescence from the atoms in its path. The energy of each X-ray photon is characteristic of the element which produced it. The EDS microanalysis system collects the X-rays, sorts and plots them by energy, and automatically identifies and labels the elements responsible for the peaks in this energy distribution.

The EDS data are typically compared with either known or computer-generated standards to produce a full quantitative analysis showing the sample composition.

Data output: Plots the original spectrum showing the number of X-rays collected at each energy, as seen above. Maps of element distributions over areas of interest and quantitative composition tables can also be provided as necessary.

Atomic Force Microscope/Scanning Probe Microscopy

Atomic Force Microscopy and Scanning Probe Microscopy (AFM/SPM) provide topographic information down to the Angstrom level. Additional properties of the sample, such as thermal and electrical conductivity, magnetic and electric field strength, and sample compliance can simultaneously be obtained using a specialty probe. Many applications require little or no sample preparation.

Principle of Operation

The Atomic Force Microcope uses a physical probe raster scanning across the sample using piezoelectric ceramics. A feedback loop is used to maintain a constant interaction between the probe and the sample. The position of the probe and the feedback signal are electronically recorded to produce a three dimensional map of the surface or other information depending on the specialty probe used.

Data Output: is either a three dimensional image of the surface or a line profile with height measurements. The surface roughness parameters of Ra or RMS are also available with either of the above outputs.

Other types of feature analysis include Partical Grain Size Analysis, Bearing Ratio, Fractal Dimension, Power Spectrum, and Fast Fourier Transform.

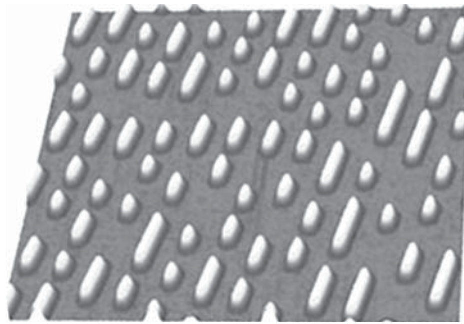
Applications Include

- Materials Evaluation
  - Surface roughness on implanted silicon wafers.
  - Thermal properties such as thermal conductivity, glass transition temperature (Tg), and melting temperature of various phases of a blended polymer measured down to the nanometer scale.
  - Surface profiles and magnetic field mapping of recording media or reading heads.
  - Nanomechanical testing.
- Failure Analysis
  - Rapid hot-spot analysis of powered electronic devices.
  - Defect analysis of compact disk stampers.
- Quality Control
  - Surface profiles of thin film and coatings.

- Metrology of semiconductor devices and compact disks.
- Surface finish of substrates for thin film deposition.

SPM Techniques

- *Magnetic Force Microscopy*: Digital Video Disc Surface (10 microns)



Wavelength Dispersive X-Ray Spectrometry (WDS)

WDS identifies the elemental composition of materials imaged in the *SEM* with an order of magnitude better spectral resolution, sensitivity and ability to determine concentratins of light elements than is achievable with *EDS*. Most elements are detected below 0.1% and some as low as a few ppm.

Applications Include

- Identification of *spectrally overlapped* elements, such as
  - S in the presence of Pb or Mo.
  - W or Ta in Si, or N in Ti.
  - Br in Al, common in semiconductor device failure.
- Detection of *low concentration* species (down to 100 or even 10 ppm)

  - P or S in metals.
  - Contaminants in precious metal catalysts.
  - Trace heavy metal contamination.
  - Performance-degrading impurities in high temperature solder alloys.
- Analysis of *low atomic number* elements
  - Composition of advanced ceramics and composites.
  - B in BPSG films (sensitivity to 2000 ppm).
  - Oxidation and corrosion of metals.
  - Characterization of biomedical and organically modified materials.

Principle of Operation

The characteristic X-ray photons excited by the electron beam are sorted using a diffracting crystal, whose angular placement relative to the sample and photodetector is a unique measure of their wavelengths. As with EDS, the resulting spectral distribution is automatically compared with those from actual standards or synthetic X-ray fluorescence spectra of material formulations.

WDS vs. EDS

X-ray microanalysis in the scanning electron microscope is accomplished using EDS and/or WDS. EDS is more commonly applied due to its simplicity and speed, while WDS offers an important and often critical refinement of EDS data by providing

- Analysis for light elements with at least an order of magnitude higher sensitivity than available (ultrathin X-ray window) EDS instruments.

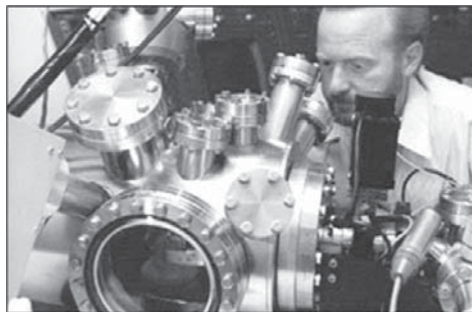
- Resolution of severely overlapped spectrum peaks for improved element specificity.
- Lowered detection limits over the entire periodic table.
- More accurate quantitative analyses.

Scanning Auger Microanalysis (SAM)

SAM provides elemental and chemical composition for all elements with an atomic number greater than helium. Its sampling depth of 2-3 nm allows films as thin as a few monolayers to be analyzed. Auger also produces images of the distributions of elements along the surface and produces profiles of composition vs. depth from 1 to 2000 nm.

Applications Include

- Materials evaluation and identification
  - Surface contaminants
  - Surface homogeneity
  - Diffusion profiles
  - Particle sizes
  - Catalyst degradation
  - Interfaces

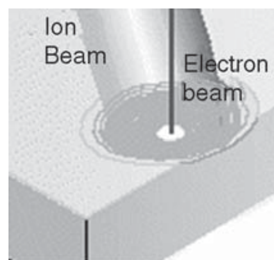


- Failure Analysis
  - Corrosion characterization
  - Stain identification

  - Lifted lead bond evaluation
  - Material delamination analysis
  - Metal embrittlement evaluation
- Quality control screening
  - "Good" to "bad" sample comparison.
  - Material and plating/coating thickness determination.
  - Surface process modification.

Principle of Operation

The sample is scanned with a focused beam of about 5 kV electrons, causing low energy Auger electrons to be ejected from its surface. The kinetic energies of these Auger electrons provide an analysis for the chemical elements present in the top few atomic layers. An auxiliary argon ion beam may be used to remove near-surface layers by "sputtering" to expose a fresh surface for analysis, producing a profile showing the dependence of sample composition on depth.



Fourier Transform Infrared Spectroscopy (FTIR)

FTIR spectroscopy is used primarily for qualitative and quantitative analysis of organic compounds, and also for determining the chemical structure of many inorganics.

FTIR analysis applications include:

- Materials Evaluation and Identification

  - Organic compounds
  - Structure of many inorganic compounds
  - Deformulations
  - Forensics
  - Material homogeneity

- Failure analysis
  - Micro-contamination identification
  - Adhesive performance
  - Material delamination
  - Corrosion chemistry.
- Quality control screening
  - "Good" to "bad" sample comparison
  - Evaluation of cleaning procedure effectiveness
  - Comparison of materials from different lots or vendors.

Principle of Operation

Because chemical bonds *absorb infrared energy at specific frequencies* (or wavelengths), the basic structure of compounds can be determined by the spectral locations of their IR absorptions. The plot of a compound's IR transmission vs. frequency is its "fingerprint", which when compared to reference spectra identifies the material. FTIR spectrometers offer speed and sensitivity impossible to

achieve with earlier wavelength-dispersive instruments. This capability allows rapid analysis of micro-samples down to the nanogram level in some cases, making the FTIR unmatched as a problem-solving tool in organic analysis.

The FTIR microscope accessory (shown in the photo above) allows spectra from a few nanograms of material to be obtained quickly, with little sample preparation, resulting in more data at lower cost. In some cases, thin films of residue are identified with a sensitivity that rivals or even exceeds electron or ion beam-based surface analysis techniques.

There are few sample constraints; solids, liquids and gases can be accommodated. Many contaminants present on reflective surfaces such as solder pads or printed circuitry are readily analyzed *in situ* using the FTIR microscope in reflectance mode.

More information on infrared spectroscopy:

Basic theory of infrared spectroscopy

Identifying organic structure by FTIR.

Differential Scanning Calorimetry (DSC)

Differential Scanning Calorimetry, DSC, is a thermo analytical technique in which the difference in the amount of *heat* required to increase the temperature of a sample and reference are measured as a function of temperature.

Both the sample and reference are maintained at nearly the same temperature throughout the experiment. Generally, the temperature programme for a DSC analysis is designed such that the sample holder temperature increases linearly as a function of time. Only a few milligrams of material are required to run the analysis.

Principle of Operation

When a sample undergoes a physical transformation such as a *phase transition*, more or less heat will need to flow to it than to the reference (typically an empty sample pan) to maintain both at the same temperature. Whether more or less heat must flow to the sample depends on whether the process is *exothermic* or *endothermic*.

For example, as a solid sample melts to a liquid it will require more heat flowing to the sample to increase its temperature at the same rate as the reference. This is due to the absorption of heat by the sample as it undergoes the endothermic phase transition from solid to liquid.

Likewise, as the sample undergoes exothermic processes (such as *crystallization*) less heat is required to raise the sample temperature.

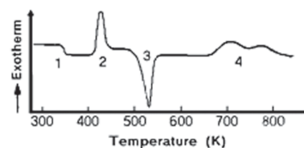
By observing the difference in heat flow between the sample and reference, differential scanning calorimeters are able to measure the amount of heat absorbed or released during such transitions. DSC may also be used to observe more subtle phase changes, such as *glass transitions*.

Applications

DSC is commonly used to measure a variety of properties in both organic and inorganic materials, from metals and simple compounds to polymers and pharmaceuticals. The properties measured include:

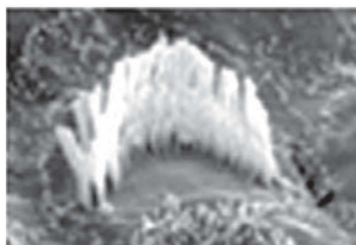
- Glass transitions
- Phase changes
- Melting
- Crystallization

- Product stability
- Cure/cure kinetics
- Oxidative stability
- Heat capacity and heat of fusion measurements



Outline of Analytical Imaging Facility Capabilities

The Analytical Imaging Facility provides a comprehensive light and electron microscope imaging facility dedicated to bringing state of the art methods in modern imaging to biomedical scientists with all levels of expertise. The AIF staff has been cross-trained to offer a seamless transition from classical histology, to high resolution light microscope imaging in 3D, to state of the art electron microscopy.



This unified approach facilitates the efficient and appropriate complementary use of these methods in research. For the infrequent user, the AIF provides a completely assisted technical support service. For the trained microscopist, the AIF is an available equipment resource. A significant effort

is devoted to training investigators who require microscopy techniques to advance their projects.

The Services

Transmission Electron Microscopy

JEOL 1200EX. This instrument offers the highest basic performance as a 120 kv transmission electron microscope employing a uniquely designed 3-stage 6-lens imaging system. It offers operational ease, excellent image quality and high resolution at low to high magnifications. It is equipped with side entry goniometer stage, minimum dose system, bottom mounted high resolution Gatan video camera and side mounted wide angle Gatan video camera.

JEOL 100CXII. This high performance 100 kv transmission electron microscope features a cool beam electron gun, high image contrast, high-speed cascade differential evacuation, optimum underfocus system using an image wobbler and a side entry goniometer.

Cryo Transmission Electron Microscopy for Single Molecule Imaging

The College, HHMI and NIH (by way of an awarded Shared Instrumentation Grant) have supported establishing a full Cryo EM programme. The technology spans the resolution range from electron microscopy to X-ray crystalography and allows for imaging of single molecules in their hydrated state.

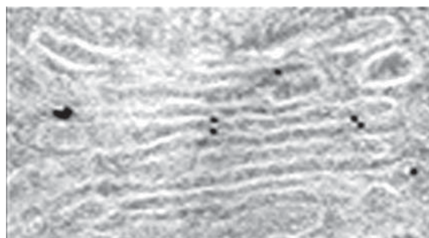
Scanning Electron Microscopy

JEOL 6400. This high performance Scanning Electron Microscope operates with accelerating voltage from 0.2 kv to 35 kv utilizing a high brightness LaB6 filament. It offers full

keyboard operation, framestore with digital image processing and digital image capture on a PC running analy SIS software.

Specimen Preparation for Electron Microscopy

The staff of the AIF offer full service sample preparation for many standard and state of the art EM techniques. These include:



- *Embedding:* utilizing either epoxy or acrylic resins at ambient or low temperatures.
- *Thin Sectioning:* with a Reichert Ultracut E or Leica UCT ultramicrotome.
- Negative Staining.
- *Freeze Fracture* using a Cressington CFE-50 Freeze Etch Unit.
- *Immunogold Labeling* following pre or post embedding protocols.
- *Critical Point Drying* with a Tousimis Samdri 790 Critical Point Dryer and Sputter Coating using a Denton Sputter Coater for preparing cells and tissues for SEM Imaging.
- Cryoultramicrotomy utilizing a Leica UCT cryoultramicrotome for optimizing epitope availability and *morphological preservation for immunogold labeling.*
- Slam Freeze Cryofixation using a Life Cell CF100 Slam

Freezer, which can be followed by High Resolution Rotary Shadowing in a Cressington CFE-50 equipped with e-beam guns for platinum or tungsten-tantalum evaporation. Slam Freezing followed by rotary shadowing is a powerful technique to obtain high resolution images of cells or macromolecules in 3-D, that have been frozen in the hydrated state.

- Slam Freeze Cryofixation using a Life Cell CF100 Slam Freezer followed by *Freeze Substitution* and *Low Temperature Embedding* in a Bal Tec FSU-010 Freeze Substitution Unit. Freeze Substitution is an alternative method for optimizing epitope preservation for immunogold labeling.

Photographic Documentation for Electron Microscopy

The AIF offers a photographic service for producing high quality electron micrographs on conventional photographic paper or high resolution scanning and direct digital printing for poster, lecture, web and journal publication.

Light Microscopy

Routine light microscopy

A Zeiss AxioSkop II with optics for brightfield, darkfield (through the condenser or via true oblique illumination), phase contrast, Nomarski, polarized light and epi-fluorescence with1.25X through 100X objectives serves as the “routine” microscope. Images are recorded with a colour Zeiss AxioCam.

To meet the growing demand for imaging of live material, epecially of eGFP or other fluorescently tagged cells, or of

cells in culture dishes, the AIF has Olympus inverted microscopes with a wide array of optics and photography options. On other inverted systems in the AIF researchers continue to use video digitizing technology for imaging motile cells in phase contrast.

Stereo Dissection Microscope

Lower magnification imaging is achieved with a Zeiss SV11 ("STEMI") with a Retiga 1300 digital camera, optical tunable filter for colour imaging and IP Lab for image capture. Reflected light is provided either by a ring illuminator or by two point illumination, transmitted light is provided with a continuously adjustable 100% transmittance to darkfield slider and epi-illumination for fluorescence is provided by a mercury arc lamp with filters for dapi, CFP, GFP, YFP or rhodamine/RFP.

BioRad Radiance 2000 Laser Scanning Confocal Microscope

Thin optical sections of much higher resolution than normal epi-fluorescence can be obtained from live or fixed cultured cells, *vibratome sections*, or intact tissue with colocalization of up to *three different fluorescent probes* and one reflectant probe. Collected images can be *reconstructed in 3D*, enhanced, or analyzed using a variety of techniques. Data can be readily ported to other platforms for analysis or for final presentation.

Leica SP2 AOBS Laser Scanning Confocal Microscope

True spectral imaging with thin optical sections of much higher resolution than normal epi-fluorescence can be

obtained from cultured cells, vibratome sections, or intact tissue with simultaneous colocalization of multiple fluorescent probes, reflectance and transmitted light. Collected images can be reconstructed in 3D, enhanced, or analyzed using a variety of techniques. Data can be readily ported to other platforms for analysis or for final presentation. This instrument also has powerful capabilities for FRAP, *FRET* and time lapse applications

Leica SP5 AOBS Laser Scanning Confocal Microscope

Newer model of SP2. Arriving late December 2007.

Zeiss Live/DUO Confocal Microscope

High speed confocal microscope designed specifically for photoactivation or bleaching via a separate scanner than the imaging path. This confocal with a 100 mW laser at 489 nm and 50 mW lasers at 405 and 561 nm will be used primarily for live cell applications.

PerkinElmer UltraVIEW RS-3 Spinning Disk Laser Confocal Microscope

Preferable for imaging live cell cultures due to reduced phototoxcity, thin optical sections may be imaged as time lapse volumes. The 9 fps full field 12 bits imaging system has laser lines at 488, 568 and 647 nm for exciting three popular ranges of fluorescent probes, a piezo for high speed and reproducable Z axis control, environmental control and Nikon optics.

Expected move to new satellite facility on second floor of Michael F. Price Centre for Genetic and Translational

Medicine in the Harold and Muriel Block Research Pavilion in January/February 2008.

Multi Photon Confocal Microscopy

Multi-photon microscopy relies on excitation of fluorophores or harmonic generation by femptosecond pulses of highly concentrated long wavelength light.

Practically, this allows for imaging multiple fluorescent wavelengths deep in live tissue. The system reduces bleaching and other problems endemic to epi-fluorescent microscopy, may be more sensitive due to its lack of a confocal pinhole, and solves other problems of light scatter. *The instrument is at the centre of the intravital imaging programme at AECOM.*

FRET

Fluorescence Resonance Energy Transfer, the transfer of energy from a donor fluorophore within 7 nm of an acceptor fluorophore, can be used to measure binding interactions between and within molecules. The AIF provides acceptor bleaching for FRET imaging and measurements on three confocal microscopes and ratio FRET with widefield microscopy. Widefield FLIM in the time domain using a gated cooled CCD camera with LaVision software may be available.

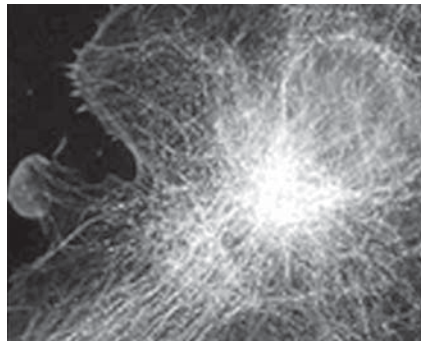
TIRF

Total Internal Reflection Fluorescent microscopy provides excitation of fluorophores only within 100 nm of the subtrate. Therefore, only molecules immediately apposed to the coverslip are excited and imaged. Objective illuminated TIRF is provided on an Olympus IX71 with either a 60X or 100X

N.A. 1.45 or a 100X N.A. 1.65 with capability to do TIRF/FRET using probes over the visible spectrum from CFP through red. Image collection and automated shuttering are provided with an Andor EM camera and Uniblitz shutters running under IPLab.

D.I.C. (Nomarski), Darkfield, Phase Contrast, IRM, and Epifluorescence with Digital Imaging

Four inverted microscope stations for high spatial resolution, wide dynamic range (low light to bright light) with *time lapse* and deconvolution capabilities. 12 bit Cooke Sensicam QE cooled CCD cameras mounted on high efficiency throughput Olympus IX70 or IX81 inverted microscopes with state of the art infinity corrected optics.



May be used to collect multiple fluorescent probes and transmitted light (brightfield, phase contrast or Nomarski) images with IPLab software running on PCs. Environmental chambers for the Olympus microscopes are available for *in vivo* work.

Many applications including spot photometry. Also, focus motors for collection of serial sections for deconvolution. Deconvolution produces images that are confocal-like in their resolution but may have a benefit of imaging weak signal or a wide dynamic range. Standard fluorescent filters include

FITC, rhodamine, Cy3, Cy5, Dapi, GFP, CFP, YFP among others more esoteric ones and a 50/50 mirror for *IRM*.

Exhaustive Photon Reassignment (EPR)

EPR deconvolution complements the other deconvolution techniques offered at the AIF by providing preservation of the total energy of the sampled volume for quantitative analysis of very dim specimens. Automated imaging of multiply probed serial optical sections is performed with a piezo controller and a Photometrics 15 bit cooled CCD camera on a high efficiency upright Olympus microscope. This is a specialty technique to image single or few molecules with precise locating within 70 nm.

Microinjection

Microinjection is a method to deliver solutions (proteins, DNA or RNA, other chemicals) directly into individual cells in culture. The AIF has two automated Eppendorf systems for use on any inverted microscopes in the Facility including the confocal, multirphoton, and other digital imaging systems.

Motion Analysis

High speed (200 FPS under bright illumination), real time (30 FPS video), or time lapse (approx. 100 ms to hours) imaging with fluorescence and transmitted light can be performed on inverted microscopes with temperature regulated environmental chambers.

Images can be made into movies for video or web presentations. Sophisticated morphometric measurements may be made over time. Quantification of images includes intensity changes of fluorescence, changes in cell or particle

velocity, direction, shape and size over time and schematic visualization of such changes. In some cases, volume changes can be measured.

Volume Rendering and 3D Quantitation

For 3D rendering or reconstructions the staff operate and train Imaris Bitplane, Voxx, a number of plugins within ImageJ and Volocity. The staff train investigators in more simple 3D imaging and quantitation via analysis of serial sections with ImageJ and I.P. Lab including the authoring of scripts for automation and result reporting.

Single Photon Uncaging

Uncaging is the activation by removing a photo-labile blocking group from DNA, RNA, protein or small molecules. The uncaging station consists of an Olympus IX70, two Hg arc lamps for UV uncaging and epifluorescence, UV corrected and phase contrast optics for uncaging and viewing cell behaviour, shutters for high speed and timed uncaging and image collection, and a Cooke Sensicam for recording uncaged fluorescence. This system shares a microscope with a *microinjection apparatus* for ease of loading cells for live experiments. A 337 nm laser with has been purchased for the system and is under development on a separate microscope stand in the Biophotonics Innovation Laboratory.

Hard Copy and Presentation

On all imaging platforms, digitized picture files are in standard formats and can be converted easily to other formats; data can be exported to other computer systems or reproduced on a variety of hard copy devices.

Adobe Photoshop CS is most widely used for figure preparation and we are happy to assist. A Fujix Pictrography 3000 colour printer makes continuous tone output at 400 PPI indistinguishable from real photographs.

Standard laser printing can be used for draft grayscale as well as for crisp graphs and text. Both still images and moving sequences can be prepared for web presentation. The AIF maintains in its inventory tools for video; however, use is by special appointment as video is being phased out.

HIGH-THROUGHPUT CRYO-TRANSMISSION ELECTRON MICROSCOPE (TEM)

Scientific Drivers

Future advances in the biosciences will depend heavily on the ability to link cell biology and structural biology through a comprehensive understanding of the structure and function of individual molecular machinery.

For instance, membrane proteins account for approximately 20-30% of the proteome and form the responsive interface between cellular and sub-cellular compartments and their environment. Thus, one of the great challenges of cell biology, proteomics and structural biology

is atomic-resolution structure determination of membrane protein complexes and dynamic macromolecular assemblies, including whole viruses.

A high-throughput cryo-TEM platform is the only possible tool for elucidating the three-dimension- al structure of entire cells with sufficient resolution to examine the arrangement and interactions of internal macromolecular complexes. Examples of the types of work that this facility will enable include:

- Determining how the 3-D biology of the human islet/ beta cell relates to the development, physiology and dysfunction of the human endocrine pancreas and its role in diabetes.
- Characterising caveolae—which have been linked to cancer, cholesterol regulation and muscular dystrophy—by using electron tomography and immunolabelling methods. Revealing the structure of membrane proteins for the design of new and highly specific drugs. With the ability to probe the molecular basis of disease.

Capabilities and the National Research Capacity

The cryo-TEM facility at the University of Queens-land is the only fully established life-sciences facility in Australia or New Zealand capable of collecting and processing atomic-resolution images at low temperatures (-160°C), as well as undertaking 3-D electron tomography of organelles, cells and tissues at ambient and low temperatures.

As such, it is one of only a handful of such facilities in the world. The NCRIS investment will build on the flagship FEI

Tecnai F30 microscope to create a high-throughput platform capable of the high output essential to make genuine inroads into key questions in molecular biology, medicine and biotechnology. The completed system will offer Australian researchers a quantum leap in technology and productivity for the emerging techiques of electron tomography, electron crystallography and single particle analysis.

The high-throughput system will be sited in custom-built laboratories within the Australian Institute for Bioengineering and Nanotechnology (AIBN) and the Queensland Biosciences Precinct (QBP). Practically, this is necessary to build on the previous investments in the FEI Tecnai F30 and associated sample prepara- tion and screening equipment and laboratories.

Scientifically, this builds on the unparalleled experience within the Centre for Microscopy and Microanalysis in cryo-electron microscopy, particularly the recent developments in rapid image-processing pipelines for single particle analysis, electron crystallography and electron tomography.

Moreover, it provides easy access for the approximately 1000 researchers from the Institute for Molecular Bioscience, the new AIBN and the Queensland Brain Institute (QBI).

Nature and Level of Demand

Since commissioning of the Tecnai F30, more than 120 national and international research projects have been undertaken and the instrument is presently near fully usage.

Given the proximity to major existing and new research centres in biosciences and nanotechnology, demand is expected to rapidly increase with the new high-throughput capability. Expected usage is approximately 20 projects (3000 hours) per annum.

Understanding how the SEM Works and how to use it on a College Level

- Electron Source
- Electromagnetic Lens
- Electron Optical Column
- Ray Diagrams
- Electron Beam/Specimen Interactions
- Vacuum
- Specimen Chamber
- Specimen Preparation

The electron beam comes from a filament, made of various types of materials. The most common is the Tungsten hairpin gun. This filament is a loop of tungsten which functions as the cathode. A voltage is applied to the loop, causing it to heat up. The anode, which is positive with respect to the filament, forms powerful attractive forces for electrons. This causes electrons to accelerate toward the anode. Some accelerate right by the anode and on down the column, to the sample. Other examples of filaments are Lanthanum Hexaboride filaments and field emission guns.

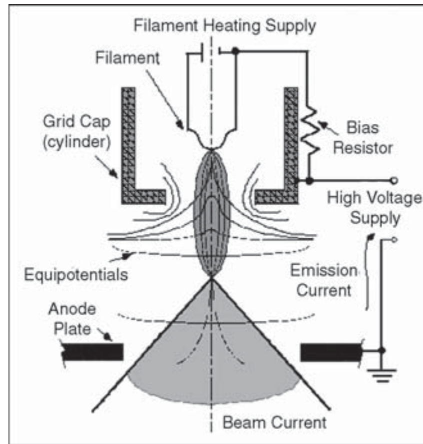


Fig. Forces in a Cylindrical Magnetic Lens

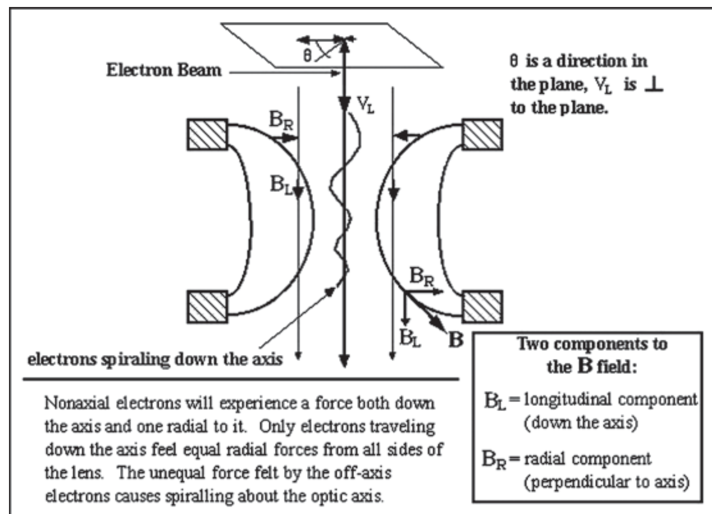
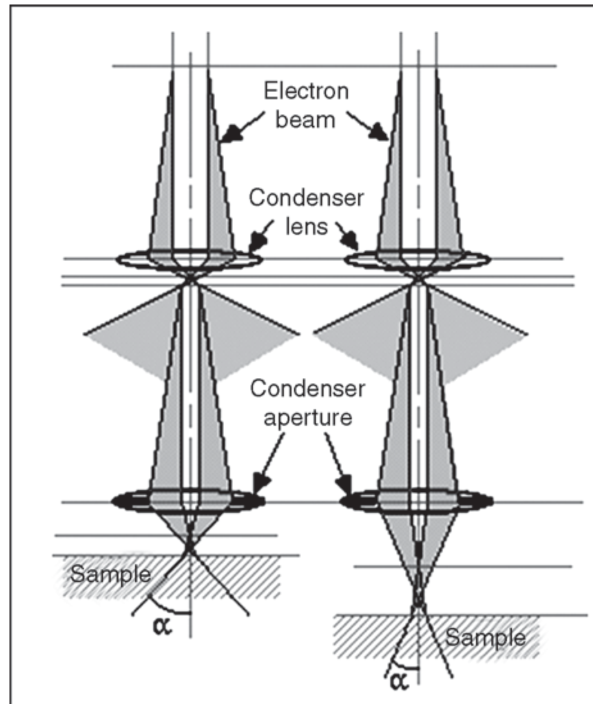


Fig. Beam's Path through the Column

A beam of electrons is generated in the electron gun, located at the top of the column, which is pictured to the left. This beam is attracted through the anode, condensed by a condenser lens, and focused as a very fine point on the sample by the objective lens. The scan coils are energized (by varying the voltage produced by the scan generator) and create a magnetic field which deflects the beam back and forth in a controlled pattern.



The varying voltage is also applied to the coils around the neck of the Cathode-ray tube (CRT) which produces a pattern of light deflected back and forth on the surface of the CRT. The pattern of deflection of the electron beam is the same as the pattern of deflection of the spot of light on the CRT.

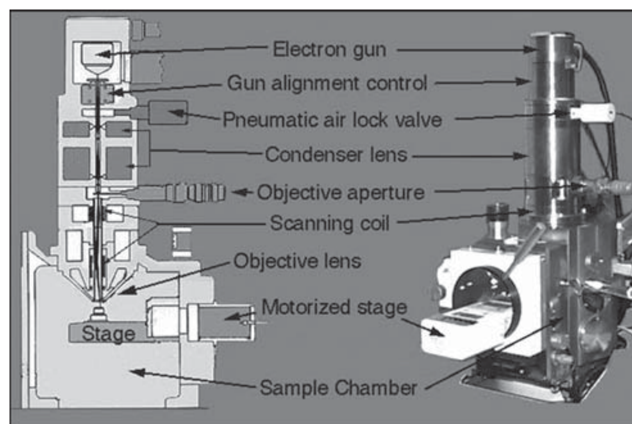


Fig. SEM Ray Diagrams

The electron beam hits the sample, producing secondary electrons from the sample. These electrons are collected by

a secondary detector or a backscatter detector, converted to a voltage, and amplified. The amplified voltage is applied to the grid of the CRT and causes the intensity of the spot of light to change. The image consists of thousands of spots of varying intensity on the face of a CRT that correspond to the topography of the sample.

These schematics show the ray traces for two probe-forming lens focusing conditions: small working distance (left) and large working distance (right). Both conditions have the same condenser lens strength and aperture size. However, as the sample is moved further from the lens, the following occurs:

- The working distance S is increased.
- The demagnification decreases.
- The spot size increases.
- The divergence angle *alpha* is decreased.

The decrease in demagnification is obtained when the lens current is decreased, which in turn increases the focal length f of the lens.

The resolution of the specimen is decreased with an increased working distance, because the spot size is increased. Conversely, the depth of field is increased with an increased working distance, because the divergence angle is smaller.

4

Data Analysis and Design

The choice of a data representation for a problem often affects our thinking about the process. Sometimes the description of a process dictates a particular choice of representation. On other occasions, it is possible and worthwhile to explore alternatives. In any case, we must Analyse and define our data collections.

Contract, Purpose, Header

We also need a contract, a definition header, and a purpose statement. Since the generative step has no connection to the structure of the data definition, the purpose statement should not only specify what the function does but should also include a comment that explains in general terms how it works.

Function Examples

In our previous design recipes, the function examples

merely specified which output the function should produce for some given input. For algorithms, examples should illustrate how the algorithm proceeds for some given input. This helps us to design, and readers to understand, the algorithm. For functions such as move-until-out the process is trivial and doesn't need more than a few words.

Template

Our discussion suggests a general template for algorithms:

```
(define (generative-recursive-fun problem)
 cond
 [(trivially-solvable? problem)
 (determine-solution problem)]
 [else
 (combine-solutions
 ... problem...
   (generative-recursive-fun (generate-problem-
  1 problem))
    (generative-recursive-fun (generate-
problem-n problem)))]))
```

ANALYSIS OF ALGORITHMS

PROGRAMS

When analyzing a Programme in terms of efficiency, we want to look at questions such as, "How long does it take for the Programme to run?" and "Is there another approach that will get the answer more quickly?" Efficiency will always be less important than correctness; if you don't care whether a Programme works correctly, you can make it run very quickly indeed, but no one will think it's much of an achievement! On the other hand, a Programme that gives a correct answer after ten thousand years isn't very useful either, so efficiency is often an important issue.

The term "efficiency" can refer to efficient use of almost any resource, including time, computer memory, disk space, or network bandwidth. In this section, however, we will deal exclusively with time efficiency, and the major question that we want to ask about a Programme is, how long does it take to perform its task?

It really makes little sense to classify an individual Programme as being "efficient" or "inefficient." It makes more sense to compare two (correct) Programmes that perform the same task and ask which one of the two is "more efficient," that is, which one performs the task more quickly. However, even here there are difficulties.

The running time of a Programme is not well-defined. The run time can be different depending on the number and speed of the processors in the computer on which it is run and, in the case of Java, on the design of the Java Virtual Machine which is used to interpret the Programme.

It can depend on details of the compiler which is used to translate the Programme from high-level language to machine language. Furthermore, the run time of a Programme depends on the size of the problem which the Programme has to solve. It takes a sorting Programme longer to sort 10000 items than it takes it to sort 100 items. When the run times of two Programmes are compared, it often happens that Programme A solves small problems faster than Programme B, while Programme B solves large problems faster than Programme A, so that it is simply not the case that one Programme is faster than the other in all cases.

In spite of these difficulties, there is a field of computer science dedicated to analyzing the efficiency of Programmes.

The field is known as Analysis of Algorithms. The focus is on algorithms, rather than on Programmes as such, to avoid having to deal with multiple implementations of the same algorithm written in different languages, compiled with different compilers, and running on different computers. Analysis of Algorithms is a mathematical field that abstracts away from these down-and-dirty details.

Still, even though it is a theoretical field, every working Programmemer should be aware of some of its techniques and results. This section is a very brief introduction to some of those techniques and results. Because this is not a mathematics book, the treatment will be rather informal. One of the main techniques of analysis of algorithms is asymptotic analysis. The term “asymptotic” here means basically “the tendency in the long run.” An asymptotic analysis of an algorithm’s run time looks at the question of how the run time depends on the size of the problem.

The analysis is asymptotic because it only considers what happens to the run time as the size of the problem increases without limit; it is not concerned with what happens for problems of small size or, in fact, for problems of any fixed finite size. Only what happens in the long run, as the problem size increases without limit, is important.

Showing that Algorithm A is asymptotically faster than Algorithm B doesn’t necessarily mean that Algorithm A will run faster than Algorithm B for problems of size 10 or size 1000 or even size 1000000 — it only means that if you keep increasing the problem size, you will eventually come to a point where Algorithm A is faster than Algorithm B. An asymptotic analysis is only a first approximation, but in

practice it often gives important and useful information. Using this notation, we might say, for example, that an algorithm has a running time that is $O(n^2)$ or O(n) or O(log(n)). These notations are read "Big-Oh of n squared," "Big-Oh of n," and "Big-Oh of log n" (where log is a logarithm function). More generally, we can refer to O(f(n)) ("Big-Oh of f of n"), where f(n) is some function that assigns a positive real number to every positive integer n. The "n" in this notation refers to the size of the problem.

Before you can even begin an asymptotic analysis, you need some way to measure problem size. Usually, this is not a big issue. For example, if the problem is to sort a list of items, then the problem size can be taken to be the number of items in the list. When the input to an algorithm is an integer, as in the case of an algorithm that checks whether a given positive integer is prime, the usual measure of the size of a problem is the number of bits in the input integer rather than the integer itself. More generally, the number of bits in the input to a problem is often a good measure of the size of the problem.

To say that the running time of an algorithm is O(f(n)) means that for large values of the problem size, n, the running time of the algorithm is no bigger than some constant times f(n). (More rigorously, there is a number C and a positive integer M such that whenever n is greater than M, the run time is less than or equal to C*f(n).) The constant takes into account details such as the speed of the computer on which the algorithm is run; if you use a slower computer, you might have to use a bigger constant in the formula, but changing the constant won't change the basic

fact that the run time is O(f(n)). The constant also makes it unnecessary to say whether we are measuring time in seconds, years, CPU cycles, or any other unit of measure; a change from one unit of measure to another is just multiplication by a constant. Note also that O(f(n)) doesn't depend at all on what happens for small problem sizes, only on what happens in the long run as the problem size increases without limit. To look at a simple example, consider the problem of adding up all the numbers in an array. The problem size, n, is the length of the array. Using A as the name of the array, the algorithm can be expressed in Java as:

```
total = 0;
for (int i = 0; i < n; i++)
total = total + A[i];
```

This algorithm performs the same operation, total = total + A[i], n times. The total time spent on this operation is a*n, where a is the time it takes to perform the operation once. Now, this is not the only thing that is done in the algorithm. The value of i is incremented and is compared to n each time through the loop.

This adds an additional time of b*n to the run time, for some constant b. Furthermore, i and total both have to be initialized to zero; this adds some constant amount c to the running time.

The exact running time would then be (a+b)*n+c, where the constants a, b, and c depend on factors such as how the code is compiled and what computer it is run on. Using the fact that c is less than or equal to c*n for any positive integer n, we can say that the run time is less than or equal to (a+b+c)*n. That is, the run time is less than or equal to a

constant times n. By definition, this means that the run time for this algorithm is O(n).

If this explanation is too mathematical for you, we can just note that for large values of n, the c in the formula (a+b)*n+c is insignificant compared to the other term, (a+b)*n. We say that c is a "lower order term." When doing asymptotic analysis, lower order terms can be discarded. A rough, but correct, asymptotic analysis of the algorithm would go something like this: Each iteration of the for loop takes a certain constant amount of time. There are n iterations of the loop, so the total run time is a constant times n, plus lower order terms (to account for the initialization). Disregarding lower order terms, we see that the run time is O(n).

Note that to say that an algorithm has run time O(f(n)) is to say that its run time is no bigger than some constant times f(n) (for large values of n). O(f(n)) puts an upper limit on the run time. However, the run time could be smaller, even much smaller. For example, if the run time is O(n), it would also be correct to say that the run time is $O(n^2)$ or even $O(n^{10})$. If the run time is less than a constant times n, then it is certainly less than the same constant times n^2 or n^{10}.

Of course, sometimes it's useful to have a lower limit on the run time. That is, we want to be able to say that the run time is greater than or equal to some constant times f(n) (for large values of n). The notation for this is Ω(f(n)), read "Omega of f of n." "Omega" is the name of a letter in the Greek alphabet, and Ω is the upper case version of that letter. (To be technical, saying that the run time of an algorithm is

$\Omega(f(n))$ means that there is a positive number C and a positive integer M such that whenever n is greater than M, the run time is greater than or equal to C*f(n).) O(f(n)) tells you something about the maximum amount of time that you might have to wait for an algorithm to finish; $\Omega(f(n))$ tells you something about the minimum time.

The algorithm for adding up the numbers in an array has a run time that is $\Omega(n)$ as well as O(n). When an algorithm has a run time that is both $\Omega(f(n))$ and O(f(n)), its run time is said to be $\Theta(f(n))$, read "Theta of f of n." (Theta is another letter from the Greek alphabet.) To say that the run time of an algorithm is $\Theta(f(n))$ means that for large values of n, the run time is between a*f(n) and b*f(n), where a and b are constants (with b greater than a, and both greater than 0).

Let's look at another example. Consider the algorithm that can be expressed in Java in the following method:

```
 /**
 * Sorts the n array elements A[0], A[1]...,      A[n-1]
into increasing order.
 */
 public static simpleBubbleSort(int[] A, int      n) {
  for (int i = 0; i < n; i++) {
 // Do n passes through the array...
  for (int j = 0; j < n-1; j++) {
   if (A[j] > A[j+1]) {
 // A[j] and A[j+1] are out of order, so swap     them
      int temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
   }
  }
 }
}
```

Here, the parameter n represents the problem size. The outer for loop in the method is executed n times. Each time the outer for loop is executed, the inner for loop is exectued

n-1 times, so the if statement is executed n*(n-1) times. This is n^2-n, but since lower order terms are not significant in an asymptotic analysis, it's good enough to say that the if statement is executed about n^2 times.

In particular, the test A[j] > A[j+1] is executed about n^2times, and this fact by itself is enough to say that the run time of the algorithm is W(n^2), that is, the run time is at least some constant times n^2. Furthermore, if we look at other operations — the assignment statements, incrementing i and j, etc. — none of them are executed more than n^2 times, so the run time is also O(n^2), that is, the run time is no more than some constant times n^2. Since it is both W(n^2) and O(n^2), the run time of the simpleBubbleSort algorithm is Q(n^2).

You should be aware that some people use the notation O(f(n)) as if it meant Θ(f(n)). That is, when they say that the run time of an algorithm is O(f(n)), they mean to say that the run time is about equal to a constant times f(n). For that, they should use Θ(f(n)). Properly speaking, O(f(n)) means that the run time is less than a constant times f(n), possibly much less.

So far, the analysis has ignored an important detail. We have looked at how run time depends on the problem size, but in fact the run time usually depends not just on the size of the problem but on the specific data that has to be processed. For example, the run time of a sorting algorithm can depend on the initial order of the items that are to be sorted, and not just on the number of items.

To account for this dependency, we can consider either the worst case run time analysis or the average case run

time analysis of an algorithm. For a worst case run time analysis, we consider all possible problems of size n and look at the longest possible run time for all such problems. For an average case analysis, we consider all possible problems of size n and look at the average of the run times for all such problems. Usually, the average case analysis assumes that all problems of size n are equally likely to be encountered, although this is not always realistic — or even possible in the case where there is an infinite number of different problems of a given size.

In many cases, the average and the worst case run times are the same to within a constant multiple. This means that as far as asymptotic analysis is concerned, they are the same. That is, if the average case run time is $O(f(n))$ or $\Theta(f(n))$, then so is the worst case. However, later in the book, we will encounter a few cases where the average and worst case asymptotic analyses differ.

So, what do you really have to know about analysis of algorithms to read the rest of this book? We will not do any rigorous mathematical analysis, but you should be able to follow informal discussion of simple cases such as the examples that we have looked at in this section.

Most important, though, you should have a feeling for exactly what it means to say that the running time of an algorithm is $O(f(n))$ or $\Theta(f(n))$ for some common functions $f(n)$. The main point is that these notations do not tell you anything about the actual numerical value of the running time of the algorithm for any particular case.

They do not tell you anything at all about the running time for small values of n. What they do tell you is something

about the rate of growth of the running time as the size of the problem increases.

Suppose you compare two algorithms that solve the same problem. The run time of one algorithm is $\Theta(n^2)$, while the run time of the second algorithm is $\Theta(n^3)$. What does this tell you? If you want to know which algorithm will be faster for some particular problem of size, say, 100, nothing is certain. As far as you can tell just from the asymptotic analysis, either algorithm could be faster for that particular case — or in any particular case. But what you can say for sure is that if you look at larger and larger problems, you will come to a point where the $\Theta(n^2)$ algorithm is faster than the $\Theta(n^3)$ algorithm. Furthermore, as you continue to increase the problem size, the relative advantage of the $\Theta(n^2)$ algorithm will continue to grow. There will be values of n for which the $\Theta(n^2)$ algorithm is a thousand times faster, a million times faster, a billion times faster, and so on. This is because for any positive constants a and b, the function $a*n^3$ grows faster than the function $b*n^2$ as n gets larger. (Mathematically, the limit of the ratio of $a*n^3$ to $b*n^2$ is infinite as n approaches infinity.)

This means that for "large" problems, a $\Theta(n^2)$ algorithm will definitely be faster than a $\Theta(n^3)$ algorithm. You just don't know — based on the asymptotic analysis alone — exactly how large "large" has to be. In practice, in fact, it is likely that the $\Theta(n^2)$ algorithm will be faster even for fairly small values of n, and absent other information you would generally prefer a $\Theta(n^2)$ algorithm to a $\Theta(n^3)$ algorithm.

So, to understand and apply asymptotic analysis, it is essential to have some idea of the rates of growth of some

common functions. For the power functions n, n^2, n^3, n^4..., the larger the exponent, the greater the rate of growth of the function.

Exponential functions such as 2^n and 10^n, where the n is in the exponent, have a growth rate that is faster than that of any power function. In fact, exponential functions grow so quickly that an algorithm whose run time grows exponentially is almost certainly impractical even for relatively modest values of n, because the running time is just too long.

Another function that often turns up in asymptotic analysis is the logarithm function, log(n). There are actually many different logarithm functions, but the one that is usually used in computer science is the so-called logarithm to the base two, which is defined by the fact that $\log(2^x) = x$ for any number x. (Usually, this function is written $\log_2(n)$, but I will leave out the subscript 2, since I will only use the base-two logarithm in this book.)

The logarithm function grows very slowly. The growth rate of log(n) is much smaller than the growth rate of n. The growth rate of n*log(n) is a little larger than the growth rate of n, but much smaller than the growth rate of n^2.

POINTS

- *Introduction*: Analysis of Selection Sort
- *Introduction*: Analysis of Merge Sort
- Asymptotic Notation
- Asymptotic Notation Continued
- Heapsort
- Heapsort Continued

- Priority Queues (more heaps)
- Quicksort
- Bounds on Sorting and Linear Time Sorts
- Stable Sorts and Radix Sort
- Begin Dynamic Programmeming
- More Dynamic Programmeming
- *Begin Greedy Algorithms*: Huffman's Algorithm
- Dÿkstra's Algorithm
- Beyond Asymptotic Analysis: Memory Access Time
- B-Trees
- More B-Trees: Insertion and Splitting
- Union/Find
- Warshall's Algorithm, Floyd's Algorithm
- Large Integer Arithmetic
- RSA Public-Key Cryptosystem
- Begin Algorithms and Structural Complexity Theory
- Continue Algorithms and Structural Complexity Theory
- End Algorithms and Structural Complexity Theory
- Generating Permutations and Combinations
- Exam review with sample questions and solutions

AUGMENTING-PATH ALGORITHMS

The neat part of the Ford-Fulkerson algorithm described above is that it gets the correct result no matter how we solve (correctly!!) the sub-problem of finding an augmenting path. However, every new path may increase the flow by only 1, hence the number of iterations of the algorithm could be

very large if we carelessly choose the augmenting path algorithm to use. The function *max_flow* will look like this, regardless of the actual method we use for finding augmenting paths:

```
  int max_flow()
    result = 0
       while (true)
   // the function find_path returns the path
capacity of the
   augmenting path found
   path_capacity = find_path()
   // no augmenting path found
   if (d = 0) exit while
   else result += path_capacity
 end while
 return result
```

To keep it simple, we will use a 2-dimensional array for storing the capacities of the residual network that we are left with after each step in the algorithm. Initially the residual network is just the original network. We will not store the flows along the edges explicitly, but it's easy to figure out how to find them upon the termination of the algorithm: for each edge x-y in the original network the flow is given by the capacity of the backward edge y-x in the residual network. Be careful though; if the reversed arc y-x also exists in the original network, this will fail, and it is recommended that the initial capacity of each arc be stored somewhere, and then the flow along the edge is the difference between the initial and the residual capacity.

We now require an implementation for the function *find_path*. The first approach that comes to mind is to use a depth-first search (DFS), as it probably is the easiest to implement. Unfortunately, its performance is very poor on some networks, and normally is less preferred to the ones

discussed next. The next best thing in the matter of simplicity is a breadth-first search (BFS). Recall that this search usually yields the shortest path in an un-weighted graph. Indeed, this also applies here to get the shortest augmenting path from the source to the sink. In the following pseudocode we will basically: find a shortest path from the source to the sink and compute the minimum capacity of an edge (that could be a forward or a backward edge) along the path - the path capacity. Then, for each edge along the path we reduce its capacity and increase the capacity of the reversed edge with the path capacity.

```
    int bfs()
      queue Q
       push source to Q
       mark source as visited
       keep an array from with the
   semnification: from[x] is the
       previous vertex visited in the shortest
     path from the source to x;
       initialize from with -1 (or any other
    sentinel value)
        while Q is not empty
          where = pop from Q
        for each vertex next adjacent to where
        if next is not visited and
    capacity[where][next] > 0
          push next to Q
          mark next as visited
          from[next] = where
          if next = sink
            exit while loop
      end for
  end while
  // we compute the path capacity
  where = sink, path_cap = infinity
  while from[where] > -1
     prev = from[where]// the previous vertex
     path_cap = min(path_cap,
      capacity[prev][where])
     where = prev
```

```
end while
   // we update the residual network; if no
path is found the while
   loop will not be entered
   where = sink
      while from[where] > -1
        prev = from[where]
        capacity[prev][where] -= path_capacity
           capacity[where][prev] +=
       path_capacity
   where = prev
end while
// if no path is found, path_cap is infinity
if path_cap = infinity
   return 0
else return path_cap
```

As we can see, this is pretty easy to implement. As for its performance, it is guaranteed that this takes at most $N * M/2$ steps, where N is the number of vertices and M is the number of edges in the network. This number may seem very large, but it is over-estimated for most networks. For example, in the network we considered 3 augmenting paths are needed which is significantly less than the upper bound of 28. Due to the $O(M)$ running time of BFS (implemented with adjacency lists) the worst-case running time of the shortest-augmenting path max-flow algorithm is $O(N * M^2)$, but usually the algorithm performs much better than this.

Next we will consider an approach that uses a priority-first search (PFS), that is very similar to the Dijkstra heap method explained here. In this method the augmenting path with a maximum path capacity is preferred. Intuitively this would lead to a faster algorithm, since at each step we increase the flow with the maximum possible amount. However, things are not always so, and the BFS implementation has better running times on some networks.

We assign as a priority to each vertex the minimum capacity of a path (in the residual network) from the source to that vertex. We process vertices in a greedy manner, as in Dijkstra's algorithm, in decreasing order of priorities. When we get to the sink, we are done, since a path with a maximum capacity is found. We would like to implement this with a data structure that allows us to efficiently find the vertex with the highest priority and increase the priority of a vertex (when a new better path is found) - this suggests the use of a heap which has a space complexity proportional to the number of vertices.

In TopCoder matches we may find it faster and easier to implement this with a priority queue or some other data structure that approximates one, even though the space required might grow to being proportional with the number of edges. This is how the following pseudocode is implemented. We also define a structure node that has the members vertex and prioritywith the above significance. Another field from is needed to store the previous vertex on the path.

```
int pfs()
 priority queue PQ
 push node(source, infinity, -1) to PQ
 keep the array from as in bfs()
 // if no augmenting path is found, path_cap
will remain 0
 path_cap = 0
 while PQ is not empty
     node aux = pop from PQ
     where = aux.vertex, cost = aux.priority
     if we already visited where continue
     from[where] = aux.from
     if where = sink
       path_cap = cost
       exit while loop
```

```
        mark where as visited
        for each vertex next adjacent to where
           if capacity[where][next] > 0
              new_cost = min(cost,
             capacity[where][next])
              push node(next, new_cost, where) to
           PQ
       end for
   end while
   // update the residual network
   where = sink
   while from[where] > -1
      prev = from[where]
      capacity[prev][where] -= path_cap
      capacity[where][prev] += path_cap
      where = prev
   end while
      return path_cap
```

The analysis of its performance is pretty complicated, but it may prove worthwhile to remember that with PFS at most $2M1gU$ steps are required, where U is the maximum capacity of an edge in the network. As with BFS, this number is a lot larger than the actual number of steps for most networks. Combine this with the $O(M\ 1g\ M)$complexity of the search to get the worst-case running time of this algorithm.

Now that we know what these methods are all about, which of them do we choose when we are confronted with a max-flow problem? The PFS approach seems to have a better worst-case performance, but in practice their performance is pretty much the same. So, the method that one is more familiar with may prove more adequate. Personally, I prefer the shortest-path method, as I find it easier to implement during a contest and less error prone.

MAXIMUM FLOW PROBLEM

The maximum flow problem is again structured on a network; but here the arc capacities, or upper bounds, are

the only relevant parameters. The problem is to find the maximum flow possible from some given source node to a given sink node. A network model is in Fig. All arc costs are zero, but the cost on the arc leaving the sink is set to -1. Since the goal of the optimization is to minimize cost, the maximum flow possible is delivered to the sink node.

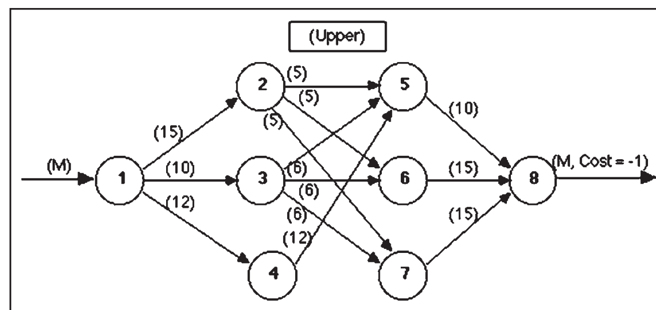


Fig. Network Model for the Maximum Flow Problem.

The solution to the example is in Fig. The maximum flow from node 1 to node 8 is 30 and the flows that yield this flow are shown on the figure. The heavy arcs on the figure are called the minimal cut.

These arcs are the bottlenecks that are restricting the maximum flow. The fact that the sum of the capacities of the arcs on the minimal cut equals the maximum flow is a famous theorem of network theory called the max flow min cut theorem. The arcs on the minimum cut can be identified using sensitivity analysis.

MAX-FLOW/MIN-CUT RELATED PROBLEMS

How to recognize max-flow problems? Often they are hard to detect and usually boil down to maximizing the movement of something from a location to another. We need to look at the constraints when we think we have a working solution based on maximum flow - they should suggest at least an

$O(N^3)$ approach. If the number of locations is large, another algorithm (such as dynamic Programmeming or greedy), is more appropriate.

The problem description might suggest multiple sources and/or sinks. For example, in the sample statement in the beginning of this article, the company might own more than one factory and multiple distribution Centres. How can we deal with this? We should try to convert this to a network that has a unique source and sink.

In order to accomplish this we will add two "dummy" vertices to our original network - we will refer to them as super-source and super-sink. In addition to this we will add an edge from the super-source to every ordinary source (a factory). As we don't have restrictions on the number of trucks that each factory can send, we should assign to each edge an infinite capacity.

Note that if we had such restrictions, we should have assigned to each edge a capacity equal to the number of trucks each factory could send. Likewise, we add an edge from every ordinary sink (distribution Centres) to the super-sink with infinite capacity.

A maximum flow in this new-built network is the solution to the problem - the sources now become ordinary vertices, and they are subject to the entering-flow equals leaving-flow property. You may want to keep this in your bag of tricks, as it may prove useful to most problems.

What if we are also given the maximum number of trucks that can drive through each of the cities in the country (other than the cities where the factory and the distribution Centre are located)? In other words we have to deal with vertex-

capacities too. Intuitively, we should be able to reduce this to maximum-flow, but we must find a way to take the capacities from vertices and put them back on edges, where they belong.

Another nice trick comes into play. We will build a network that has two times more vertices than the initial one. For each vertex we will have two nodes: an in-vertex and an out-vertex, and we will direct each edge x-y from the out-vertex of x to the in-vertex of y.

We can assign them the capacities from the problem statement. Additionally we can add an edge for each vertex from the in to the out-vertex.

The capacity this edge will be assigned is obviously the vertex-capacity. Now we just run max-flow on this network and compute the result. Maximum flow problems may appear out of nowhere. Let's take this problem for instance:"You are given the in and out degrees of the vertices of a directed graph.

Your task is to find the edges (assuming that no edge can appear more than once)." First, notice that we can perform this simple test at the beginning.

We can compute the number M of edges by summing the out-degrees or the in-degrees of the vertices. If these numbers are not equal, clearly there is no graph that could be built. This doesn't solve our problem, though.

There are some greedy approaches that come to mind, but none of them work.

We will combine the tricks discussed above to give a max-flow algorithm that solves this problem. First, build a network that has 2 (in/out) vertices for each initial vertex. Now draw

an edge from every out vertex to every in vertex. Next, add a super-source and draw an edge from it to every out-vertex. Add a super-sink and draw an edge from every in vertex to it. We now need some capacities for this to be a flow network. It should be pretty obvious what the intent with this approach is, so we will assign the following capacities: for each edge drawn from the super-source we assign a capacity equal to the out-degree of the vertex it points to.

As there may be only one arc from a vertex to another, we assign a 1 capacity to each of the edges that go from the outs to the ins.

As you can guess, the capacities of the edges that enter the super-sink will be equal to the in-degrees of the vertices. If the maximum flow in this network equals M - the number of edges, we have a solution, and for each edge between the out and in vertices that has a flow along it (which is maximum 1, as the capacity is 1) we can draw an edge between corresponding vertices in our graph.

Note that both x-y and y-x edges may appear in the solution. This is very similar to the maximum matching in a bipartite graph that we will discuss later. An example is given below where the out-degrees are (2, 1, 1, 1) and the in-degrees (1, 2, 1, 1). Some other problems may ask to separate two locations minimally. Some of these problems usually can be reduced to minimum-cut in a network. Two examples will be discussed here, but first let's take the standard min-cut problem and make it sound more like a TopCoder problem. We learned earlier how to find the value of the min-cut and how to find an arbitrary min-cut. In addition to this we will now like to have a minimum-cut with the minimum number

of edges. An idea would be to try to modify the original network in such a way that the minimum cut here is the minimum cut with the minimum edges in the original one.

Notice what happens if we multiply each edge capacity with a constant T. Clearly, the value of the maximum flow is multiplied by T, thus the value of the minimum cut is T times bigger than the original. A minimum cut in the original network is a minimum cut in the modified one as well. Now suppose we add 1 to the capacity of each edge. Is a minimum cut in the original network a minimum cut in this one? The answer is no, as we can see in Figure shown below, if we take T = 2. Why did this happen? Take an arbitrary cut. The value of the cut will be T times the original value of the cut, plus the number of edges in it.

Thus, a non-minimum cut in the first place could become minimum if it contains just a few edges. This is because the constant might not have been chosen properly in the beginning, as is the case in the example above. We can fix this by choosing T large enough to neutralize the difference in the number of edges between cuts in the network.

In the above example T = 4 would be enough, but to generalize, we take T = 10, one more than the number of edges in the original network, and one more than the number of edges that could possibly be in a minimum-cut. It is now true that a minimum-cut in the new network is minimum in the original network as well. However the converse is not true, and it is to our advantage. Notice how the difference between minimum cuts is now made by the number of edges in the cut. So we just find the min-cut in this new network to solve the problem correctly.

Let's illustrate some more the min-cut pattern: "An undirected graph is given. What is the minimum number of edges that should be removed in order to disconnect the graph?" In other words the problem asks us to remove some edges in order for two nodes to be separated. This should ring a bell - a minimum cut approach might work. So far we have only seen maximum flow in directed graphs, but now we are facing an undirected one.

This should not be a very big problem though, as we can direct the graph by replacing every (undirected) edge x-y with two arcs: x-y and y-x. In this case the value of the min-cut is the number of edges in it, so we assign a 1 capacity to each of them. We are not asked to separate two given vertices, but rather to disconnect optimally any two vertices, so we must take every pair of vertices and treat them as the source and the sink and keep the best one from these minimum-cuts.

An improvement can be made, however. Take one vertex, let's say vertex numbered 1. Because the graph should be disconnected, there must be another vertex unreachable from it. So it suffices to treat vertex 1 as the source and iterate through every other vertex and treat it as the sink.

What if instead of edges we now have to remove a minimum number of vertices to disconnect the graph? Now we are asked for a different min-cut, composed of vertices. We must somehow convert the vertices to edges though. Recall the problem above where we converted vertex-capacities to edge-capacities. The same trick works here. First "un-direct" the graph as in the previous example.

Next double the number of vertices and deal edges the same way: an edge x-y is directed from the out-x vertex to in-y. Then convert the vertex to an edge by adding a 1-capacity arc from the in-vertex to the out-vertex.

Now for each two vertices we must solve the sub-problem of minimally separating them. So, just like before take each pair of vertices and treat the out-vertex of one of them as the source and the in-vertex of the other one as the sink (this is because the only arc leaving the in-vertex is the one that goes to the out-vertex) and take the lowest value of the maximum flow. This time we can't improve in the quadratic number of steps needed, because the first vertex may be in an optimum solution and by always considering it as the source we lose such a case.

5

Combinational Logic Design

INTRODUCTION

Combinational logic is probably the easiest circuitry to design. The outputs from a combinational logic circuit depend only on the current inputs. The circuit has no remembrance of what it did at any time in the past. Much of logic design involves connecting simple, easily understood circuits to construct a larger circuit that performs a much more complicated function. Several simple, often-used combinational logic circuits are the following:

Analysis Procedure

The bulk of the Combinational Analysis module is accessed through a single window of that name allowing you to view truth tables and Boolean expressions. This window can be opened in two ways.

Via the Window Menu

Select Combinational Analysis, and the current Combinational Analysis window will appear. If you haven't viewed the window before, the opened window will represent no circuit at all. Only one Combinational Analysis window exists within Logisim, no matter how many projects are open. There is no way to have two different analysis windows open at once.

Via the Project Menu

From a window for editing circuits, you can also request that Logisim analyze the current circuit by selecting the Analyze Circuit option from the Project menu. Before Logisim opens the window, it will compute Boolean expressions and a truth table corresponding to the circuit and place them there for you to view. For the analysis to be successful, each input must be attached to an input pin, and each output must be attached to an output pin. Logisim will only analyze circuits with at most eight of each type, and all should be single-bit pins. Otherwise, you will see an error message and the window will not open.

In constructing Boolean expressions corresponding to a circuit, Logisim will first attempt to construct a Boolean expressions corresponding exactly to the gates in the circuit. But if the circuit uses some non-gate components (such as a multiplexer), or if the circuit is more than 100 levels deep (unlikely), then it will pop up a dialog box telling you that deriving Boolean expressions was impossible, and Logical will instead derive the expressions based on the truth table, which will be derived by quietly trying each combination of inputs and reading the resulting outputs.

After analyzing a circuit, there is no continuing relationship between the circuit and the Combinational Analysis window. That is, changes to the circuit will not be reflected in the window, nor will changes to the Boolean expressions and/or truth table in the window be reflected in the circuit. Of course, you are always free to analyze a circuit again; and, as we will see later, you can replace the circuit with a circuit corresponding to what appears in the Combinational Analysis window.

Limitations

Logical will not attempt to detect sequential circuits: If you tell it to analyze a sequential circuit, it will still create a truth table and corresponding Boolean expressions, although these will not accurately summarize the circuit behavior. (In fact, detecting sequential circuits is *provably impossible*, as it would amount to solving the Halting Problem. Of course, you might hope that Logical would make at least some attempt - perhaps look for flip-flops or cycles in the wires - but it does not.) As a result, the Combinational Analysis system should not be used indiscriminately: Only use it when you are indeed sure that the circuit you are analyzing is indeed combinational!

Design Procedure

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined by combining the values of the applied inputs using logic operations. A combinational circuit performs an operation that can be specified logically by a set of Boolean expression. In addition

to using logic gates, sequential circuits employ elements that store bit values. Sequential circuit outputs are a function of inputs and the bit value in storage elements. These values, in turn, are a function of previously applied inputs and stored values. As a consequence, the outputs of a sequential circuit depend not only on the presently applied values of the inputs, but also on pas inputs, and the behavior of the circuit must be specified by a sequence in time of inputs and internal stored bit values. A combinational circuit consists of input variables, output variables, logic gates and interconnections. The interconnected logic gates accept signals from the inputs and generate signals at the output. The n input variables come from the environment of the circuit, and the m output variables are available for use by the environment. Each input and output variable exists physically as a binary signal that represents logic 1 or logic 0.

For n input variables, there are 2^n possible binary input combinations. For each binary combination of the input variables, there is one possible binary value on each output. Thus, a combinational circuit can be specified by a truth table that lists the output values for each combination of the input variables. A combinational circuit can also be described by m Boolean function, one for each output variable. Each such function is expressed as function of the n input variables.

Combinational Circuit Design

The design of combinational circuit starts from a specification of the problem and culminates in a logic diagram or set of Boolean equations from which the logic diagram can be obtained.

The procedure involves the following steps:

1. From the specifications of the circuit, determine the required number of inputs and outputs, and assign a letter symbol to each.
2. Derive the truth table that defines the required relation ship between inputs and outputs.
3. Obtain the simplified Boolean functions of each outputs as function of the input variables.
4. Draw the logic diagram.
5. Verify the correctness of the design.

Binary Adder-subtractor

Half Adder/Subtractor

This figure shows the configuration for a Half Adder/ Subtractor. The first logic gate which is a XOR allows the circuit to do the complement of the binary bit when we want to do a subtraction. In the case that is needed to implement a addition the XOR keep the number the same.

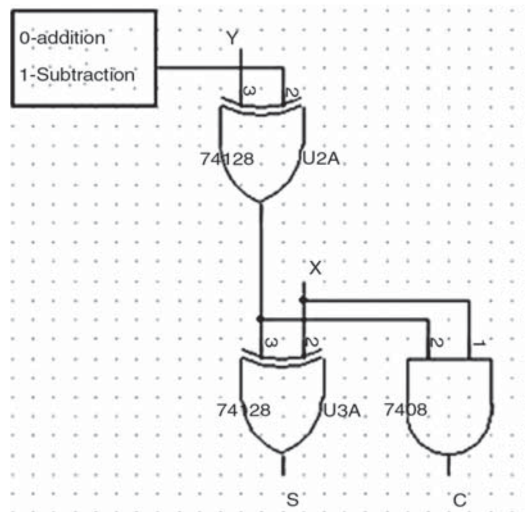


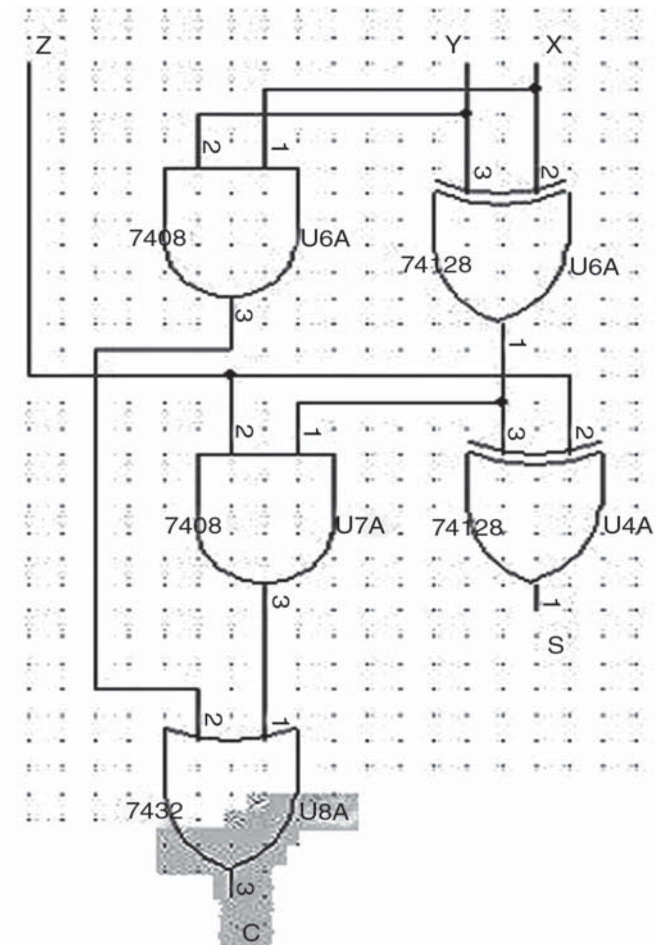
Table. Truth Table.

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder/Subtractor

Full Adder Circuit

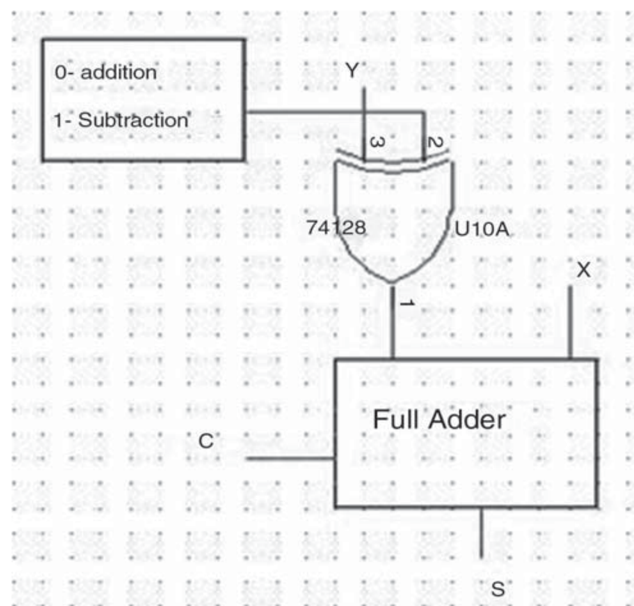
This circuit adds two binary one bit numbers. Also, it manages a carry that could come from another circuit.



[(AA)'*(BB)']' = A+B

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This figure shows the configuration for a Full Adder/ Subtractor. The box with the name of "Full Adder" has all the logic to do the addition and subtraction depending of the inputs that it receives. The XOR gate that is outside the box allows the circuit to do the complement of the binary bit when we want to do a subtraction. In the case that is needed to implement an addition the XOR keep the number the same. In the previous graph the internal logic for the box "Full Adder" is shown.



3-bit Adder/Subtractor

The 3-bit adder/Subtractor was implemented with three Full adder circuits and three XOR gates outside which

implemented the operation (addition/subtraction) selected by the user. In this circuit, we have three inputs for the first three bits binary number and three inputs for the second three bits binary number. When the addition is selected the XOR gates keep the binary numbers the same and add them together. On the other hand, when a subtraction is selected the XOR gates complement the second number which is represented with a "Y" and after that, the Full Adder circuit adds both numbers together. The 3-bit Adder/Subtractor circuit has four outputs. The first three outputs represent the three bits that were the result of the subtraction or addition. The last bit represents a carry.

Binary Multiplier

Binary multiplication uses the same technique as decimal multiplication. In fact, binary multiplication is much easier because each digit we multiply by is either zero or one. Consider the simple problem of multiplying 110_2 by 10_2. We can use this problem to review some terminology and illustrate the rules for binary multiplication.

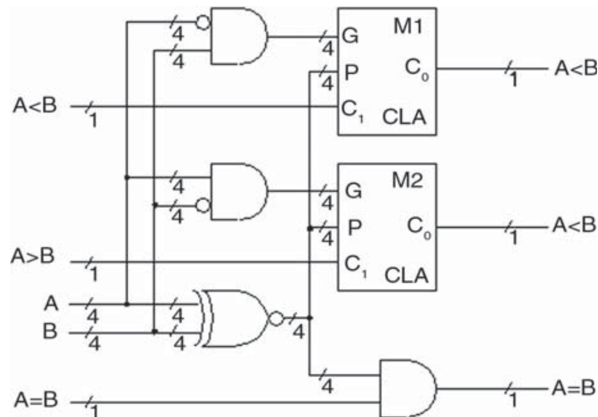
Step		Example
1.	First, we note that 110_2 is our multiplicand and 10_2 is our multiplier.	110 x 10
2.	We begin by multiplying 110_2 by the rightmost digit of our multiplier which is 0. Any number times zero is zero, so we just write zeros below.	110 x 10 000
3.	Now we multiply the multiplicand by the next digit of our multiplier which is 1. To perform this multiplication, we just need to copy the multiplicand and shift it one column to the left as we do in decimal multiplication.	110 x 10 000 110
4.	Now we add our results together. The product of our multiplication is 1100_2.	110 x 10 000 110 1100

When performing binary multiplication, remember the following rules:

1. Copy the multiplicand when the multiplier digit is 1. Otherwise, write a row of zeros.
2. Shift your results one column to the left as you move to a new multiplier digit.
3. Add the results together using binary addition to find the product.

Magnitude Comparator

74L85 4-Bit Magnitude Comparator

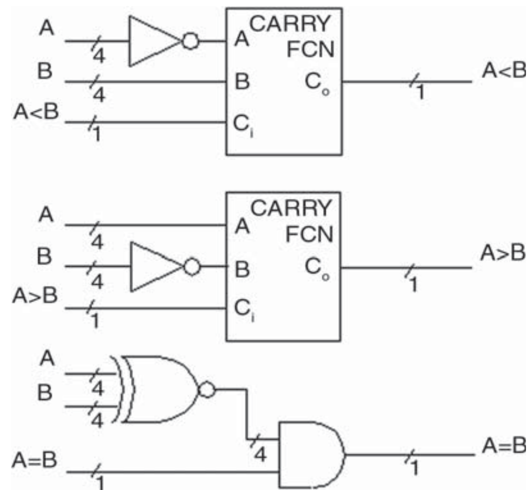


Statistics

11 inputs; 3 outputs; 33 gates;

Function

The 74L85 magnitude comparator can be functionally modeled as above. This is a simplification of implementing a magnitude comparator by a carry function with an inverted input bus as shown.



Using this concept, common elements of the three comparator functions $A < B$, $A > B$, and $A = B$ are combined to construct the model shown above, which maps directly onto the gate-level realization of the 74L85.

Decoders

Introduction

In both the multiplexer and the demultiplexer, part of the circuits *decode* the address inputs, i.e. it translates a binary number of n digits to 2^n outputs, one of which (the one that corresponds to the value of the binary number) is 1 and the others of which are 0. It is sometimes advantageous to separate this function from the rest of the circuit, since it is useful in many other applications. Thus, we obtain a new combinatorial circuit that we call the *decoder*.

It has the following truth table (for n = 3):

a2 a1 a0 | d7 d6 d5 d4 d3 d2 d1 d0

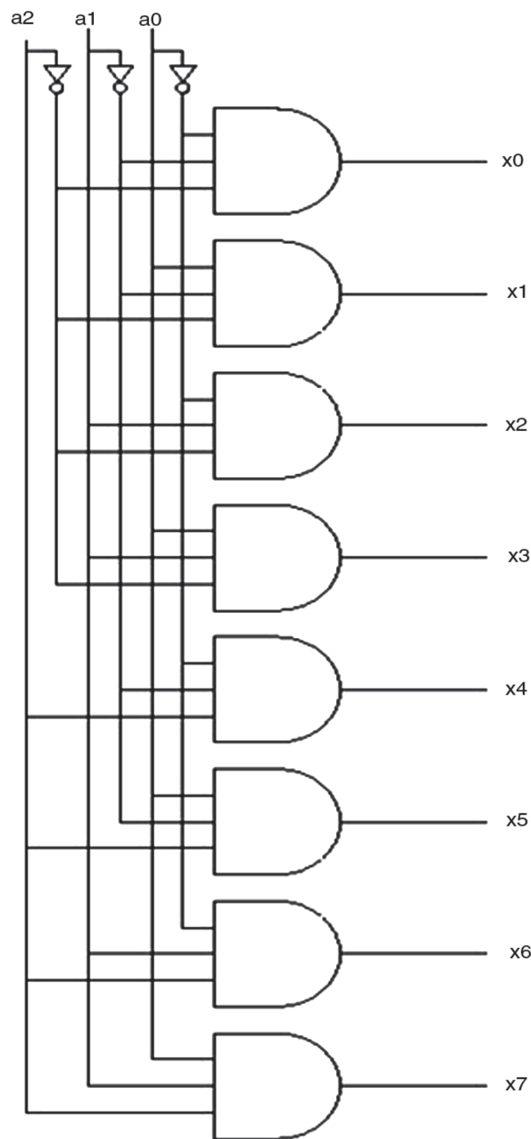
---

0 0 0 | 0 0 0 0 0 0 0 1

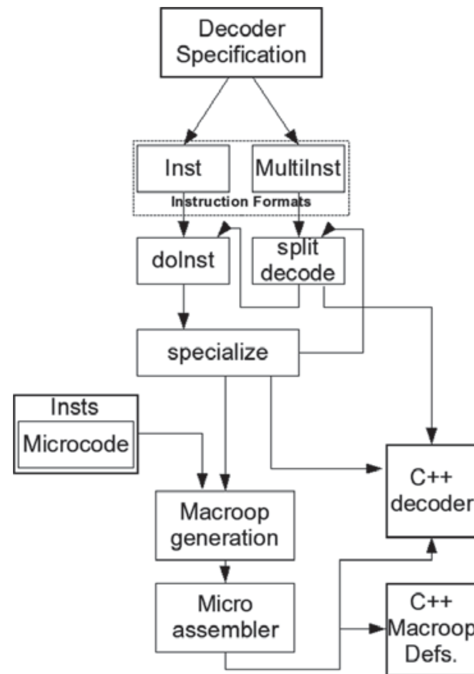
0 0 1 | 0 0 0 0 0 0 1 0

0 1 0 | 0 0 0 0 0 1 0 0
0 1 1 | 0 0 0 0 1 0 0 0
1 0 0 | 0 0 0 1 0 0 0 0
1 0 1 | 0 0 1 0 0 0 0 0
1 1 0 | 0 1 0 0 0 0 0 0
1 1 1 | 1 0 0 0 0 0 0 0

Here is the circuit diagram for the decoder:



Decoder Generation



X86's decoder is generated using an ISA description like all the other ISAs, although how it does that is a bit different. Most of the instructions for most of the other ISAs are defined by passing chunks of code that perform the instruction into an instruction format. The format is basically a template which puts wraps that bit of code in the structure needed to support it and you have your instruction object. Because almost all of the instructions in x86 are microcoded and many can be encoded in multiple ways and hence appear in the decoder more than once, and because the same non-trivial decoding rules apply to many different instructions, X86 uses the decoder as a layer of indirection and defines the majority of its instructions elsewhere.

X86 almost exclusively uses only two different instruction formats, Inst and MultiInst. MultiInst is just a compact way of describing multiple related Insts. An Inst essentially selects

an instruction like XOR and provides a specification for its operands. Inside the instruction format, the instruction name and operand specification are passed to a python function called “specializeInst” which figures out what to do with it. If the operand specification describes more than one version of the instruction, for instance one that uses memory and one that uses registers, the instruction’s information is passed into another function, “doSplitDecode”, which separates out those versions and passes each individually back through the same system. This goes on until the instructions have been fully split out and code has been generated to figure out what version to use. As a nice bonus, the MultiInst format doesn’t add much complexity to this model since it can simply jump right into do Split Decode and continue as normal. There is one additional format for string instructions that works similar to MultiInst, except instead of specializing the instruction based on its operands, it specializes it based on its prefixes. At this point, the code for selecting the right version of an instruction is put into the C++ decoder function. Almost all of this function is built this way, with the minor exception of small bits of logic that glue everything together and make large scale distinctions the number of opcode bytes.

3 to 8 Decoder

The 3 to 8 decoder unit takes 3 address lines as input and outputs 8 address enable lines. Which of the 8 output is enabled is dependant upon the configuration of the 3 address line. If A0, A1, and A2 are all 0, then address 0 is enabled. If A0=0, A1=1, A2=0, then address 3 is enables. With 3 address lines, the number of words that can be addressed is 8 (2^3=8).

Encoders

An encoder is a circuit that changes a set of signals into a code. Lets begin making a 2-to-1 line encoder truth table by reversing the 1-to-2 decoder truth table.

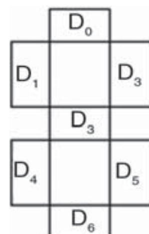
D_1	D_0	A
0	1	0
1	0	1

This truth table is a little short. A complete truth table would be

D_1	D_0	A
0	0	
0	1	0
1	0	1
1	1	

One question we need to answer is what to do with those other inputs? Do we ignore them? Do we have them generate an additional error output? In many circuits this problem is solved by adding sequential logic in order to know not just what input is active but also which order the inputs became active.

A more useful application of combinational encoder design is a binary to 7-segment encoder. The seven segments are given according



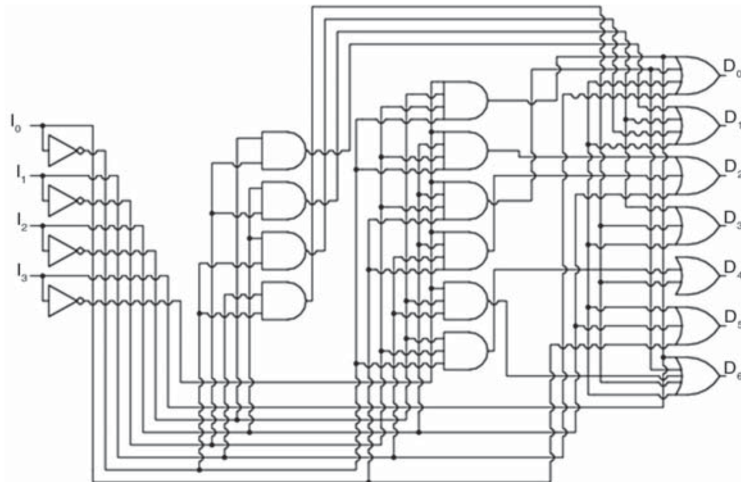
Our truth table is:

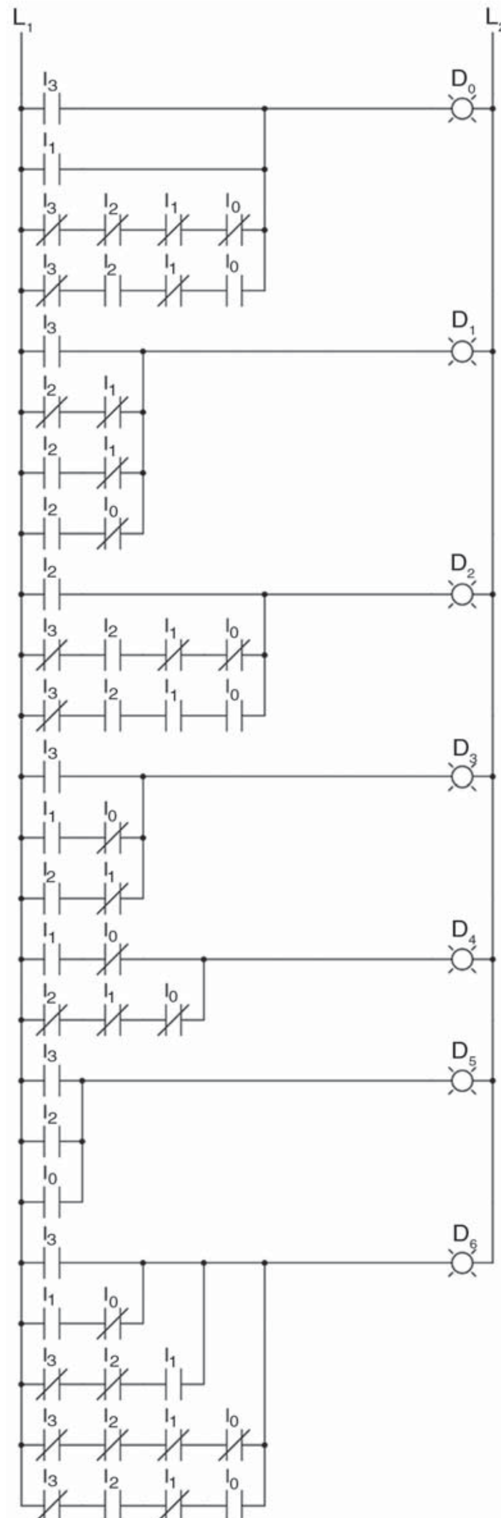
I_3	I_2	I_1	I_0	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0	1
0	0	1	1	1	0	1	1	0	1	1
0	1	0	0	0	1	1	1	0	1	0
0	1	0	1	1	1	0	1	0	1	1
0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	1	0	1	0	0	1	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Deciding what to do with the remaining six entries of the truth table is easier with this circuit. This circuit should not be expected to encode an undefined combination of inputs, so we can leave them as "don't care" when we design the circuit. The boolean equations are;

$$D_0 = I_3 + I_1 + \overline{I_3}\,\overline{I_2}\,\overline{I_1}\,\overline{I_0} + \overline{I_3}I_2\overline{I_1}I_0$$
$$D_1 = I_3 + \overline{I_2}\,\overline{I_1} + I_2\overline{I_1} + I_2\overline{I_0}$$
$$D_2 = I_2 + \overline{I_3}\,\overline{I_2}\,\overline{I_1}I_0 + \overline{I_3}I_2I_1I_0$$
$$D_3 = I_3 + I_1\overline{I_0} + I_2\overline{I_1}$$
$$D_4 = I_1\overline{I_0} + \overline{I_2}\,\overline{I_1}\,\overline{I_0}$$
$$D_5 = I_3 + I_2 + I_0$$
$$D_6 = I_3 + I_1\overline{I_0} + \overline{I_3}\,\overline{I_2}I_1 + \overline{I_3}\,\overline{I_2}\,\overline{I_1}\,\overline{I_0} + \overline{I_3}I_2\overline{I_1}I_0$$

and the circuit is;




L_1 L_2
I_3 D_0
I_1
I_3 I_2 I_1 I_0
I_3 I_2 I_1 I_0
I_3 D_1
I_2 I_1
I_2 I_1
I_2 I_0
I_2 D_2
I_3 I_2 I_1 I_0
I_3 I_2 I_1 I_0
I_3 D_3
I_1 I_0
I_2 I_1
I_1 I_0 D_4
I_2 I_1 I_0
I_3 D_5
I_2
I_0
I_3 D_6
I_1 I_0
I_3 I_2 I_1
I_3 I_2 I_1 I_0
I_3 I_2 I_1 I_0

Introduction

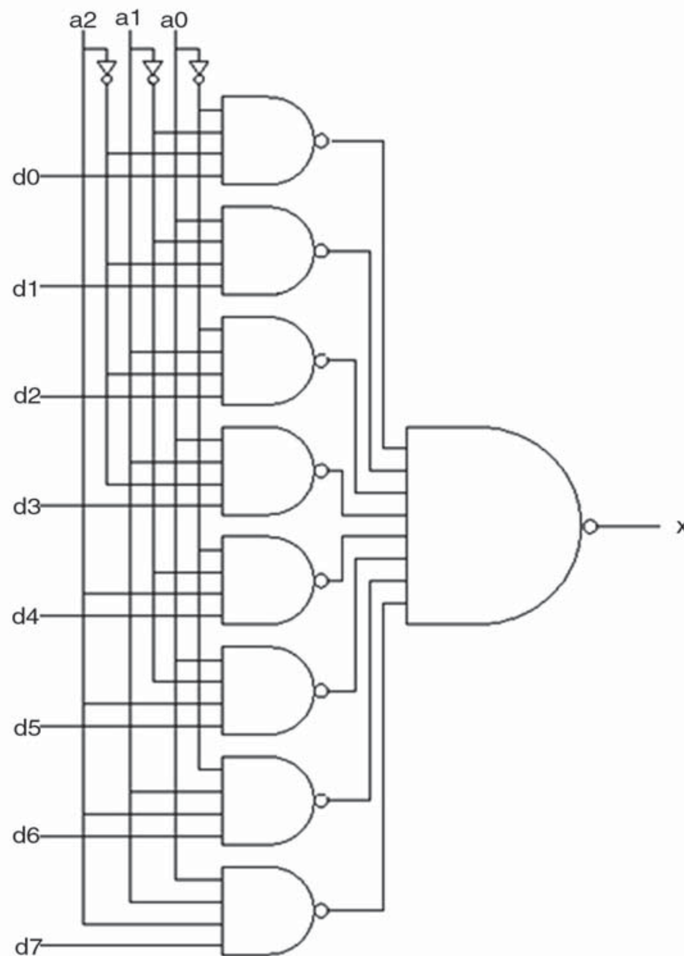
A multiplexer is a combinatorial circuit that is given a certain number (usually a power of two) *data inputs*, let us say 2^n, and n *address inputs* used as a binary number to select one of the data inputs. The multiplexer has a single output, which has the same value as the selected data input. In other words, the multiplexer works like the input selector of a home music system. Only one input is selected at a time, and the selected input is transmitted to the single output. While on the music system, the selection of the input is made manually, the multiplexer chooses its input based on a binary number, the address input. The truth table for a multiplexer is huge for all but the smallest values of n. We therefore use an abbreviated version of the truth table in which some inputs are replaced by '-' to indicate that the input value does not matter. Here is such an abbreviated truth table for n = 3. The full truth table would have $2^{(3 + 23)}$ = 2048rows.

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	|	x
-	-	-	-	-	-	-	-	-	-	-	---	-
0	0	0	-	-	-	-	-	-	-	0	|	0
0	0	0	-	-	-	-	-	-	-	1	|	1
0	0	1	-	-	-	-	-	-	0	-	|	0
0	0	1	-	-	-	-	-	-	1	-	|	1
0	1	0	-	-	-	-	-	0	-	-	|	0
0	1	0	-	-	-	-	-	1	-	-	|	1
0	1	1	-	-	-	-	0	-	-	-	|	0
0	1	1	-	-	-	-	1	-	-	-	|	1
1	0	0	-	-	-	0	-	-	-	-	|	0
1	0	0	-	-	-	1	-	-	-	-	|	1
1	0	1	-	-	0	-	-	-	-	-	|	0
1	0	1	-	-	1	-	-	-	-	-	|	1
1	1	0	-	0	-	-	-	-	-	-	|	0
1	1	0	-	1	-	-	-	-	-	-	|	1
1	1	1	0	-	-	-	-	-	-	-	|	0
1	1	1	1	-	-	-	-	-	-	-	|	1

We can abbreviate this table even more by using a letter to indicate the value of the selected input, like this:

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	|	x
-	-	-	-	-	-	-	-	-	-	-	---	-
0	0	0	-	-	-	-	-	-	-	c	|	c
0	0	1	-	-	-	-	-	-	c	-	|	c
0	1	0	-	-	-	-	-	c	-	-	|	c
0	1	1	-	-	-	-	c	-	-	-	|	c
1	0	0	-	-	-	c	-	-	-	-	|	c
1	0	1	-	-	c	-	-	-	-	-	|	c
1	1	0	-	c	-	-	-	-	-	-	|	c
1	1	1	c	-	-	-	-	-	-	-	|	c

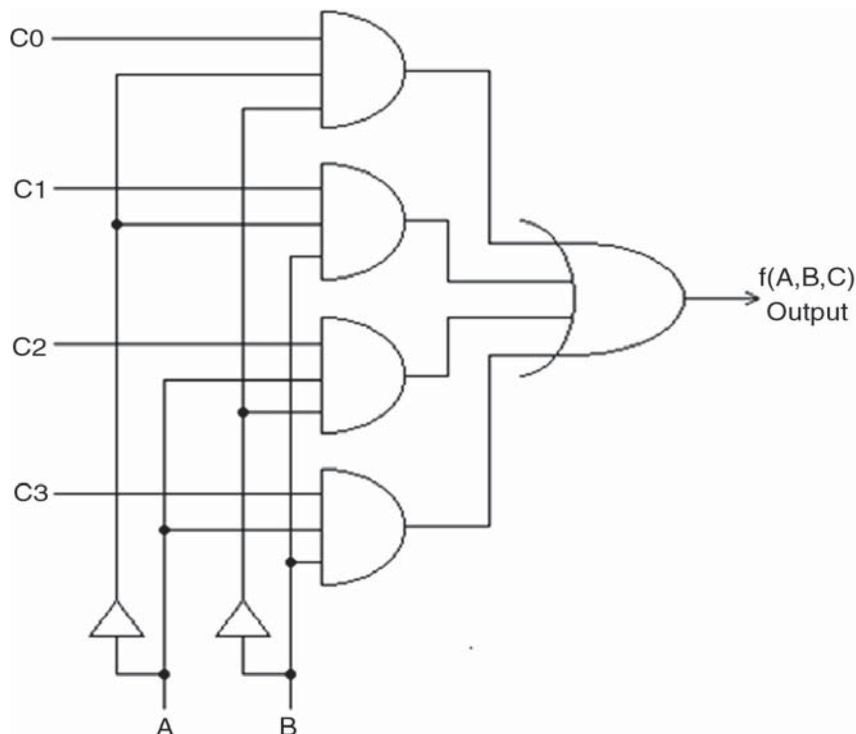
The same way we can simplify the truth table for the multiplexer, we can also simplify the corresponding circuit. Indeed, our simple design method would yield a very large circuit. The simplified circuit looks like this:



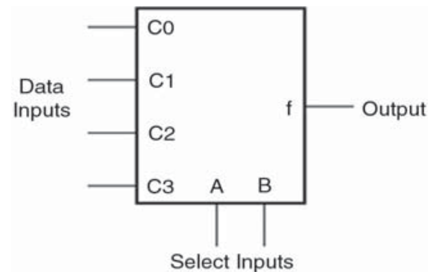
A multiplexer performs the function of selecting the input on any one of 'n' input lines and feeding this input to one output line. Multiplexers are used as one method of reducing the number of integrated circuit packages required by a particular circuit design. This in turn reduces the cost of the system.

Assume that we have four lines, *C*0, *C*1, *C*2 and *C*3, which are to be multiplexed on a single line, *Output (f)*. The four input lines are also known as the *Data Inputs*. Since there are four inputs, we will need two additional inputs to the multiplexer, known as the *Select Inputs*, to select which of the *C* inputs is to appear at the output. Call these select lines *A and B*.

The gate implementation of a 4-line to 1-line multiplexer is shown below:

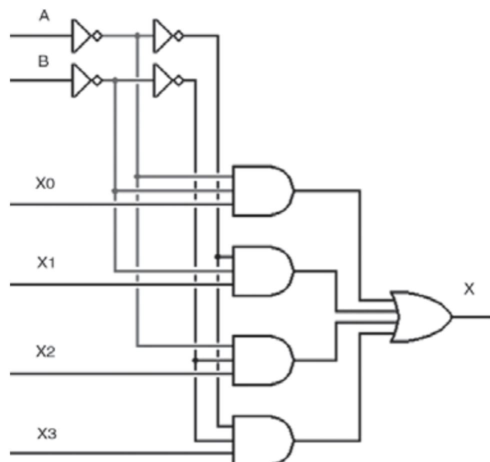


The circuit symbol for the above multiplexer is:



4 Input Multiplexer

The multiplexer concept is not limited to two data inputs. If we add a second addressing input, B, we can control as many as four data inputs, as shown to the left. A third and fourth addressing input will allow the multiplexer to control eight or sixteen inputs, respectively. Inputs A and B are the addressing inputs to this multiplexer. They select which of the four data inputs will be transmitted to the final output, X. If the data inputs are to be multiplexed for transmission to a distant location, the inputs must cycle through all four possible addresses more than twice for each single cycle of each of the data inputs. Otherwise the input data cannot be reconstructed accurately at the receiving end.



6

Conceptual Model

A mental model captures ideas in a problem domain, while a conceptual model represents ‘concepts’ (entities) and relationships between them. A Conceptual model in the field of computer science is also known as a domain model. Conceptual modeling should not be confused with other modeling disciplines such as data modelling, logical modelling and physical modelling. The conceptual model is explicitly chosen to be independent of design or implementation concerns, for example, concurrency or data storage. The aim of a conceptual model is to express the meaning of terms and concepts used by domain experts to discuss the problem, and to find the correct relationships between different concepts. The conceptual model attempts to clarify the meaning of various, usually ambiguous terms, and ensure that problems with different interpretations of the terms and concepts cannot occur. Such differing

interpretations could easily cause confusion amongst stakeholders, especially those responsible for designing and implementing a solution, where the conceptual model provides a key artifact of business understanding and clarity.

Once the domain concepts have been modeled, the model becomes a stable basis for subsequent development of applications in the domain. The concepts of the conceptual model can be mapped into physical design or implementation constructs using either manual or automated code generation approaches. The realization of conceptual models of many domains can be combined to a coherent platform. A conceptual model can be described using various notations, such as UML or OMT for object modelling, or IE or IDEF1X for Entity Relationship Modelling. In UML notation, the conceptual model is often described with a class diagram in which classes represent concepts, associations represent relationships between concepts and role types of an association represent role types taken by instances of the modelled concepts in various situations. In ER notation, the conceptual model is described with an ER Diagram in which entities represent concepts, cardinality and optionality represent relationships between concepts. Regardless of the notation used, it is important not to compromise the richness and clarity of the business meaning depicted in the conceptual model by expressing it directly in a form influenced by design or implementation concerns.

REQUIREMENTS ANALYSIS

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into

determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users. Requirements analysis is critical to the success of a development project. Requirements must be documented, actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Requirements can be architectural, structural, behavioural, functional, and non-functional.

Overview

Conceptually, requirements analysis includes three types of activity:

- Eliciting requirements: the task of communicating with customers and users to determine what their requirements are. This is sometimes also called requirements gathering.
- Analyzing requirements: determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues.
- Recording requirements: Requirements might be documented in various forms, such as natural-language documents, use cases, user stories, or process specifications.

Requirements analysis can be a long and arduous process during which many delicate psychological skills are involved. New systems change the environment and relationships between people, so it is important to identify all the

stakeholders, take into account all their needs and ensure they understand the implications of the new systems. Analysts can employ several techniques to elicit the requirements from the customer. Historically, this has included such things as holding interviews, or holding focus groups (more aptly named in this context as requirements workshops) and creating requirements lists. More modern techniques include prototyping, and use cases. Where necessary, the analyst will employ a combination of these methods to establish the exact requirements of the stakeholders, so that a system that meets the business needs is produced.

Requirements Engineering

Systematic requirements analysis is also known as *requirements engineering*. It is sometimes referred to loosely by names such as *requirements gathering*, *requirements capture*, or requirements specification. The term *requirements analysis* can also be applied specifically to the analysis proper, as opposed to elicitation or documentation of the requirements, for instance. Requirements Engineering can be divided into discrete chronological steps:

- Requirements elicitation,
- Requirements analysis and negotiation,
- Requirements specification,
- System modeling,
- Requirements validation,
- Requirements management.

Requirement engineering according to Laplante (2007) is "a subdiscipline of systems engineering and software

engineering that is concerned with determining the goals, functions, and constraints of hardware and software systems." In some life cycle models, the requirement engineering process begins with a feasibility study activity, which leads to a feasibility report. If the feasibility study suggests that the product should be developed, then requirement analysis can begin. If requirement analysis precedes feasibility studies, which may foster outside the box thinking, then feasibility should be determined before requirements are finalized.

Requirements Analysis Topics

See Stakeholder analysis for a discussion of business uses. Stakeholders (SH) are people or organizations (legal entities such as companies, standards bodies) which have a valid interest in the system. They may be affected by it either directly or indirectly. A major new emphasis in the 1990s was a focus on the identification of *stakeholders*. It is increasingly recognized that stakeholders are not limited to the organization employing the analyst. Other stakeholders will include:

- anyone who operates the system (normal and maintenance operators)
- anyone who benefits from the system (functional, political, financial and social beneficiaries)
- anyone involved in purchasing or procuring the system. In a mass-market product organization, product management, marketing and sometimes sales act as surrogate consumers (mass-market customers) to guide development of the product

- organizations which regulate aspects of the system (financial, safety, and other regulators)
- people or organizations opposed to the system (negative stakeholders; see also Misuse case)
- organizations responsible for systems which interface with the system under design
- those organizations who integrate horizontally with the organization for whom the analyst is designing the system

Stakeholder Interviews

Stakeholder interviews are a common technique used in requirement analysis. Though they are generally idiosyncratic in nature and focused upon the perspectives and perceived needs of the stakeholder, very often without larger enterprize or system context, this perspective deficiency has the general advantage of obtaining a much richer understanding of the stakeholder's unique business processes, decision-relevant business rules, and perceived needs.

Consequently this technique can serve as a means of obtaining the highly focused knowledge that is often not elicited in Joint Requirements Development sessions, where the stakeholder's attention is compelled to assume a more cross-functional context. Moreover, the in-person nature of the interviews provides a more relaxed environment where lines of thought may be explored at length.

Joint Requirements Development (JRD) Sessions

Requirements often have cross-functional implications that are unknown to individual stakeholders and often missed or incompletely defined during stakeholder interviews.

These cross-functional implications can be elicited by conducting JRD sessions in a controlled environment, facilitated by a trained facilitator, wherein stakeholders participate in discussions to elicit requirements, analyze their details and uncover cross-functional implications. A dedicated scribe and Business Analyst should be present to document the discussion. Utilizing the skills of a trained facilitator to guide the discussion frees the Business Analyst to focus on the requirements definition process. JRD Sessions are analogous to Joint Application Design Sessions. In the former, the sessions elicit requirements that guide design, whereas the latter elicit the specific design features to be implemented in satisfaction of elicited requirements.

Contract-Style Requirement Lists

One traditional way of documenting requirements has been contract style requirement lists. In a complex system such requirements lists can run to hundreds of pages. An appropriate metaphor would be an extremely long shopping list. Such lists are very much out of favour in modern analysis; as they have proved spectacularly unsuccessful at achieving their aims; but they are still seen to this day.

Strengths

- Provides a checklist of requirements.
- Provide a contract between the project sponsor(s) and developers.
- For a large system can provide a high level description.

Weaknesses

- Such lists can run to hundreds of pages. It is virtually

impossible to read such documents as a whole and have a coherent understanding of the system.

- Such requirements lists abstract all the requirements and so there is little context
- This abstraction makes it impossible to see how the requirements fit or work together.
- This abstraction makes it difficult to prioritize requirements properly; while a list does make it easy to prioritize each individual item, removing one item out of context can render an entire use case or business requirement useless.
- This abstraction increases the likelihood of misinterpreting the requirements; as more people read them, the number of (different) interpretations of the envisioned system increase.
- This abstraction means that it's extremely difficult to be sure that you have the majority of the requirements. Necessarily, these documents speak in generality; but the devil, as they say, is in the details.
- These lists create a false sense of mutual understanding between the stakeholders and developers.
- These contract style lists give the stakeholders a false sense of security that the developers must achieve certain things. However, due to the nature of these lists, they inevitably miss out crucial requirements which are identified later in the process. Developers can use these discovered requirements to renegotiate the terms and conditions in their favour.

- These requirements lists are no help in system design, since they do not lend themselves to application.

Alternative to Requirement Lists

As an alternative to the large, pre-defined requirement lists Agile Software Development uses User stories to define a requirement in every day language.

Measurable Goals

Best practices take the composed list of requirements merely as clues and repeatedly ask "why?" until the actual business purposes are discovered. Stakeholders and developers can then devise tests to measure what level of each goal has been achieved thus far. Such goals change more slowly than the long list of specific but unmeasured requirements. Once a small set of critical, measured goals has been established, rapid prototyping and short iterative development phases may proceed to deliver actual stakeholder value long before the project is half over.

Prototypes

In the mid-1980s, prototyping was seen as the best solution to the requirements analysis problem. Prototypes are Mockups of an application. Mockups allow users to visualize an application that hasn't yet been constructed. Prototypes help users get an idea of what the system will look like, and make it easier for users to make design decisions without waiting for the system to be built. Major improvements in communication between users and developers were often seen with the introduction of prototypes. Early views of applications led to fewer changes later and hence reduced overall costs considerably. However,

over the next decade, while proving a useful technique, prototyping did not solve the requirements problem:

- Managers, once they see a prototype, may have a hard time understanding that the finished design will not be produced for some time.
- Designers often feel compelled to use patched together prototype code in the real system, because they are afraid to 'waste time' starting again.
- Prototypes principally help with design decisions and user interface design. However, they can not tell you what the requirements originally were.
- Designers and end-users can focus too much on user interface design and too little on producing a system that serves the business process.
- Prototypes work well for user interfaces, screen layout and screen flow but are not so useful for batch or asynchronous processes which may involve complex database updates and/or calculations.

Prototypes can be flat diagrams (often referred to as wireframes) or working applications using synthesized functionality. Wireframes are made in a variety of graphic design documents, and often remove all color from the design (i.e. use a greyscale color palette) in instances where the final software is expected to have graphic design applied to it. This helps to prevent confusion over the final visual look and feel of the application.

Use Cases

A use case is a technique for documenting the potential requirements of a new system or software change. Each use

case provides one or more *scenarios* that convey how the system should interact with the end-user or another system to achieve a specific business goal. Use cases typically avoid technical jargon, preferring instead the language of the end-user or *domain expert*. Use cases are often co-authored by requirements engineers and stakeholders. Use cases are deceptively simple tools for describing the behaviour of software or systems. A use case contains a textual description of all of the ways which the intended users could work with the software or system. Use cases do not describe any internal workings of the system, nor do they explain how that system will be implemented. They simply show the steps that a user follows to perform a task. All the ways that users interact with a system can be described in this manner.

Software Requirements Specification

A software requirements specification (SRS) is a complete description of the behaviour of the system to be developed. It includes a set of use cases that describe all of the interactions that the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains nonfunctional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints). Recommended approaches for the specification of software requirements are described by IEEE 830-1998. This standard describes possible structures, desirable contents, and qualities of a software requirements specification.

Types of Requirements

Requirements are categorized in several ways. The following are common categorizations of requirements that relate to technical management:Customer Requirements Statements of fact and assumptions that define the expectations of the system in terms of mission objectives, environment, constraints, and measures of effectiveness and suitability (MOE/MOS). The customers are those that perform the eight primary functions of systems engineering, with special emphasis on the operator as the key customer. Operational requirements will define the basic need and, at a minimum, answer the questions posed in the following listing:

- *Operational distribution or deployment*: Where will the system be used?
- *Mission profile or scenario*: How will the system accomplish its mission objective?
- *Performance and related parameters*: What are the critical system parameters to accomplish the mission?
- *Utilization environments*: How are the various system components to be used?
- *Effectiveness requirements*: How effective or efficient must the system be in performing its mission?
- *Operational life cycle*: How long will the system be in use by the user?
- *Environment*: What environments will the system be expected to operate in an effective manner?

Architectural Requirements

Architectural requirements explain what has to be done by identifying the necessary system architecture of a system.

Structural Requirements

Structural requirements explain what has to be done by identifying the necessary structure of a system.

Behavioural Requirements

Behavioural requirements explain what has to be done by identifying the necessary behaviour of a system.

Functional Requirements

Functional requirements explain what has to be done by identifying the necessary task, action or activity that must be accomplished. Functional requirements analysis will be used as the toplevel functions for functional analysis.

Non-functional Requirements

Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviours.

Performance Requirements

The extent to which a mission or function must be executed; generally measured in terms of quantity, quality, coverage, timeliness or readiness. During requirements analysis, performance (how well does it have to be done) requirements will be interactively developed across all identified functions based on system life cycle factors; and characterized in terms of the degree of certainty in their estimate, the degree of criticality to system success, and their relationship to other requirements.

Design Requirements

The “build to,” “code to,” and “buy to” requirements for

products and "how to execute" requirements for processes expressed in technical data packages and technical manuals.

Derived Requirements

Requirements that are implied or transformed from higher-level requirement. For example, a requirement for long range or high speed may result in a design requirement for low weight.

Allocated Requirements

A requirement that is established by dividing or otherwise allocating a high-level requirement into multiple lower-level requirements. Example: A 100-pound item that consists of two subsystems might result in weight requirements of 70 pounds and 30 pounds for the two lower-level items. Well-known requirements categorization models include FURPS and FURPS+, developed at Hewlett-Packard.

Requirements Analysis Issues

Stakeholder Issues

Steve McConnell, in his book *Rapid Development*, details a number of ways users can inhibit requirements gathering:

- Users do not understand what they want or users don't have a clear idea of their requirements
- Users will not commit to a set of written requirements
- Users insist on new requirements after the cost and schedule have been fixed
- Communication with users is slow
- Users often do not participate in reviews or are incapable of doing so
- Users are technically unsophisticated

- Users do not understand the development process
- Users do not know about present technology

This may lead to the situation where user requirements keep changing even when system or product development has been started.

Engineer/Developer Issues

Possible problems caused by engineers and developers during requirements analysis are:

- Technical personnel and end-users may have different vocabularies. Consequently, they may wrongly believe they are in perfect agreement until the finished product is supplied.
- Engineers and developers may try to make the requirements fit an existing system or model, rather than develop a system specific to the needs of the client.
- Analysis may often be carried out by engineers or programmers, rather than personnel with the people skills and the domain knowledge to understand a client's needs properly.

Attempted Solutions

One attempted solution to communications problems has been to employ specialists in business or system analysis. Techniques introduced in the 1990s like prototyping, Unified Modeling Language (UML), use cases, and Agile software development are also intended as solutions to problems encountered with previous methods. Also, a new class of application simulation or application definition tools have

entered the market. These tools are designed to bridge the communication gap between business users and the IT organization — and also to allow applications to be 'test marketed' before any code is produced. The best of these tools offer:

- electronic whiteboards to sketch application flows and test alternatives
- ability to capture business logic and data needs
- ability to generate high fidelity prototypes that closely imitate the final application
- interactivity
- capability to add contextual requirements and other comments
- ability for remote and distributed users to run and interact with the simulation

ARCHITECTURE DESCRIPTION LANGUAGE

Different communities use the term architecture description language. Some important communities are the system engineering community, the software engineering community and the enterprize modelling and engineering community In the system engineering community, an Architecture Description Language (ADL) is a language and/or conceptual model used to describe and represent system architectures. In the software engineering community, an Architecture Description Language (ADL) is a computer language used to describe and represent software architectures. This means in case of technical architecture, the architecture must be communicated to software developers. With functional architecture, the software

architecture is communicated with stakeholders and enterprize engineers. By the software engineering community several ADLs have been developed, such as Acme (developed by CMU), AADL (standardized by SAE), C2 (developed by UCI), Darwin (developed by Imperial College London), and Wright (developed by CMU). The Final Committee Draft of ISO/IEC 42010, now titled 'Systems and software engineering — Architecture Description', defines an Architecture Description Language:

> form of expression used for the description of architectures

The enterprize modelling and engineering community have also developed architecture description languages catered for at the enterprize level. Examples include ArchiMate (now an Open Group standard), DEMO, ABACUS (developed by the University of Technology, Sydney) etc. These languages do not necessarily refer to software components, etc. Most of them, however, refer to an application architecture as the architecture that is communicated to the software engineers. Most of the writing below refers primarily to the perspective from the software engineering community.

Introduction

A standard notation (ADL) for representing architectures helps promote mutual communication, the embodiment of early design decisions, and the creation of a transferable abstraction of a system. Architectures in the past were largely represented by box-and-line drawing annotated with such things as the nature of the component, properties, semantics of connections, and overall system behaviour.

ADLs result from a linguistic approach to the formal representation of architectures, and as such they address its shortcomings. Also important, sophisticated ADLs allow for early analysis and feasibility testing of architectural design decisions.

Characteristics

There is a large variety in ADLs developed by either academic or industrial groups. Many languages were not intended to be an ADL, but they turn out to be suitable for representing and analyzing an architecture. In principle ADLs differ from requirements languages, because ADLs are rooted in the solution space, whereas requirements describe problem spaces. They differ from programming languages, because ADLs do not bind architectural abstractions to specific point solutions. Modeling languages represent behaviours, where ADLs focus on representation of components. However, there are domain specific modeling languages (DSMLs) that focus on representation of components.

Minimal Requirements

The language must:

- Be suitable for communicating an architecture to all interested parties
- Support the tasks of architecture creation, refinement and validation
- Provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL

- Provide the ability to represent most of the common architectural styles
- Support analytical capabilities or provide quick generating prototype implementations

ADLs have in common:

- Graphical syntax with often a textual form and a formally defined syntax and semantics
- Features for modeling distributed systems
- Little support for capturing design information, except through general purpose annotation mechanisms
- Ability to represent hierarchical levels of detail including the creation of substructures by instantiating templates

ADLs differ in their ability to:

- Handle real-time constructs, such as deadlines and task priorities, at the architectural level
- Support the specification of different architectural styles. Few handle object oriented class inheritance or dynamic architectures
- Support analysis
- Handle different instantiations of the same architecture, in relation to product line architectures

Positive Elements of ADL

- ADLs represent a formal way of representing architecture
- ADLs are intended to be both human and machine readable

- ADLs support describing a system at a higher level than previously possible
- ADLs permit analysis of architectures – completeness, consistency, ambiguity, and performance
- ADLs can support automatic generation of software systems

Negative Elements of ADL

- There is not universal agreement on what ADLs should represent, particularly as regards the behaviour of the architecture
- Representations currently in use are relatively difficult to parse and are not supported by commercial tools
- Most ADLs tend to be very vertically optimized toward a particular kind of analysis

Common Concepts of Architecture

The ADL community generally agrees that Software Architecture is a set of components and the connections among them. But there are different kind of architectures like :

Object Connection Architecture

- Configuration consists of the interfaces and connections of an object-oriented system
- Interfaces specify the features that must be provided by modules conforming to an interface
- Connections represented by interfaces together with call graph
- Conformance usually enforced by the programming language

  - o Decomposition - associating interfaces with unique modules
  - o Interface conformance - static checking of syntactic rules
  - o Communication integrity - visibility between modules

Interface Connection Architecture

- Expands the role of interfaces and connections
  - o Interfaces specify both "required" and "provided" features
  - o Connections are defined between "required" features and "provided" features
- Consists of interfaces, connections and constraints
  - o Constraints restrict behaviour of interfaces and connections in an architecture
  - o Constraints in an architecture map to requirements for a system

Most ADLs implement an interface connection architecture.

Architecture vs. Design

So what is the difference between architecture and design? Architecture casts non-functional decisions and partitions functional requirements, whereas design specifies or derives functional requirements. The process of defining an architecture may use heuristics or iterative improvements; this may require going a level deeper to validate the choices, so the architect often has to do a high-level design to validate the partitioning.

DATA MODELS

Introduction

- A data model is mathematical formalism consisting of two parts.
  - A notation for describing data,
  - A set of operations used to manipulate that data.
- A data model is a way of organizing a collection of facts pertaining to a system under investigation.
- Data models provide a way of thinking about the world, a way of organizing the phenomena that interest us.
- They can be thought of as an abstract language, a collection of words along with a grammar by which we describe our subject.
  - By choosing a language, we pay the price of being constrained to form expressions whose words are limited to those in the language and whose sentence structure is governed by the languages grammar.
  - We are not free to use random collections of symbols for words nor can we put the words together in any ad hoc fashion.
- A major benefit we receive by following a data model stems from the theoretical foundation of the model.
  - From the theory emerges the power of analysis, the ability to extract inferences and to create deductions that emerge from the raw data.
- Different models provide different conceptualizations of the world; they have different outlooks and different perspectives.

- DBMSs are seen to be composed of three levels of abstraction:
  - Physical: This is the implementation of the database in a digital computer. It is concerned with things like storage structures and access method data structures.
  - Conceptual: This is the expression of the database designers model of the real world in the language of the data model.
  - View: Different user groups can be given access to different portions of the database. A user groups portion of the database is called their view.

TYPES OF DATA MODELS

Common Data Models

- This will presents an overview of most common data models:

Entity-Relationship Model

- The Entity-Relationship (ER) model is generally attributed to (Chen 1976).
- The ER model envisions the world as comprised of entities that are associated with each other by relationships. All of the entities of a particular type are collected together into entity sets.
- Entity sets and relationships can be depicted graphically in an ER-diagram.

Entities

- Entities are distinguishable real-world objects such as employees, maps, airplanes, or bus schedules.

  - Distinguishable means that all entities can be uniquely identified.
  - Entities have common attributes that define what it means to be such an entity.
  - Any particular real-world object does not necessarily have a single or best representation as an entity.
- For any given real-world object, different modelers can choose different sets of attributes of the object that are of interest to their particular situation.
- This results in the same object being modeled differently.
- Entities are collected into entity sets.
  - Entity sets are depicted as rectangles in ER diagrams.
  - Their attributes are depicted as ellipses attached to the rectangles by lines.

Relationships

- A relationship is a list of entity sets.
  - Notation: two entity sets A and B that stand in relationship r is written A r B. See the next bullet for examples.
- Types of relationships:
  - Aggregating relationships:
    - One-one: if A r B and r is one-one then each entity of B is in relationship with at most one entity of A and vice-versa.
    - For example, if CAPTAIN commands VESSEL and commands is one-one then, in our model,

each vessel has at most one captain and each captain commands at most one vessel at a time.

- Many-one: if A r B and r is many-one then each entity of A is in relationship with at most one entity of B but not vice-versa.
- For example, if CREW assigned-to VESSEL and assigned-to is many-one then, in our model, a vessel has many crew members but a crew member is assigned to only one vessel.
- Many-many: if A r B and r is many-many then each entity of A can be in relationship with any number of B entities and vice-versa.
- For example, if VESSEL patrols REGION and patrols is many-many then, in our model, a vessel patrols many regions and a region is patrolled by many ships.
- Isa (read is a) relationships: if A is a B then A is a specialization of B, or, conversely, B is a generalization of A.
- For example, if CAPTAIN is a CREW then, in our model, captains have all the attributes of crew members but not vice versa.
- The is a relationship allows hierarchies to be established among entity sets.

* A Relationship is depicted by a lozenge with lines connecting it to the relevant entity sets.
* The Entity-Relationship model lacks an underlying formalism and is, therefore, used more for general conceptualization than for creating physical models

  - (Indeed, some authors do not acknowledge the ER model as a data model at all).
  - It is not uncommon for a conceptual design to be expressed in the ER model and then translated into another model for implementation.

Network Model

- The network data model is based upon the concept of a structure such as is found in programming languages like C or Pascal.
  - ER entities can be modeled as structures with the entitys attributes corresponding to the structures fields.
  - Entities are distinguished by their location, *i.e.*, the physical address of the structure that is holding them. Thus, two structures of identical value represent two separate entities.
- Entity sets can be implemented as files whose records match the structures.
- Relationships are created with explicit linkages (*viz.* pointers) from structure to structure.
- Codasyl is an example of a DBMS based on the network model.
- The network model has neither formal semantics nor a high-level query language. Database manipulation was done via custom programmes often written in COBOL.
- Network model databases are hand-coded and, therefore, can be very efficient in their space utilization and query execution times; all the

relationships are hardwired or pre computed and built into the structure of the database itself.

- The price for such performance is inflexibility and great difficulty of use (among many other things).

Relational Model

- The relational model was introduced by Codd and has been the inspiration of an entire generation of database management systems that are based on the concept of a relation which is a set of tuples.

Tuples

- A tuple is a set of facts that are related to each other in some way (perhaps only by the fact theyve been put together in a set).
- Each fact in a tuple is a datum whose value comes from a specified domain (*e.g.,* the domain of all integers, the domain of all character strings of length 255 or less, *etc.*)
- Formally, let D1,..., Dn be n sets of values constituting n domains (n is usually greater than zero but that is not strictly necessary). A tuple t is a set of values t = {d1,..., dn}, such that d1 is an element of D1,..., and dn is an element of Dn. The domains are called attributes.

Relations

- Formally, let D1,..., Dn be n domains. A relation R is a set of tuples over the Cartesian product D1 x... x Dn.
- In English, a relation is a (possibly complete) subset

of all the possible tuples formed by the Cartesian product of the domains.

- Since tuples are sets (of values) and a relation is also a set (of tuples), relations are sets of sets.
  - A file is a list of records
  - A table is a list of rows
  - A relation is a set of tuples.
- Relations are naturally represented as tables.
  - Tables are not relations because relations cannot have duplicate tuples and there is no such stricture on tables. However, it is perhaps convenient to think about relations as tables so long as the distinction remains clear.
  - Most (if not all) commercial relational DBMSs violate this principle: they allow duplicate tuples.
- The use of relations as a data modeling tool becomes apparent when we have a relation, say, OUR_DEM with fields {quadname, zone_code, mappingcenter}.
  - It happens that the USGS has a digital elevation model named Placitas NM in UTM zone 13 that was created by the Forest Service Mapping Center.
  - Then, the presence of a tuple in the OUR_DEM relation whose
    - Quadname attribute has the value Placitas NM and
    - Zone_code attribute has the value 13, and
    - Mappingcenter attribute has the value FS,
    - Indicates that we have the Placitas DEM in our possession.

Tuples, Relations and Keys

- Relations are sets of tuples: Consequently, no two tuples that are elements of the same relation can have identical values for all their attributes. That is to say, there are no duplicate tuples in a relation.
- All tuples in a relation can be distinguished by the values of their attributes.
  - Any set of attributes whose values necessarily uniquely identify a tuple are said to be a key.
- Database designers choose some attribute set to be a key for their databases relations.
  - This key is known as the primary key.
- If the primary key of one table appears as an attribute of a different relation, the key is known as a foreign key in the other relation.
- A key uniquely identifies its tuple. Therefore, a tuples key is often used as a surrogate for the entire tuple.

Relationships

- Not surprisingly, the relational model represents relationships with relations.
- Key attributes are denoted in bold face.
- If you wish to work with these examples, you can download either:
  - The Microsoft Access97.mdb file by SAVING the files at:
  - ASCII text for the tables by clicking their names below.
    - The attribute names are on the first row, character strings are delimited with double quotes (“) and the fields are comma delimited.

- Aggregating relationships are represented by embedding the primary key of one relation into another relation as a foreign key:
  - One-one: if A r B and r is one-one then the primary key of A can be embedded in B or vice versa or both.
    - For example, suppose CAPTAIN commands VESSEL and that commands is one-one.
    - Suppose further that cptn_name is the primary key of CAPTAIN and vessel_name is the primary key of VESSEL.
    - Then, CAPTAIN could have an attribute commands whose value is that of vessel_name for the vessel that captain commands.
    - It is equally reasonable to have an attribute commanded by in VESSEL whose value is that of name for the captain commanding the vessel.
    - Many-one: if A r B and r is many-one then the primary key of B can be embedded in A but not vice versa.
  - For example, suppose CREW assigned-to VESSEL and assigned-to is many-one.
  - Suppose further that crew_name is the primary key of CREW and vessel_name is the primary key of VESSEL.
  - Then, CREW could have an attribute assigned_to whose value is that of vessel_name for the vessel this crew member serves on.
  - However, VESSEL cannot have an attribute roster because roster would have to be a set (many crew

members per vessel) and the relational model stipulates that all domains are atomic; no collections.

  - Many-many: if A r B and r is many-many then neither primary key can be embedded the other table. Again, the difficult lies in the atomicity rule for domains. So, for a many-many relationship, we must create a separate relation whose attributes include but are not limited to the primary keys from A and B.

- For example, if VESSEL patrols REGION and patrols is many-many.
- Suppose further that vessel_name is the primary key of VESSEL and region_name is the primary key of REGION. Then we have a third relation PATROLS with attributes vessel_name and region_name.
- such relations are sometimes called join supports
- such relations are no different in any way from any other relation is a relationships are handled as the other relationships:
  - One-one: Suppose CAPTAIN isa CREW.
- Then there is a one-one relationship between CAPTAIN and CREW so the primary key of CREW can be used as the key in CAPTAIN.
- The one-one nature of this relationship indicates that the two tuples really give details of the same entity; they are sort of like a single tuple that has been split in two.
  - Many-one: Suppose we are modeling WWII

combat vessels, known collectively as "ship(s) of the line" (SOTL). It happens that a ship design can be used as the plan for many individual vessels (obviously).

- The design is known as a class and the vessels made to that design are said to belong to that class.
- For example, the USS Missouri belongs to the Iowa class of battleships.
- We model this relationship with a relation SOTL which has a single tuple for each class of warship. Thus, VESSEL isaSOTL.
- The SOTL relation has attributes that are common to all ships of the line. For WWII vessels, this might include attributes such as the number of primary guns, size of the primary guns, *etc.*
- The tuple in SOTL for the Iowa battleships gives information that is common to all Iowa class battleships (*e.g.*, nine 16-inch guns, *etc.*).
- The tuple in VESSEL for the USS Missouri holds the information specific to that vessel including the fact it belongs in the Iowa class.
- Therefore, the primary key of SOTL is embedded in VESSEL, not vice versa.
- Compare many-one is a relationships with one-one is a relationships.
- Ullman restricts relationships to be one-one.

Query Languages

- Codd invented two early languages for dealing with relations: one was algebraic and the other was based

on first-order predicate logic. These languages have the same expressive power.

- Relational Algebra
- [An] algebraic notation where queries are expressed by applying specialized operators to relations
- See for a presentation of the relational algebra.
  - Relational Calculus
- [A] logical notation where queries are expressed by writing logical formulas that the tuples in the answer must satisfy.
- See for a presentation of the relational calculus.

• The most common commercial query language is the Structured Query Language, or SQL.

- Despite its reputation as a relational query language, SQL does not fully support the relational model (it includes things that are not in the model and omits things that are.

3.6. Relational Database Management System (RDBMS)

- A relational database management system is a DBMS based on the relational model as defined.
- There is no commercially available DBMS that fully implements the relational model as defined. Some are coming closer. Not everyone agrees that this strict lack of conformance is a Bad Thing.

Advantages of the Relational Model

- Codd presents many advantages of the relational model. Some of them are highlighted below:
- The relational model is truly a mathematically

complete data model. This solid theoretical underpinning is responsible for

- Ad hoc query languages whose queries can be automatically compiled, executed, and optimized without resorting to programming
- Correctness: The semantics of the relational algebra are sound and complete
- Predictable: the consistent semantics enables users to easily anticipate the result of a given query

- Adaptability: Making a change in the structure of the tables in the network model requires programmatic making changes to all the databases queries. As a result, the network model is inflexible in the extreme.
  - The relational model cleanly separates the logical from the physical model and this decoupling mitigates or eliminates these problems.
  - Also, the relational models integrity constraints are very helpful in ensuring that structural changes did not adversely effect the meaning of the database.
- Multiple views: it is straightforward to present different user groups different views of the same database.
- Concurrency: A full theory of transaction concurrency control exists which depends upon the theoretical formalisms of the relational model.
  - This theory guarantees the correct execution of concurrent queries (indeed, it defines what is correct).

Object Model

- The word object is similar to the Entity-Relationship concept of an entity although object is more general.
  - I recommend taking object in the spirit of objects in the physical world.
  - Objects are things but they are not limited to physical, tangible things. For example, data structures (*e.g.*, a hash table) can be objects.
  - All objects are distinct and, like the network model, are made distinct by an identifying attribute, the object ID.
- Like the other models, the object model assumes that objects can conceptually be collected together into meaningful groups. These groups are called classes.
- An object grouping is meaningful because objects of the same class must have common attributes, behaviours, and relationships with other objects.
- Unlike entity sets and relations, classes do not actually hold the objects of that class.
  - Classes are purely conceptual.
  - There is nothing in the object model that is equivalent to either a entity set or a relation (there could be but its not required by the model).
- Like the network model, the relationships among objects are specified via a physical link (pointer) between objects.
- According to Rumbaugh *et al.* (1991), The object model describes the structure of objects in a system their identity, their relationships to other objects, their attributes, and their operations.

- The DARPA Open OODB project proposes the following as the essential features of the OO data model:
  - Object identity: The ability of the system to distinguish between two different objects that have the same state. The state of an object can be shared by several objects via object identity.
  - Encapsulation: a kind of abstraction that enforces a clean separation between the external interface (behaviour) of an object and its internal implementation. Encapsulation requires that all access (or interaction) with objects be done by invoking the services provided by their external interface.
  - Complex state: The ability to define data types whose implementation has a nested structure. The state of an object could be built from records of primitive types, other objects, or [collections] of objects.
  - Type extensibility: The ability to define new data types from previously defined types by enhancing or changing the structure or behaviour of the types. Type inheritance is a mechanism used to define new types by enhancing already existing behaviour.
  - Genericity: The types of the object data model with which the object query language collaborates must be generic. That is, as a new type is added to the system, it must be queriable.
- There is no universally agreed upon object data model but The Object-Oriented Database System Manifesto

(Atkinson, *et al.* 1989) gives a framework being considered from which to derive a standard.

- According to Rao, The object-oriented database (OODB) paradigm is the combination of object-oriented programming language (OOPL) systems and persistent systems.

  The power of the OODB comes from the seamless treatment of both persistent data, as found in databases, and transient data, as found in executing programmes.
  - Note that the emphasis with OODB, like the network model, is towards programmers, not end users.
  - This point is further emphasized by the primary interface to OODBs being OOPLs.
- I suggest for a good introduction to object-oriented design and analysis.

Inheritance (isa) Relationships and Typing

- Many object-oriented models take classes to be a typing mechanism (for example, Eiffel and C++.
- The type of an object is its class; an object is an instance of its class.
  - For example, the number 2.3 is an instance of the class of rational numbers.
- Interpreting classes to be types implies the inherent ability of users to create their own data domains.
- Inheritance can be viewed from two perspectives:
  - Incremental: the process of adding attributes and functions to an existing class (the base class).

- New attributes/functions can added to the New class that were not in the base class.
- This is a technique for code reuse.
- No typing information is implied by this relationship.
- For example, suppose that there is a class PERSISTENT that has the functionality of automatically storing its objects in a database. Any class that inherits PERSISTENT magically gains the ability to do likewise.
  - Subtyping: a technique for arranging class definitions in a hierarchy satisfying the condition that members of the subclass are also members of the superclass.
- Subtyping constitutes the isa relationship.
- Old attributes/functions can change type so long as the new type is more specific (it inherited either directly or indirectly) than the original base class.
- Old attributes/functions cannot be removed.
- Old functions can be provided with new implementations so long as the interface to the function remains unchanged (or is changed via specialization as indicated above)

• Various object models span the gambit of inheritance relationships:
  - Full repeated multiple inheritance.
  - Single inheritance.
  - No inheritance.

Encapsulation

- Objects encapsulate their attributes and the behaviours.

This implies:

  - There is no interaction with an object that does Not go through a publicly published interface
  - Objects manipulate their own state; the definition of class includes the object's behaviour manifested as functions and procedures.
  - An objects state cannot be manipulated by anything external to them (at least, not without permission).

- For example, in a non object-oriented language such as C, let's say a programmer writes a procedure to change the values of a structure holding the position of a graphics primitive.
  - In an object-oriented language, the programmer creates a graphics primitive class that has its positional information along with an internal procedure that changes its own position.
  - The programmer sends a message to the object requesting it to change its own position.
- The advantage of encapsulation is that the implementation of any behaviour can be changed without effecting any other class in the system. This helps de-couple the classes and reduces the complexity of the system.

Comparison to the Relational Model

The object model differs from the relational model in (at least) the following ways:

- The object model allows complex objects to be attribute domains; this is prohibited in the relational model.
- The only complex type available in the relational model is the relation.
  a. The object model restricts all system entities to be objects which is a more general concept than a relation (relations can be objects but not all objects are relations).
- The relational model allows no duplicate tuples and, consequently, entities are identified by their attribute values.
  a. The object model assumes the existence of an object ID which uniquely identifies its object and is, possibly, invisible to the user.
- Objects are instances of classes and classes constitute the typing system of the model.
  a. There is no concept of class-level typing in the relational model; everything is a relation.
  b. The relational model supports user-defined domains but this is applied at the attribute level whereas, with the object model, the class is also a type.
  c. The equivalent in the relational world would be for relations to constitute types, as well.
- There is no generally accepted formal object model.
  a. The relational model is well-defined, sound, and complete.
     – Relations hold all tuples. There is no equivalent for objects; there is no set or anything else that contains all the objects of a class.

- There are many higher-order, non-programming query languages for the relational model. There are few equivalents for the object model.
- The object model is aimed more at programmers than at end users; the reverse is true of the relational model.