Software Tools and Methodologies for the Design
of Digital Electronic Systems

Presented at the CERN School of Computing, Oxford, 14-27 August, 1988
SOFTWARE TOOLS AND METHODOLOGIES FOR THE DESIGN
OF DIGITAL ELECTRONIC SYSTEMS.

M. F. Letheren
CERN, Geneva, Switzerland

ABSTRACT:
This paper discusses software tools and associated methodologies for the design of digital electronic
systems, focusing on the traditional types of tool that are widely available in commercial systems
today, leaving aside the more recent developments in tools for synthesis of logic or layout. The
areas examined are those of design capture, design verification through simulation and timing anal-
ysis, layout of boards and Application Specific Integrated Circuits (ASICs), and issues in testing
with emphasis on ASICs. Some comments are given on the practical experience gained with these
tools at CERN.

1. Introduction

The enormous advances in electronics of the last decades have been made possible by progress in
the three inter-related domains of technology, tools and methodology. The principal driving force
behind developments in electronics has been the steady advance in the technology of monolithic
semi-conductor integrated circuits, which has resulted in an increase in gate count of over five orders of
magnitude since the first ICs appeared. Other aspects of the improvements in technology are the dra-
matic reduction of manufacturing costs per gate, increased circuit speeds and enhanced reliability.

On the other hand, as system complexity has increased, design time and cost have also increased.
The cost of developing and applying adequate tests to complex systems has also exploded. The stimu-
lus of commercial competition has resulted in rapid evolution of tools, hand in hand with the develop-
ment of new design methodologies, in an attempt to handle the ever increasing complexity of design
and test. Our discussion of design tools will therefore also describe the associated design methodolo-
gies.

1.1 Scope of application areas and terminology

Development of electronic systems involves many different activities, most of which can be assist-
ed by software tools (see Figure 1). The application of tools to the conceptual design process is often
referred to as Computer Aided Engineering (CAE). The application of tools to the design of physical
implementations (board or integrated circuit layout) is categorised under the area of Computer Aided
Design (CAD), whilst tools related to manufacturing fall under Computer Aided Manufacturing
(CAM). Tools and methodologies for prototype and/or production testing come under the generic
term of Computer Aided Testing (CAT). Industry is making great efforts to integrate tools for factory
automation and management, an area of activity covered by the umbrella term Computer Integrated
Manufacturing (CIM).
1.2 The Design Process

Figure 2 shows how the typical design process starts from an abstract, high level system requirements specification, is gradually expanded through successively more detailed levels, finally ending with a complete description of how the system shall be implemented and tested. It is essentially a top-down process using the concept of hierarchy to handle design complexity. Most design methodologies simplify the design process by using a library of pre-designed and tested modules that can be inserted into the design hierarchy at an appropriate level (e.g. at board level these modules correspond to standard, mass produced components; at the IC level they correspond to so called standard cells or macro cells). When the library of components is used, the design process is a combination of top-down design (starting from the abstract system description), followed by a bottom-up phase (starting from the available components or macros).

In theory therefore, design is a linear top-down, bottom-up process with possible iteration between steps in order to correct errors. The goal of the design tools is to accelerate each of these steps, to eliminate trivial mistakes by automating the processing of large volumes of detailed design data, and to detect conceptual design flaws through simulation (or other verification processes) after each design step. When these goals are met, the number of lengthy, costly design cycles is reduced. However, the architecture of modern, integrated tool sets allows a more flexible flow with data feedback from one step effecting design decisions taken in another. In this way one can more easily realise highly optimised system designs and readily explore different design strategies.

1.3 Tool Classes

The traditional human-driven design style uses the top-down/bottom-up method enhanced with analysis tools for verification of each step. However, we are now witnessing the emergence of a class of tools that automatically synthesise detailed implementations of logic or structure (layout) from high level abstract representations\(^1\) (behavioural or structural), producing implementations that are correct by construction. The design is specified in a form independent of technology or process, and can be

---

\(^1\) It is interesting to note that a simple form of synthesis tool has been in widespread use for many years. Programmable logic assemblers and compilers synthesise a device fuse map from a behavioural description in the form of boolean equations, truth tables, or state diagrams.
compiled for a particular technology or process by a Silicon compiler or a Silicon assembler. Many people believe that synthesis tools, such as silicon compilers, will eventually have profound consequences on hardware design, just as high level languages have had in software engineering. The exciting area of hardware synthesis is covered in detail elsewhere in these proceedings [1].

These lectures will concentrate on tools developed for the traditional design methodologies, where the decomposition of the high level design representation into successively more detailed levels is still largely a manual process. The designer can use tools either to automate those steps that can be achieved through repetitive application of an algorithm, or he can use tools that check the correctness of manual steps by ensuring design rules have not been violated, or by verifying that the behaviour of a lower level representation is equivalent to that of a higher level representation (simulation). Different types of analysis tool can be employed for design verification after each stage of the top-down design refinement process.
Until recently, most CAE tools have been strongly typed, i.e. not applicable to both analogue and digital designs. For example, simulation of mixed analogue and digital designs has been broken down into the separate study of the analogue and digital parts using quite different tools. Now we are beginning to see the emergence of tools that can handle mixed analogue and digital designs. These lectures will concentrate on tools and methodologies for design of digital systems.

During a design project it is possible to select the 'best' tool for each task, but in practice, interfacing tools from different vendors is not without problems, and requires considerable investment for the development and support of interface software. For this reason many users prefer an integrated tool set supported by a single vendor, where all tools run on the same machine, have the same style of man-machine interface, avoid library compatibility problems by all using the same component libraries, and interface to the others through a shared design database (eliminating the need for format translation steps).

1.4 Process scaling, device and interconnect performance

Integrated circuit manufacturing processes can be characterised by their technology (CMOS, NMOS, ECL, etc.) and the minimum feature size (or minimum line width) that can be reliably manufactured. Leading commercial CMOS processes today use minimum line widths around 1 micron. It is interesting to see how circuit performance scales as the minimum line width is reduced by a factor $\alpha$. Under plausible assumptions, reference [2] shows that CMOS processes scale as follows:

- Transistor switching time $\propto 1/\alpha$
- Transistor power dissipation $\propto 1/\alpha^2$
- Number of transistors per unit area $\propto \alpha^2$ (or less)
- Total power dissipation per unit area $\propto$ constant
- Device power-delay product (a standard metric of performance) $\propto 1/\alpha^3$

Most chip architectures contain global interconnections that must traverse from one side of the die to the other. As a result of the very strong decrease in production yields with increasing die area, the largest die size that can be manufactured remains roughly constant at about 1 cm square. Thus large IC designs, that pack as many devices on a single die as possible will contain global wiring of the order 1 cm long. As the process dimensions are scaled down the resistance of global metal interconnect lines increases like $\alpha^2$ and their total capacitance remains constant (it is assumed that dimensions scale by the same factor in the vertical direction in order to avoid etching problems). The result is that global wiring delays actually increase like $\alpha^2$, while transistor switching times decrease like $1/\alpha$. Figure 3 shows that, for die sizes of the order 1cm, global wiring delays dominate over transistor delays for sub-micron processes. Today large CMOS designs, using processes with minimum line widths around 1 micron, demonstrate gate delays of the order 400 ps and global wiring delays of the order of a few nanoseconds.

Although technology improvements may reduce the interconnect delays (e.g. by replacing aluminium by a metal of lower resistivity), it seems that semiconductor process technology has now advanced to a point where some fundamental changes in design methodologies and tools begin to be necessary. These changes are not only required in order to handle the large volume of design data and the difficulties of testing complex devices. The emerging design world, in which device interconnect delays begin to dominate the intrinsic device propagation delays, will no longer be handled by a simple set of decoupled tools used in a linear sequence. Design capture, verification and layout will become closely integrated so that important layout constraints and critical parts of the circuit can be specified by the designer at the design capture stage. This information will be automatically forwarded to the
layout tool, which itself will be closely integrated with the design verification tools in order that post-layout verification can be done. The linear design process will become much more iterative. In the future, logic will become "cheap" and wires "expensive", reversing the situation with which most designers are accustomed, and probably acting as a major influence on developments in systems architectures, design methods and tools.

As device speeds and system clock frequencies increase similar timing problems will show up at the level of board layout. Board designers will have to pay more attention to the layout of critical wiring paths, clock skew, cross-talk, noise and other analogue-like behaviour of digital circuits. At the same time board layout is being complicated by new components with hundreds of pins, housed in new packages that require board layout techniques not easily handled by most existing tools.

2. Design Capture

Design capture is the process of entering the basic design specification in a machine-readable form. A design can be specified structurally (i.e. as a network of interconnected components), or as a behavioural specification without reference to internal structure, or by using a mixture of the two previous approaches.

Traditional engineering practice consists of building up a structural specification in the form of a schematic diagram, or netlist (a textual specification of the component types and their interconnectivity). Most CAE/CAD systems include an interactive schematic capture tool.

The behavioural form of design capture can be used to establish a formal, unambiguous "black box" specification of the functionality of a system or sub-system. It can be verified by simulation and can then be subcontracted to another designer, or a synthesis tool, to be turned into a structural description that can be built to implement the desired behaviour. Hardware description languages can also be used to specify the structure of a system algorithmically. For example, a high level procedural
layout description can be executed to generate detailed IC mask layouts.

2.1 Schematic Capture

Figure 4 shows how, using an interactive graphics system, a schematic is constructed by extracting component symbols from a component library, placing them and interconnecting them with wires and busses. Wires and busses can be assigned names, and components can be assigned certain parameters (e.g. a physical location reference parameter denoting the position of the component on a printed circuit board).

![Diagram](image)

*Figure 4:* Components of a typical schematic capture and compilation tool.

Usually the library definitions of the components include additional information such as part type, stock part number, package type, package pin numbers for each pin of the symbol, etc. This additional information is stored in the drawing file for later use by other tools. Usually the symbol library will also contain pointers to entries in another library, where further information about the components is stored. For example, there may be a pointer to an entry in a library used by a printed circuit board layout system. This would describe package geometries, power and ground pins, rules for swapping logically equivalent pins during layout etc. Other pointers could reference entries in a library containing models for a simulator.

Many schematic capture systems allow the user to modify parameter values associated with a component instance (a library can be thought of as holding component declarations, copies of which
are instanced in the schematic) and to add user defined parameters, or notes, in order to transmit specific information to another tool, or its operator. For example, one may wish to designate certain nets as critical to the layout system, or force a particular gate into a particular package during the gate packaging procedure.

Most schematic capture tools are implemented on workstations, or personal computers, that economically provide the medium to high resolution, medium speed, 2-dimensional graphics and comparatively modest computational resources required. Nearly all modern schematic capture systems have a user-friendly human interface, typically including the following features:

1. Operation under a multi-tasking operating system and a window manager allows simultaneous viewing of the schematic while running other applications (e.g. simulation).

2. Pop-up, or pull-down menus guide beginners and reduce the learning time.

3. Multiple, hierarchical menus reduce menu clutter and operator fatigue.

4. Use of icons in menus (of debatable usefulness).

5. Help windows are available to provide the user with context relevant information.

6. For experienced users, an optional command language interface (preferably allowing unambiguous command verb truncation) avoids tedious menu picking operations with the mouse.

7. Pan and zoom operations are simplified by providing an overview window that shows the complete schematic at reduced scale and outlines the working area, which is shown in detail in the main window. Panning is achieved by dragging the outline of the working area around inside the overview window. Changing the size of the working area outline in the overview window results in immediate zoom in (or zoom out) in the main window.

8. Operator fatigue can be reduced by using the technique of snapping a placed component onto a grid, or snapping a wire end point onto a nearby pin.

9. Individual objects can be collected into groups and then operations (move, copy, etc.) can be made on the group.

10. A sequence of commands can be captured in a user defined command macro and then the macro can be executed to speed up repetitive application of the same command sequence. A more flexible system displays the command macro in the command window and allows the user to edit it before it is executed.

11. All operator key strokes and mouse operations are recorded in a key stroke file. The key stroke file can be used to drive the schematic editor in playback mode in order to recover a work session after a crash of the schematic editor or operating system. It also forms an ideal mechanism for reporting bugs to the vendor.

12. An undo command reverses the effect of the last executed command, allowing recovery from mistakes or operations producing unsatisfactory results.
Hierarchical schematics support the top-down design methodology.

Many schematic editors support multi-level drawing hierarchies, enabling the top-down design methodology to be applied (see Figure 5). Each level of the drawing hierarchy can include multiple schematic sheets. An off-line compilation procedure interprets the schematic graphics, checking for electrical design rule violations. A linking procedure then links signals that traverse between sheets or between levels of the design hierarchy. Normally each page can be compiled separately, and sub-trees of the hierarchy can be linked (and verified by simulation) before the complete design hierarchy is entered.

The page compilation process checks the drawing for simple rule violations, for example:

1. Duplicate use of component or signal names.
2. Multiple outputs driving the same net.
3. On-page signals that either have no drive, or no sink.
4. Signals that go off-page but have not been assigned a name (and therefore cannot be linked to their counterparts on other pages).
5. Named wires that join to a bus whose separately declared contents do not include the name of the wire.

6. etc.

The compiled and linked design is stored in a design database. In an integrated CAE/CAD system, all tools will be driven directly by the design information entered into the database by the schematic editor. This eliminates the need of re-capturing the design in a form suitable for each new tool, and, even more important, eliminates inconsistencies in the different design databases used by the tools (in a heterogeneous system, built from loosely coupled tools, these may creep in through errors made in re-capturing the design for each different tool, through bugs in interface programs, or through forgetting to update a design modification in one tool's database). Tools that are not part of the integrated tool set can be interfaced to the database by user written programs that extract and format information into a netlist format accepted by the external tool. Integrated systems use databases in binary format (for efficiency), but usually provide a means of extracting (and sometimes also entering) data through a set of routines that can be called from a user program, thus hiding the detailed internal structure of the database from the programmer.

Many integrated tool sets are provided with design management software that keeps track of design versions and modification history. Simple automatic checks can prevent much lost time; for example, warning that a modified page has not been recompiled when the user tries to run the linker.

2.2 Design Capture by Hardware Description Languages

The schematic capture package is a natural tool for capturing structure. Many CAE systems augment schematic capture with structural or behavioural design capture by use of hardware description languages (HDLs). Some examples are the ISPS language (see references [3] and [4]) for describing and simulating computer architectures, or the well known languages PALASM [5] and ABEL [6] used to capture functionality for programmable logic synthesis systems. We will use the IEEE standard VHDL\(^2\) [7] as an example of design capture by hardware description language. VHDL was developed as a means of capturing a digital design in a standard form for purposes of design verification, synthesis and testing. A second function of the standard is to provide unambiguous, verified (by simulation) documentation for procurement of systems. The present text does not aim to describe VHDL in detail, for which the reader can consult references [8] and [9], but uses VHDL to give the flavour of design capture by hardware description language.

A typical VHDL support system, shown in Figure 6, allows module descriptions in VHDL to be compiled and stored in a database. Hierarchy is supported by allowing module descriptions to reference other modules. Separate compilation of modules is possible. A linker is used to expand the hierarchy and prepare a design for use by an application tool. The VHDL support system therefore acts as a design capture and management system for the applications tools.

In VHDL a hardware module is described by an entity. In all hierarchical systems, a module can be used as a component at a higher level of the design representation hierarchy. Such components are represented at the higher level by an abstraction that defines only their interface to the outside world. In a schematic capture system this abstraction of the interface is the component's graphical symbol. In VHDL, the corresponding component interface is the entity declaration, as shown by the trivial exam-

---

\(^2\) VHDL = VHSIC Hardware Description Language. VHDL was originally developed in the context of the United States' Department of Defence VHSIC program (VHSIC = Very High Speed Integrated Circuits).
ple for a 2-to-1 multiplexer Mux in Figure 7. The entity declaration names the entity’s ports and specifies their direction and type (TTL_BIT). In the example, the user-defined type TTL_BIT identifies the technology and data carrier width (1). Type checking avoids erroneous clashing of different technologies or attempts to link data carriers of different widths.

The contents of a VHDL entity are described by the VHDL architecture construct. A VHDL architecture can include instances of other separately declared entities, so that a hierarchical system description can be built up. The VHDL language allows architectures to specify the contents of an entity in 3 different styles:

1. **Structural:**
   This style is equivalent to design capture by netlist. An interactive graphics schematic capture system with a programmable netlist extraction interface could be used to produce a VHDL structural style description. Figure 8 shows a trivial structural VHDL architecture describing a 2-to-1 multiplexer at the gate level. The component types instance in the architecture are first declared, together with their port lists (which must match with the port lists made in the entity declarations of the components shown in Figure 7). Internal signals are declared and then used to specify the connectivity between the ports of component instances.

2. **Data Flow:**
   This style of architecture allows the modules to be described in terms of a data-flow diagram specifying the flow of data between a set of concurrent processes. In its simplest form, the data-flow style architecture can include a set of concurrent assignments to declared local signals, or the output ports of the entity. The order of execution of the assignment statements is driven by changes in the data carriers, not by the lexical ordering of statements in the source code, nor by flow-of-control constructs as found in procedural languages. The data-flow style description for the 2-to-1 multiplexer is shown in Figure 9. In this style the signals act as data carriers. The data-flow style does not define structure, it defines the flow and transformation of information. Using the possibility of having the assignments made only when a conditional guard expression
entity Mux is
  port (a, b, s: in TTL_BIT);
  port (c: out TTL_BIT);
end Mux;

entity Inverter is
  port (input: in TTL_BIT);
  port (output: out TTL_BIT);
end Inverter;

entity Nand2 is
  port (input1, input2: in TTL_BIT);
  port (output: out TTL_BIT);
end Nand2;

Figure 7: VHDL entity declarations for Mux, Inverter and Nand2.

becomes true, the dataflow style can easily describe functionality in terms of the familiar state machine model (see reference [9] for detailed discussion).

3. Behavioural:
This architecture style specifies the function of a black box using an algorithmic description. The behaviour of the module is described by a process using procedural programming language constructs of the VHDL language. No reference is made to internal structure of the black box (except that internal states will need to be defined, in the form of local variables, to correctly model the behaviour of a sequential circuit). Figure 10 shows a behavioural style architecture for the 2-to-1 multiplexor. The arguments of the process statement define the process sensitivity list, i.e. when any of the signals in the sensitivity list changes the process will be executed.

It is possible to mix the structural, data-flow and behavioural styles in the same VHDL description. Thus, instead of limiting the data-flow architecture to simple assignment statements, multiple procedural processes can be invoked concurrently in the framework of a data-flow architecture in order to model the parallelism of hardware. Each of the processes executes its code sequentially, but runs concurrently with the others. Language constructs are available to synchronise concurrent processes (e.g. WAIT on signal_name).
architecture gate_level_structural_example of Mux is

component Inverter
    port (input: in TTL_BIT);
    port (output: out TTL_BIT);
end component;

component Nand2
    port (input1, input2: in TTL_BIT);
    port (output: out TTL_BIT);
end component;

signal not_s, sig1, sig2: TTL_BIT;

begin
  I1: Inverter portmap (s, not_s);
  G1: Nand2 portmap (a, s, sig1);
  G2: Nand2 portmap (b, not_s, sig2);
  G3: Nand2 portmap (sig1, sig2, c);
end gate_level_structural_example;

Figure 8: A gate level structural VHDL architecture for the entity Mux.

The advantage of VHDL over a schematic approach is that the user can specify desired behaviour directly, instead of having to first "invent" a network structure of primitives that will exhibit the system behaviour he desires, and then capture that structure. In fact, one of VHDL's design goals was to support creation of technology independent system descriptions that can first be verified by simulation, then compiled by a synthesis tool into a structural implementation in a given technology. Thus using VHDL for behaviour capture for input to a synthesis tool is fundamentally different from using a schematic capture tool for input to verification and layout tools. At the moment the only application tools available for VHDL are simulators, but other types of tool, including synthesis tools, are under development [10].
architecture dataflow_example of Mux is

    signal not_s, sig1, sig2: TTL_BIT;

begin
  P0: not_s <= not s  after 2ns;
  P1: sig1 <= not a and s  after 5ns;
  P2: sig2 <= not b and not_s after 5ns;
  P3: c <= not sig1 and sig2 after 5ns;
end dataflow_example;

Figure 9: A data-flow VHDL architecture for the entity Mux.

architecture behavioural_example of Mux is

    initialize c: TTL_BIT to 'X';

begin
  process (a,b,s)
    begin
      case s is
        when '0' =>  c <= a  after 12ns;
        when '1' =>  c <= b  after 10ns;
        when 'X' =>  c <= 'X' after 10ns;
      end case;
    end process;
end behavioural_example;

Figure 10: A behavioural style VHDL architecture for the entity Mux.

3. Design Verification

The traditional method of verifying a design is to construct and test a prototype. However, in the case of integrated circuit design the cost and turn around time for manufacturing the prototype are sufficiently large to encourage designers to invest substantial efforts to produce designs that have a high probability of functioning correctly at production of the first prototype. This incentive, together with the availability of cheaper and more powerful workstation hardware, has led to the development and widespread use of simulation in IC design. The cost, sophistication and user-friendliness of these tools have now evolved to the point where their use in board level design is advantageous.
3.1 Overview of Simulation

Simulation is the time domain analysis of a network of interconnected functional modules, each of known behaviour. The purpose of simulating the network is to verify the design process; namely the (usually manual) decomposition of a high level representation of the system into a network of simpler modules. The design decomposition process may proceed through several levels as shown in Figure 5, and at each step the network may be simulated in order to check some, or all, of the following points:

1. it has functionality that is equivalent to that of the higher level system specification and reaches the desired level of system performance (maximum clocking rate, system throughput, etc.).

2. it does not contain timing problems (spikes, races, setup and hold-time violations, etc.).

3. it will function correctly when constructed with any combination of component samples, taking into account the range of timing characteristics guaranteed by the component manufacturer. This application of simulation is commonly referred to as timing verification.

4. it can be effectively tested for the presence of faults by the application of a set of specially developed test vectors. Evaluating the effectiveness of the set of test vectors is carried out via fault simulation.

A crucial issue in simulation is the accuracy of modeling. Most simulators have built-in modeling assumptions over which the user has little or no control, for example:

1. the representation of logic levels,

2. the modeling of technology dependence via logic strengths and the resolution of signal contention,

3. the modeling of the flow of time and the delay model.

Modeling decisions made on the above aspects determine the accuracy with which the simulator mimics the flow of signals between the functional units of the system.

The behaviour of the functional modules of the network can be modeled at different levels of abstraction. Some commonly employed modeling levels are the circuit, switch, logic, functional, behavioural, and physical levels. These modeling levels are discussed in section 3.7. The high level models hide detail and can usually be evaluated with less computer time than the lower level models. Often the designer will not have enough information about a component to develop a detailed model. The choice of device modeling level is a trade-off of modeling accuracy against run time and difficulty of development of the model.

There exist many different simulators that are specialised for modeling at one level of abstraction (e.g. the behavioural level, logic level, or circuit level). However, there is another class of simulator that, within the same tool, can support several levels in the hierarchy of modeling abstraction. These hierarchical simulators allow a single simulator to verify the top-down design process after each stage as it moves from the architecture level down to the lowest level of decomposition. A hierarchical sim-
ulator that allows "mix and match" of different levels of modeling abstraction in the same run is known as a mixed-level simulator. A simulator operating at only one level of modeling abstraction may well be superior to a mixed-level simulator at that level, but on the other hand, there may be a considerable interfacing effort needed to port the design from one tool to another as the design process progresses top-down. The mixed-level simulator clearly has advantages in large projects, where different parts of the design have advanced to different levels of detailed implementation. The mixed-level simulator also makes possible the simulation of systems containing complex standard parts (microprocessors etc.) that can be modeled relatively easily at high levels of abstraction, but would be very difficult to model at a low level, because of the complexity of writing such detailed models (even if all the necessary information about the internal structure of the part were available).

Today one can find many hierarchical, mixed-level simulators that span the range architectural to logic or switch level. The inclusion of circuit level modeling into this range is much more difficult because analogue simulators use fundamentally different models (discrete states and times are replaced by continuous variables, simple boolean expressions or algorithmic descriptions by sets of differential equations). Nevertheless, some companies are now advertising so called mixed-mode analogue/digital simulators. It appears that these are just the kernels of a digital and an analogue simulator tightly coupled through internal communication [11]. Severe performance problems may be encountered when circuits contain feedback between analogue and digital parts and the time advance mechanisms of the two simulators are not synchronised [11].

3.2 Modeling Multi-valued logic

Almost all digital simulators model at least 3 different logic values, namely '0' (false), '1' (true), and 'X' (unknown). The unknown value X is used to represent the condition of a memory device that has not been initialised, or the uncertain outcome of the operation of a device outside of its specified normal operating conditions (e.g. simultaneous assertion of the "clear" and "set" lines of a latch would be modeled by setting the latch with an 'X' value), or the clashing of different logic values of equal strength where the outcome of the contention is uncertain. The appearance of an X value in the simulator output signals a potential problem to the designer. Additional logic values are found in simulators that are used to verify system operation in the min/max range of device timing characteristics (see section 3.9).

3.3 Modeling Technology and signal contention

To be useful, a simulator should be able to model technology dependent behaviour accurately. For example, the outcome of clashing different logic values on a bus depends on the technology of the driving devices (open-collector TTL, tri-state TTL, ECL, etc.). Resolving signal contention is usually modeled by assigning a strength to each output and applying the rule that the strongest output dominates. Typically a simulator may work with 4 different strengths, in order of decreasing strength: Unknown (U), Forcing (F), Resistive (R), and High impedance (Z).

The full representation of the state of an output in a given technology then consists of the pair (Strength, Value). Table 1 shows the 12 possible states used by a simulator that models with 3 logic values and 4 strengths.

For example a totem pole TTL output stage can drive either an F0 or F1 state. An open-collector TTL output should be modeled by having it drive an F0 state when the output transistor is conducting and a ZX state when the output transistor is turned off (the output has to be pulled up to an R1 state
<table>
<thead>
<tr>
<th>Strength</th>
<th>U</th>
<th>F</th>
<th>R</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UX</td>
<td>FX</td>
<td>RX</td>
<td>ZX</td>
</tr>
<tr>
<td>0</td>
<td>U0</td>
<td>F0</td>
<td>R0</td>
<td>Z0</td>
</tr>
<tr>
<td>1</td>
<td>U1</td>
<td>F1</td>
<td>R1</td>
<td>Z1</td>
</tr>
</tbody>
</table>

*Table 1:* Typical states used by a 12-state simulator.

by an external resistor). A tri-state output drives F0 or F1 while enabled and ZX when disabled. The unknown strength U is used in those technologies where the strength of the 0 and 1 levels are different (e.g. NMOS or ECL); in such technologies any driver that drives an unknown level X must do so with an unknown strength U.

Table 2 shows how a typical simulator might resolve contention. Note that almost universally the simulator designers adopt the strategy of assigning the most pessimistic outcome. This is in order that potential problems will not be overlooked by the user. This strategy is sometimes disputed by first time users; it is however based on many years of practical experience with these tools.

Most simulators have their state and contention resolving models hard coded into them. They are therefore designed for use with a specific technology or set of technologies. The user has no control over the accuracy of these aspects of modeling. It is interesting to note that the VHDL language, on the other hand, has been designed to be technology independent — the user can accommodate new technologies or, to some extent, control the accuracy of modeling since he can declare a new set of logic states, and write a contention resolving function that, when called with clashing states, returns to the simulator the outcome of the clash appropriate for the new technology.

---

3 Some simulators simplify open-collector modeling by driving a Z1 state to the output when the logic output value is true. However, this is inaccurate and may result in the designer not noticing a forgotten pull up resistor.

4 Some simulators model an assignable decay time after the removal of the tri-state enable during which the output state takes the Z strength but retains the previous logic value. After the expiration of the decay time the logic value changes to the unknown X. This mechanism models the leakage of residual charge which maintains a definite level during a short time.

5 Some simulators encountered difficulties when the use of tri-state and bi-directional pins became common.
\[
\begin{array}{cccccccc}
Z0 & Z1 & ZX & R0 & R1 & RX & F0 & F1 & FX & U0 & U1 & UX \\
Z0 & Z0 & ZX & Z0 & R1 & RX & F0 & F1 & FX & U0 & U1 & UX \\
Z1 & Z1 & ZX & R0 & R1 & RX & F0 & F1 & FX & U0 & U1 & UX \\
ZX & ZX & R0 & R1 & RX & F0 & F1 & FX & U0 & U1 & UX \\
R0 & R0 & RX & RX & F0 & F1 & FX & U0 & U1 & UX \\
R1 & R1 & RX & F0 & F1 & FX & U0 & U1 & UX \\
RX & RX & F0 & F1 & FX & U0 & U1 & UX \\
F0 & F0 & FX & FX & U0 & U1 & UX \\
F1 & F1 & FX & U0 & U1 & UX \\
FX & FX & U0 & UX & UX \\
U0 & U0 & UX & UX \\
U1 & U1 & UX & UX \\
UX & UX & UX & UX & UX \\
\end{array}
\]

*Table 2: The 12-state contention resolving matrix.*

### 3.4 Modeling time and delays

Digital simulators may model time in such a way that an *event* (a change of state on a device output) can occur only at a discrete integer time value. For such simulators the basic time unit limits the resolution with which event timing can be simulated. However, if the time unit is chosen to be much less than the typical device delay, a fairly accurate simulation of the effects of differences in delays of different devices can be obtained. Other simulators allow events to occur at any time.

Simulators make various approximations to the propagation delays of devices. Commonly encountered delay value models are:

1. zero delay model
2. unit delay model
3. fixed delay model
4. assignable delay model

The zero and unit delay models were used in early gate level simulators; they trade off accuracy of timing for performance and simplicity of implementation by assigning the same delay (0 or 1 time unit) to all devices. The zero and unit delay models allow logic verification, but are ineffective at uncovering timing problems in the design. The fixed delay model accurately specifies a different delay for each device type, but each device of a given type has the same delay. Assignable delays allow each instance of a device type to be assigned its own delay value (permitting the modeling of the effect of fan-out load on circuit speed).

Another important aspect of delay modeling is that of the signal transport mechanism through the device. The 4 principal models used are described in Figure 11.
3.5 Components of a simulation system

Figure 12 shows the basic elements of a simulation tool. The simulator requires information about the connectivity of the network under simulation and the models for the functionality of the components. Connectivity data is provided by programs that parse a netlist, or a set of schematics. Modeling data are provided by a modeling language compiler. Most simulators are driven by a set of linked list data structures (called tables hereafter) containing fanout lists for each output, signal state tables, pointers to modeling primitives, timing parameters, etc. A linking stage builds the exact data structures required by the simulator. A simulator process executes the simulation algorithm and is
controlled and monitored by a user interface process that mimics a software logic state analyzer. An interactive monitor task, analogous to a software debugger, allows the user to set break points, trace signals, inspect and initialize internal model states, etc. User friendly monitor tasks present waveforms graphically, and are often tightly coupled to the schematic editor (running in another window), so that the user can pick signals and devices from the schematic window and immediately see their state reflected in the monitor's window. Not all simulators are truly interactive; some dump the trace of selected signals in a file which is subsequently analyzed by a post-simulation processor.
3.6 The event driven simulation algorithm

Most digital simulators use the principle of the event driven algorithm shown in Figure 13. Event driven simulation drives an input stimulus forward in time, propagating its effect through all branches of the fanout tree. System behaviour is modeled as a series of discrete events in each of which the new state of a device output is propagated to the associated signal. When an event occurs, the change is propagated through the signal's associated fanout list to all device inputs connected to the signal. Many events can occur concurrently, so all events at the current simulation time are first propagated to their fanouts. Next, each of these fanout devices is evaluated using the new states of their fan in signals, i.e. their future changed output states are calculated and scheduled to produce new events after the elapse of the device's delay time by queuing them in an event queue. Once all fanout devices have been evaluated time is advanced to the time of the next scheduled event(s) found in the event queue. Events are removed from the queue and used to drive the simulation through another time step. The principle of selective trace is used to speed up execution by cutting off propagation of device outputs that do not change from their previous value (shown in the flow diagram of Figure 13).

The implementation details of the time advance mechanism and the event queue management algorithm also affect the efficiency of the simulator. Some simulators use a fixed time increment mechanism, which often requires many wasteful time steps to be executed in the event queue manager before simulation time is advanced to the time of the next scheduled event. Other event queue managers are able to jump simulation time directly to the next scheduled event time, so saving the overhead of searching the event queue at times when it contains no events for execution. There are many different detailed implementations of the event queuing algorithm in use in real simulators. Figure 14 indicates the principle of the widely used time wheel event queuing method.

Event driven simulation is the most common, but not the only algorithm. For example, reference [12] describes a demand driven algorithm which starts from a user request for the state of a particular node, and drives this demand backwards in time through the network until it can be resolved in terms of the network's input stimuli. This algorithm does not waste time propagating the simulation down branches of the fanout tree that the user will not look at. In addition it removes the overhead of maintaining an event queue. It is claimed to run 2 to 3 times faster than event driven simulation.

3.7 Modeling Device Behaviour

The applications for which a simulator is suitable depend on the level of modeling abstraction supported by the evaluation routines used to calculate new states at the device outputs and schedule them in the event queue. The main levels of device modeling abstraction in use are described below.

3.7.1 Circuit and Analogue Behavioural Modeling Levels

Analogue simulators model devices at the circuit level by treating time, voltage and current as continuous functions. A set of differential equations are solved using numerical integration methods and the circuit input stimuli as boundary conditions to derive the circuit's behaviour. This method is fundamentally different from the event-driven digital simulation algorithm in which the variables are restricted to a small set of discrete values. It is briefly mentioned here for the sake of completeness; more detailed information can be found in [13].

The widely used program SPICE [14], originally designed for analogue simulation of full custom integrated circuits, has the analogue models of primitive devices built into the source code. SPICE
Figure 13: Event driven simulation.

Selective trace cuts off propagation through the fanout list whenever an evaluated output does not change.

Primitive models include resistors, capacitors, inductors, transmission lines, voltage and current sources, diodes, bipolar junction transistors (BJT), junction field effect transistors (JFET), MOS FETs, etc. These generic device models are characterised for a particular device and process technology by supplying values for model parameters (e.g. the Gummel-Poon BJT model requires 40 parameters). A difficulty often encountered with SPICE is to obtain the set of parameter values that characterise a particular device or process. Another practical difficulty is that the numerical methods used to solve the set of simultaneous differential equations do not always converge.

More recent analogue simulators like SABER [15] are provided with a library of circuit level device models and a modeling language that enables users to develop their own analogue models (these can describe any analogue behaviour, viz: electronic, mechanical, etc.). The SABER modeling lan-
guage can be used to build models at the circuit level (by a composition of analogue primitives), or at the higher, analogue behavioural level (where a black-box transfer function is used without reference to circuit structure). With programs like SPICE, the computational complexity of circuit level modeling limits their application to relatively small circuits (a few hundred devices) in practice. Larger analogue systems can be modeled using the analogue behavioural modeling approach.

3.7.2 Switch Level

Switch level modeling is used to simulate larger digital MOS circuits with less accuracy, but higher execution speeds than circuit level simulators can achieve. The MOS transistor is modeled as a switch with some simple analogue behaviour. The switch level includes modeling of transistor delay, rise and fall times, loading effects and effects such as charge sharing (i.e., redistribution of stored charge when a pass transistor opens a path between two isolated circuit nodes).

3.7.3 Logic and Functional Levels

A logic simulator contains logic level primitive behaviours hard coded into its evaluation process. Typical primitives encountered at this level are logic operators such as NAND, NOR, NOT, AND, OR, XOR, a DELAY primitive, MOS pass transistors, and memory elements such as various types of flip-flop and latch. These primitives can be directly invoked to provide the behaviour of a simple logic device; in addition the user will have to specify the required timing information (delays, and timing constraints such as maximum clock frequency, minimum pulse width, setup and hold times, etc). Table 3 shows the behaviour of the AND primitive used by a simulator employing 3-valued logic (0, 1, X).

The user models his system by specifying the structure of the network of logic level components. The VHDL structural architecture shown in Figure 8 describes a gate level model of a multiplexor. More complex components that cannot be directly modeled by one of the logic simulator's primitives are handled using macros. An equivalent circuit of the complex component is declared using logic simulator primitives. This may be done graphically, or in a netlist style. The equivalent circuit is
stored in a macro library. A macro processor will then expand any occurrences of the complex component in the system structural description into logic primitives. The simulator still operates at the logic modeling level and no performance improvement is achieved through higher level modeling.

A functional level simulator may have built-in, higher level primitive models such as random access memory (RAM), read only memory (ROM), programmable control structures (PLA), arithmetic logic units (ALU), etc. Often a functional level simulator will allow modeling of device behaviour using boolean expressions or truth tables. These will be directly used by the simulator’s evaluation routines and will not be broken down into an equivalent circuit. Performance is significantly enhanced by this modeling approach as only one evaluation pass is required and it avoids the overhead of evaluating the multiple devices of the equivalent circuit and scheduling events through the fanout tree.

A functional model will be described using a non-procedural, data-driven language (i.e. the order in which statements are written has no influence on the sequence in which they are executed; execution order is determined uniquely by changes in the data and by data inter-dependencies). Figure 9 shows a simple example.

### 3.7.4 Behavioural Level

Although commonly called the behavioural modeling level, a better description for this level of modeling would be algorithmic. The behavioural modeling approach can describe the functionality of a black-box without reference to its internal structure. This is done using a behavioural hardware description language (HDL). Behavioural HDLs are usually based on extensions to familiar, procedural programming languages such as C, Pascal, Ada, etc. A trivial VHDL behavioral model of a multiplexer is shown in Figure 10.

Most HDLs allow the description of both structure and behaviour, so that a system can be described in terms of a network of black boxes. The behaviour of each black box is described using the behavioural description features of the HDL. The HDL contains extensions to the standard programming language that make the description of timing and concurrency of hardware possible. Generally, multiple processes can be declared and will be simulated pseudo-concurrently in order to model concurrency in hardware. Each process consists of code that is executed sequentially. Whenever the simulator needs to evaluate a black box its evaluation routines (see Figure 13) call the corresponding process code. Processes can be synchronised by using either statements that suspend execution of a process until a named signal is set (or reset), or statements that fire up a process whenever a named event occurs.
Thus the HDL will be compiled into code sequences that can be called by the simulator’s evaluation routines to return logic behaviour. At the same time the compiler extracts timing information into the tables used by the simulator’s event scheduling mechanism. For example, the sequential code fragment (consisting of only one case statement in this trivial example) that forms the body of the process declaration in Figure 10 will be used to evaluate the output of the multiplexor whenever one of its sensitive inputs (listed as arguments of the process statement) changes. The delay times specified in the after clause will be used to schedule the associated output event in the event queue.

Some systems actually compile the behavioural description into code using the simple primitive evaluation models and operators built into a logic simulator. This allows the behavioural description to be mixed with logic level descriptions and run on a logic level simulator. In this case the user works at a high level of abstraction but the potential for performance improvements through the high modeling level is not fully realised. The advantage of this approach is that the simulation can be accelerated by moving to an existing special purpose hardware implementation of the logic level simulator (see section 3.8).

3.7.5 Physical Modeling

In cases where insufficient information is available to allow the writing of a behavioral description, or the investment required to write the model is judged to be too high, the technique of physical modeling can be used. A real sample of the chip or board (the physical model) is stimulated with the input pattern generated by the simulator and provides the behavioural response to the simulator as shown in Figure 15.

![Figure 15: The principle of physical modeling.](image)

This technique is commonly used to model microprocessor ICs or other complex VLSI circuits. However, many VLSI circuits employ dynamic MOS technologies and will not function correctly if they are clocked at a frequency below some minimum value (which can be as high as several MHz). For these circuits the physical modeling hardware must buffer the entire stimulus pattern history in a FIFO buffer memory, from where it can be replayed at full speed. As soon as the simulator needs to evaluate the next state of a component being modeled with the physical modeler, it adds the current input stimuli to the input pattern history buffer, and then plays back the stimuli to the physical model at a selectable clock frequency (typically up to 20 MHz). The last output pattern generated by the model is picked up by the simulator.
The width and depth of the history buffer determine the number of pins the model can have and the maximum number of clock cycles that can be simulated. Some physical modeling systems allow compression of the history buffer by eliminating all patterns where only data pins are changing (only changes on store pins effect the internal state of the model) [16]. If a single physical model is used to provide behaviour for several instances of the component type, a history buffer needs to be kept for each model instance (even when a static MOS device is being modeled). In order to circumvent this limitation, most physical modeling systems are modular and allow multiple physical models to be used simultaneously.

The physical model only provides the simulator with response patterns; the designer must still write a description of the timing delays and constraints for each pin. In addition, because physical modelers only sample the value of model outputs, the user has still to write a description predicting the strength of the outputs. When physical modeling is used with a software simulator, the overhead of doing I/O in a multi-tasking operating system and the necessity of replaying the complete contents of the history buffer at each evaluation, can result in relatively slow performance. Nevertheless, the performance is likely to be faster than a behavioural model for complex devices, and probably most of the time will still be spent in the software simulating the other parts of the system. The I/O bottleneck can be avoided by integrating the physical modeling system into a special purpose hardware accelerator for simulation (described in section 3.8). For industry-standard microprocessor and peripheral chips, an alternative to the expense and complication of physical modeling is to purchase high level behavioural models from specialist modeling companies.

3.8 Hardware Acceleration of Simulation

Simulator performance is usually quoted in terms of events/sec, but sometimes in terms of evaluations/sec. With a typical network each event results in about 2.5 evaluations. A software implementation of the event driven algorithm typically results in execution rates of about 1000 events per second per MIPS of CPU speed. Thus, typical workstations today run event-driven simulations at speeds of a few thousand events per second. The amount of work done by one event depends on the level of modeling adopted. When a high modeling level is used, more work is done per event and relatively less time is spent on the overhead of the event driven simulation algorithm, resulting in improved speed compared with what can be obtained at more detailed modeling levels.

The relative simplicity of the event driven algorithm lends itself to implementation in special purpose hardware. Various hardware implementations have been made resulting in execution speeds ranging from around 40,000 events/sec (cost $25K) up to around 10^6 events/sec (cost $3M).

At the low performance end of the hardware accelerated simulators are architectures like the Personal MegaLogician from Daisy Systems Corporation [17] shown in Figure 16. This consists of a straightforward implementation of the 3 main parts of the event driven algorithm in three microprogrammed units (the state, evaluation and queue units). Each unit pipes its results into FIFO buffers at the input of the next unit, enabling all three units to work concurrently. Each unit has an attached memory that contains the tables used by the simulator. The size of these tables limits the maximum size of the network that can be handled by the hardware engine. Behavioural models are compiled into descriptions using model primitives of the hardware simulator's evaluation unit. The Personal MegaLogician uses the same model primitives as Daisy's software simulator and produces identical results. The physical modeler is integrated into the accelerator as a special hard wired unit that can work in parallel with the evaluation unit. The accelerator integrates well into the software simulation environment of that company and is priced to offer an attractive cost/performance advantage over a software simulator running on a super workstation.
At the high performance end of the simulation hardware engines we find several architectures which use multiple, concurrently operating, special purpose engines, each one with its local event queue, state unit and evaluation unit. The circuit under simulation is partitioned between the engines. Signals that cross between parts of the circuit assigned to different engines are handled by sending short messages over a high performance cross-bar network. The messages contain data that allow the sending unit to enter an event into the receiving unit’s event queue, or to push an evaluation packet onto the receiver’s evaluation stack. Each unit has to keep simulated time in lock step with a master time advance mechanism.

These architectures are modular; one adds modules until the hardware simulator has the capacity to handle the system to be simulated with the desired level of performance. Mapping the circuit to be simulated onto the multiple engines must be done in a way that minimises inter-engine communication and, at the same time, distributes the activity of the circuit uniformly over all engines so that none of them ends up idling while waiting for an overloaded engine to catch up. Reference [18] discusses the various performance related issues in parallel simulation. Some examples are the accelerators from Zycad [19], or Daisy’s GigaLogician [20]. IBM’s Yorktown Simulation Engine [21] is a highly parallel machine which, in contrast to the others, does not use event-driven simulation. Every gate is evaluated at every cycle, even if its inputs have not changed.

These high speed accelerators can be extremely fast, but one should be aware that most of them have sacrificed accuracy and flexibility for speed. Some only directly support low levels of modeling (e.g. the gate level), although some of them are provided with software that can compile behavioural models written in certain behavioural modeling languages into an equivalent gate level structural description. Others rely upon a relatively slow host computer to execute behavioural models, or to provide primitives for RAM, ROM, PLA etc. Others, like the GigaLogician, include multiple parallel hard wired units for logic level primitives, and multiple programmable processors for execution of sequential, high level language behavioural descriptions. Accelerators may sacrifice accuracy of delay modeling (using for example the unit delay model) for simplicity of implementation, perhaps forcing the adoption of a strictly synchronous design methodology. They may require time consuming translation and interfacing procedures in order to use them. They are typically used by large systems houses that can afford the necessary investment and support effort, and enforce a rigorous design methodology needed to make them useful.
Depending on the length of a simulation run, the extra compilation phase required to prepare a design for hardware assisted simulation may actually take more time than is saved by speeding up the simulation run. In these circumstances, a more powerful general purpose workstation may give better overall improvement, because it speeds up both compilation and simulation. The hardware simulation engine can be most useful towards the end of a design project, when very long runs may be needed to catch the few remaining bugs, or when the simulator is used as a work bench on which to develop prototype operating systems software for an as yet unbuilt computer, or to test applications software for an embedded processor. Another area where hardware simulation engines find application is in computation intensive fault simulation (see section 4.1).

3.9 Timing Verification

This is an important area in which a network is examined for correct operation taking into account the possible statistical spread of timing characteristics of each component. The *breadboarding* technique, in which a physical prototype is constructed and tested, may well conceal potential timing problems that later show up during production when a batch of components with slightly different timing characteristics is used. Costly redesign and field modification may then be necessary to correct the design error and, in addition, much system time may be lost tracking down the problem and getting it fixed. For example, using a timing verifier to check the design of data-acquisition electronics might save the extremely costly use of beam and experiment time to track down design errors.

As already mentioned in section 3.2 the timing verifier uses more logic values than the simulator in order to identify times in which a signal’s state is not guaranteed to be known. Figure 11 (d) shows how a transition from 0 to 1 on the input of a device is propagated to the device’s output. During the time slice defined by the device’s ambiguity delay the output state is not known with certainty, all we know is that somewhere in this interval it will change from 0 to 1. In the timing simulator, the output signal is assigned the value rising (r) during this period. Other possible logic values used by timing verifiers are typically falling (f), changing (c) (when the signal could be either rising or falling), or stable (s) (the level is known not to be changing, but it is not known whether it is is 0 or 1). Table 4 shows the truth table for an AND gate as used by a typical 6-valued logic timing verifier. The timing verifier would typically use 6-valued logic together with 4 strengths, i.e. 24 states.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>s</th>
<th>c</th>
<th>r</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>s</td>
<td>c</td>
<td>r</td>
<td>f</td>
</tr>
<tr>
<td>s</td>
<td>0</td>
<td>s</td>
<td>s</td>
<td>c</td>
<td>r</td>
<td>f</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>r</td>
<td>r</td>
<td>c</td>
<td>r</td>
<td>c</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>f</td>
<td>f</td>
<td>c</td>
<td>c</td>
<td>f</td>
</tr>
</tbody>
</table>

*Table 4: Truth table for the AND primitive in 6-valued logic.*

(s = stable, c = changing, r = rising, f = falling)
Most timing verifiers allow the user to choose the timing range values for the ambiguity delay from the minimum, nominal and maximum delay values for the components. Thus the timing verification can be done using min/max, min/nom, or nom/max ranges. Potential timing problems will show up, for example, when the data input (D) of a flip-flop is falling, rising or changing within a time window defined by the range in which the clock is in the rising state (plus and minus the setup and hold times of the flip-flop).

For synchronous designs, the designer can apply the stable state to all primary inputs of the circuit except the clock input. All possible timing problems associated with clocked elements will then be detected. Value independent timing verification saves the designer the task of inventing an exhaustive set of test patterns to exercise all possible delay paths to the clocked elements.

### 3.10 Using Simulation Tools in practice

Although, in principle, it is now possible to simulate almost any digital design by using the combined arsenal of techniques outlined above, not every designer works in an environment where the necessary financial investment and support is available or can be justified. Nevertheless, experience at CERN has shown that a large fraction of board level designs can be profitably simulated using affordable software simulators on CAE workstations. Table 5 shows job statistics for a design project that was carried out at CERN using CAE and CAD techniques.

<table>
<thead>
<tr>
<th>ACTIVITY</th>
<th>NUMBER OF WORKING DAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feasibility study (choose between analogue or digital solutions).</td>
<td>14 (12%)</td>
</tr>
<tr>
<td>Schematic capture, modeling FADC chip, simulation and modification of design.</td>
<td>33 (28%)</td>
</tr>
<tr>
<td>Design FADC daughter boards, setup test bed, test analogue daughter boards, etc.</td>
<td>20 (17%)</td>
</tr>
<tr>
<td>Design 2 wire wrapped CAMAC boards using netlist driven software (WRAP4 [22]).</td>
<td>10 (9%)</td>
</tr>
<tr>
<td>Write test programs, design small test circuits, carry out tests, file documentation and deliver tested module to the experiment.</td>
<td>40 (34%)</td>
</tr>
</tbody>
</table>

*Table 5: Statistics for a design job using CAE and CAD tools.*

The job was the design, construction, testing and documentation of a *Missing Momentum Trigger Module* for the CERN UA2 experiment [23]. It consisted of 150 ECL integrated circuits and 12 Flash Analogue to Digital Converters (FADCs) implemented on two CAMAC boards using wire wrap. The project was completed in an elapsed time of 6 months and no design errors were detected during the testing of the prototype.
Simulation can be viewed as a valuable management tool as well as an engineering tool. In industry, large designs are often partitioned between many engineers, and the interface specifications (behaviour and timing) can be precisely defined using simulation top-down. The project coordinator can guarantee successful system operation as long as the interface specifications are met. Using simulator results, he can supervise the work of each engineer and get an early warning if part of the design cannot meet the specifications, perhaps deciding to revise system architecture. Although at CERN we have not used simulation for co-ordinating large scale projects, it has been found to be a very useful tool for supervision of an individual's design work [24], allowing identification of problem areas and fast feedback on alternative solutions.

Apart from possible run time limitations for large designs, the software simulation of semi-custom digital ASIC designs is usually relatively trouble free, largely due to the absence of device modeling problems as a result of the rather simple building blocks provided in the average silicon vendor's cell library. However, approaching the design top-down, one should first develop a behavioural model of the ASIC that allows the ASIC architecture to be validated in its final system application. Many ASIC designs successfully implement the designer's intent, but have to be changed because the design goals were wrong. Depending on the other components used in the system design, simulation of an ASIC embedded in a system could be a relatively complex and costly task if it requires use of physical modeling or hardware accelerators.

Although simulation can demonstrate substantial benefits in appropriate design projects, it does have a few problems that need to be understood before designers can begin to reap the full benefits. In the remainder of this section we will discuss some of these practical problems. Simulation involves modeling decisions in several areas as outlined above. It is therefore an approximation to the truth. Simulation results can only be evaluated and weighed if the designer has a good understanding of the approximations that have been made in the simulator itself and in the models for the components that have been used. Due to inadequacies in the simulator, or the models, unexpected results can be produced in certain circumstances. The results of the simulator should always be critically reviewed, and after some experience the user will know the weaknesses of his simulator.

A simulator will not of itself find all the design errors in a system. The user still has to invent an appropriate set of test stimuli that will exercise the system and show up design errors. As with most tools their effectiveness depends very much on the skill of the user.

Almost all designers of simulators have adopted the policy of pessimism, i.e. when the exact outcome of an operation is not well defined the simulator will assume the most pessimistic outcome possible. The aim is to make sure that any possible error does not get by unnoticed. Most experienced users prefer the occasional trouble of trying to understand why the simulator reported a grave error when in reality there may not be a problem in the real circuit, to having a real problem slip through undetected.

First time users of simulators do not always have a thorough understanding of the limitations and approximations involved in the tools, or of the philosophy of pessimism built into the simulator. The considerable effort required to develop reliable models for components is often underestimated. The first time simulation user lacks experience to trade off the advantages of using a "new" component against the cost of introducing it into the simulation model library. In fact, even today, many hardware designers lack the basic software skills and understanding that is necessary to develop models. They may adopt the unsatisfactory solution of relying on an "expert" to write the model for them, in which case they will not understand the limitations and approximations made by the modeling expert, and maybe have problems interpreting simulation results. In short, initial expectations are often too high and, depending on the user's skills, the learning curve for introducing simulation into the design
method can be quite long. Nevertheless, once the user has progressed up the learning curve, the “pay back” can be very substantial.

We will next describe a few specific problems or limitations that may be encountered in some simulators. A common problem is an over pessimistic treatment of unknown states in certain cases, as illustrated by Figure 17. In this example the unknown state X on the signal A produces unknown states on the two inputs of the AND-gate, and hence leads the simulator to assign an unknown state to the output of the AND-gate. In reality the output of the AND-gate will always be 0 (once the circuit has settled to a stable state) because its two inputs always take opposite logic values due to the inverter. In order to correctly simulate this circuit the simulator would have to recognise that the two inputs to the AND gate are correlated because they reconverge from a single fanout source. The problem is not serious because a well debugged circuit should not usually generate unknowns (except immediately after initialization). However, it does contribute to the propagation of unknowns through a circuit during the debug phase, making it more difficult to find the source of unknowns in a large circuit.

![Diagram](image)

**Figure 17:** Incorrect output when reconvergent fanout is not recognised.

A reconvergent fanout problem can occur with timing verification of circuits containing clocked devices when the timing verifier does not recognise the correlation that exists between the clock edge and changes on the device outputs. A time range in which the clock is rising will produce an output signal which is changing in an even greater time range. If the output signal reconverges with the clock signal at another device, the timing verifier may not recognise that the two signals are correlated and it may therefore report a false timing problem. Figure 18 shows a very simple circuit that may not be correctly simulated on some timing verifiers due to the reconvergence (in this case on the same device) of the flip-flop output with the clock. To function correctly, the timing verifier needs built-in knowledge that tells it that the flip-flop's output will not change before the clock edge rises, and it must recognise the reconvergence in the circuit connectivity. Although most timing verifiers handle the reconvergent fanout problem correctly, certain pathological circuit topologies may be incorrectly handled by some of them, leading to false reports of timing problems.

Another problem, occurring with value-independent timing verifiers, is that they are usually too pessimistic, reporting timing problems that can be ignored because the circuit operating state in which they appear is never reached under actual operating conditions. Timing verifiers that allow verification under specific stimulus patterns circumvent the effort of sorting out whether a reported error can be ignored. On the other hand, they place the onus on the user: to invent a set of stimuli that explore all operating conditions that will be encountered in practice.
A common problem is the one of initialisation of a circuit before simulation starts. Before starting to execute with the user-defined stimuli the simulator will run through an initialisation phase where it assigns a consistent set of initial values to every node in the circuit. Normally any memory elements (RAM, latch, flip-flop etc.) are preset to an unknown state before simulation starts. In reality these memory elements will quickly fall into one of their two stable states on power up. The design may not apply a master reset signal to all memory elements because the application of the design may be insensitive to the initial states of the memory elements. The simulator will then propagate unknown states through the network, and in cases where the unknown state forms part of a feedback loop (e.g. a divide by two circuit) the simulated circuit can remain locked in the unknown state. In this case, the user will have to explicitly initialise the memory elements in a consistent way before starting simulation. Identifying all such feedback loops and correctly initialising them could be quite time consuming in a large design.

These are typical examples of minor shortcomings in the tools that the beginner will encounter and must learn to circumvent by adapting his working methods. Unfortunately many simulator vendors neglect to document the limitations and modeling accuracy aspects of their simulators, leaving the users to find out by experience and perhaps resulting in some initial disappointment. User groups and informal contact with experienced users can be invaluable for spreading this type of expertise.

---

6 A system that cannot be brought into a known state with a master reset signal is usually regarded as untestable. However, for board designs which will not be tested on an automatic board tester, designers sometimes leave out the master reset in order to gain board space, or to be able to build a faster circuit.
4. Testing

In the manufacture of integrated circuits only a certain fraction (the yield) will be free of defects and fully functional. It is important to be able to rapidly reject faulty circuits with a simple set of tests that guarantee to filter out (in principle) 100 per cent of the faulty devices. Improvements in process technology permit ever more complex ASIC designs to be produced, and so testing has become a much more critical, difficult, and expensive issue. As a result many software tools are now available to help with the development of test patterns, and design methodologies have been developed to ease the problems of testing. Of course testability is also very important for board level designs wherever large quantities will be produced.

4.1 Fault Simulation

Fault simulation techniques are used to study the behaviour of a digital system in the presence of manufacturing faults. A single fault is injected into the simulated circuit to produce a faulty machine. Each possible faulty machine is then simulated and its behaviour is compared with that of the fault-free machine.

The inputs and outputs through which a system can be accessed for testing are called the primary inputs (PIs) and primary outputs (POs) respectively. In production an IC can only be tested via its input and output pins, whereas testing a board is in principal somewhat easier because a bed-of-nails tester or a special test connector can be used to drive or sample almost any circuit node. A tester will be used to apply a series of test patterns to the PIs of the device (or board) under test, and for each one compare the pattern produced on the POs against that calculated by the simulation of the fault-free machine. A single input test pattern, together with the expected device output pattern, is called a test vector.

The set of test vectors required to adequately test complex VLSI circuits can be very long and extremely hard to construct. Some software tools are available for automatic test pattern generation (ATPG). Reference [25] describes algorithms for ATPG. ATPG tools are very expensive and do not remove the need for the designer to understand testing issues as they still require much designer guidance and a testable design.

Once a set of test patterns has been generated, the technique of fault simulation is used to:

1. Calculate the fault detection coverage of the test vector set, i.e. the percentage of all possible faults that are detectable with the test vectors.

2. Grade faults; i.e. to produce a list of faults that are detectable by each test vector. This may allow some test vectors to be eliminated because they only detect faults that have already been detected by a previous test vector.

3. Construct diagnostic dictionaries. A single test vector may detect multiple faults, so that it cannot be used to determine why the device failed. However, if the information provided by the results of all the test vectors is combined, it may be possible to identify unambiguously the single fault that caused the device to fail. Depending on the quality of the test vectors, it may happen that multiple faults lead to the same set of failing test vectors. A diagnostic dictionary can be built up to map the set of failing test vectors, and the actual pins that failed, into a short list of candidate faults respon-
sible for the failure. Although this might be expected to be very useful for diagnosing failures, it appears in practice to be of limited usefulness due to the inaccurate assumptions made in the fault models (see next section); often the diagnosis is misleading or gives a large number of candidates.

4.1.1 Fault models

There are many possible faults that can occur in an integrated circuit. Some examples are that nodes may be stuck-at-one (SA1), stuck-at-zero (SA0), two nodes may be shorted together, a node may be floating, etc. Fault simulation is almost always limited to modeling only a single SA1 or SA0 fault at a time. In practice, the computational complexity of fault simulation excludes the study of multiple faults and more sophisticated fault models. However, it appears that test vectors generated using the single SA0/SA1 fault models are quite good at detecting multiple faults and most of the other types of fault.

4.1.2 Fault Simulation Algorithms

Fault simulation is enormously expensive in computing resources. A circuit with N nodes will require 2N faulty machines to be simulated with the test vector set in order to evaluate all possible single SA0/SA1 faults. To stand a chance of detecting structural failures in the system, the fault simulation must be done at a detailed level where all N nodes will be exercised (usually, but not necessarily, at the logic level).

In serial fault simulation each faulty machine is analyzed in a separate simulation run, requiring (2N + 1) runs (including the run for the fault-free machine). It is easy to understand how development of tests for VLSI ICs can take hundreds of hours of mainframe CPU time [26].

Parallel fault simulation [27] uses a technique of packing node values (2 bits are sufficient to carry the values '0', '1', 'X') for several faulty machines into a W bit wide word and then running them all through the fault simulator in the same run. This reduces the number of runs required by a factor \((W - 1)/2\).

However, the most efficient algorithms use the technique of deducitive fault simulation, in which a fault list is propagated through the circuit, rather than trying to propagate the effects of the faults through the circuit [28]. All faults are then handled in one pass. Although only one pass is required compute times are still very long. Nearly all fault simulators on the market today use the concurrent fault simulation algorithm [29], an optimised form of the fault list propagating class of algorithms.

Finally, hardware accelerated fault simulation is also used to bring fault simulation run times down to manageable proportions. An example is the Megafault [30] from Daisy Systems Corporation, that runs on the same microprogrammable hardware as their MegaLogician.
4.2 Tools and methods for design for testability

Even for moderately sized semi-custom IC designs a major part of the development effort may be spent on developing an adequate set of test vectors [31]. Consequently, a class of tools have been developed that quickly give the designer a measure of the testability of his design. With relatively little effort these tools give an early indication of the feasibility (and effort required) to produce a set of test vectors providing an adequate fault coverage. The designer can modify his design until the testability analysis indicates that the subsequent effort required for test pattern generation and fault simulation is acceptable.

Some testability analysis tools evaluate a design by looking for the presence (or absence) of features that are known to normally enhance (or reduce) testability (e.g. test points, controlled breaking of feedback loops, initialisation of flip-flops to known states, etc.). Each identified feature contributes a positive or negative score to a total testability score.

Other packages make more quantitative measures. Bennetts [32] defines the testability of a circuit node in terms of its controllability and its observability. The controllability measures the ease (or otherwise) of driving the node to a 0 or 1 state from the primary inputs. This takes into account all possible controlling paths to that node. The observability measures the ease (or difficulty) of observing the effect of a fault (stuck at 0 or stuck at 1) on that node by setting up sensitive paths that will propagate the fault effect to the primary outputs. The testability of the node is defined as the product of its controllability and its observability. The measure of system testability is the average node testability.

A number of ad hoc design for testability techniques can be used at either the board or IC design level. Reference [33] gives a good review of many of these techniques, and in addition describes a number of systematic design methodologies that result in greatly improved testability at the cost of a small overhead in circuit area, speed and pin count. These methodologies include the scan path and Level Sensitive Scan Design (LSSD) techniques, as well as a number of Built-in Self-test (BIST) methods.

In the scan path method, testing is simplified by consistently using a special latch element for all flip-flops. In normal system operation these special latches operate as standard, simple latches. However, they have an extra control line which, when active, causes them instead to shift out their current state on a “scan out” line and to shift in a new state from a “scan in” line. The scan in and scan out lines of all latches are chained together to form the scan path. The scan path begins at a primary input and ends at a primary output. The scan path can be used by a tester to serially shift out the state of the device or system and shift in a new state. This simple method increases the controllability and observability of the circuit, reducing the difficult problem of testing sequential logic to the simpler problem of testing combinational logic.

The LSSD method was developed by IBM and is used in their products. It uses the scan path principle combined with a latch design and some additional design constraints which guarantee raceless circuit operation.

There are several BIST techniques, all of which integrate the test pattern generation circuitry and circuit response evaluation on-chip. The test pattern is either provided by a counter, or linear feedback shift registers (LFSRs) that generate a pseudo-random pattern. The circuit response for all of a large number of patterns is captured and compressed into a signature word which is compared against the expected signature (either calculated by simulation or by measuring a known good circuit). The result of the comparison produces a 1 bit “go/no-go” status indicating whether the self test was passed.
The scan path and BIST techniques are combined in the Built-in Logic Block Observer (BILBO) circuit. It is a simple circuit that has two control lines causing it to operate in 4 different modes:

1. As a normal set of latches.
2. As a pseudo-random pattern generator.
3. As a signature generator.
4. As a set of registers connected into a scan path.

The BILBO cell, or scan path latches are available in some ASIC vendor's cell libraries and are supported by their design tools.

4.2.1 Other testing problems

Redundant logic makes it impossible to detect faults in those parts of the circuit because the circuit redundancy masks the effect of the fault at the POs. Unintentional introduction of redundancy in a circuit can be tracked down and eliminated. However, circuits often contain statically redundant logic that is added to prevent a timing problem, and this will limit attainable fault coverage to less than 100 percent. Understanding these effects, mastering design for testability methodologies, and developing the skills for generation of test vectors requires considerable experience.

5. Tools for Board Layout

Modern CAD systems for Printed Circuit Board (PCB) layout typically perform some, or all, of the following functions:

1. Initial assignment of logic functions to packages.
2. Initial placement of component packages (manual or automatic).
3. Placement optimisation in order to improve board routability.
4. Optimisation of package and pin assignments in order to improve board routability.
5. Routing (manual or automatic).
6. Postprocessing router results to clean up traces, minimise use of vias, etc.
7. Production of manufacturing data and documentation.

Each of these steps will be described in more detail below. The major advantages of using advanced PCB layout tools over hand layout methods are:

1. The placement and routing process is driven by a netlist automatically generated from the schematic. It is therefore certain that the board produced will have the connectivity of the designer's schematic. This cannot be guaranteed with a hand driven layout.
2. The layout system ensures that manufacturing design rules are not violated. These can be simple rules such as the minimum pad-to-trace clearance, minimum trace-to-trace clearance, or more complex rules that, for example, forbid trace routing under certain types of component, or restrict the allowed angles at which traces can join to pads, etc. By enforcing a well established set of design rules the CAD system ensures that an optimum manufacturing yield will be obtained.

3. Once the layout itself has been completed, the CAD system can automatically produce a complete set of errorless manufacturing documentation and numerical control tapes for the manufacturing machines (e.g. drill tapes, photoplotter control tapes for generation of artwork films, solder resist masks, interface data for automatic component insertion equipment, or automatic test equipment, etc.).

There are three major classes of PCB CAD system in use:

1. The simplest class are the digitising systems that capture a hand made layout sketch, check it for design rule violations and produce manufacturing data. These systems cannot verify that the board is electrically equivalent to the designer's schematic.

2. The next class are the interactive graphics layout systems, that are driven by a schematic netlist. All layout decisions are made by the layout technician. Like the digitising class, these tools check for design rule violations, produce manufacturing data and documentation, and in addition ensure correct board connectivity by rejecting operator attempts to connect points that are not consistent with the connectivity defined in the netlist.

3. The most powerful class are the schematic driven, interactive layout systems described above, but enhanced with software (or special purpose hardware) for automatic component placement and automatic track routing. The automatic routines speed up the layout task. The best systems allow interactive monitoring and manual intervention in the autoplac and autoroute operations.

We shall only describe class 3 PCB CAD/CAM tools. A typical PCB layout tool will draw information about the design rules from a user written technology file. Data for specific component types will be extracted from a PCB component library. The connectivity to be routed (and possibly layout constraints imposed by the designer) will be available in the design input file, previously generated by compiling and linking the designer's schematics (as described in section 2.1).

5.1 Initial package and pin assignment

The designer's schematic usually employs functional elements (i.e. gates, flip-flops, ALUs, etc.), rather than complete packaged components containing multiple functional elements. The layout system will first have to pack functional elements into specific component packages of the corresponding type, as indicated in Figure 19. This is done by assigning each logic element a package reference designator (e.g. IC24). Functional elements assigned to the same package are differentiated by assigning the elements' ports the physical package pin numbers of one of the elements within the package. Once assignment has been made, the schematic can be automatically back-annotated with the package reference designator and pin number information.
The initial assignment of elements to packages is usually made either pseudo-randomly in the order in which the elements are stored in the design database, or by grouping like elements that are interconnected on the schematic. Normally the designer can control critical assignments by pre-assigning the reference designators and pin numbers on the schematic and "freezing" them before running the initial assignment routines.

5.2 Component Placement

The objective of the placement function is to place components in a way that results in an easily solved routing task. At the same time it must satisfy other (possibly conflicting) considerations such as circuit performance (clock skew, cross-talk, minimal critical delay path, etc.), uniform distribution of heat dissipation during operation in order to avoid hot spots, sufficient clearance for an automatic insertion tool to be used, etc. Reference [34] gives a good overview of placement techniques.

Normally the user can interactively place critical components, or components that must be located in predefined positions (e.g. edge connectors). He can then use the autoplace software to place the remaining components. Components are usually placed on a user defined placement grid. Once critical components are placed interactively they can be "frozen", preventing the placement optimisation software from moving them.

Regular structures like memory arrays, or bus-oriented structures, are best placed by hand. In these cases the best placement is obvious to the human operator, while the autoplace software will consume considerable compute resources only to come up with a less satisfactory placement. Autoplace is best suited to those parts of the design where there is no obvious regular layout topology.

5.2.1 Metrics for placement algorithms

Algorithmic placement uses a metric to evaluate the effectiveness of trial placements (quantified by a score (S)), and subsequently chooses the best trial placement. It will be necessary to perform placement by hand in those cases where the algorithms do not use a metric that takes into account other constraints that are critical to the particular design. Metrics used to evaluate placement effectiveness can be divided into two major classes:
1. Simple metrics that ignore possible interactions between net routings.

2. Metrics that measure the likely interaction between net routings due to congestion of the board. These metrics often give placements of superior quality to those given by the first class.

The commonest metric in the first class is total wire length summed over all nets. Another metric in the first class models pin to pin connections as springs, tries to find the equilibrium positions of all components under the action of the springs, and then moves each component to the nearest available placement grid point. These class 1 metrics reduce the total length of wiring and therefore tend to minimise occupancy of available routing space, which in turn should make the routing problem easier to solve. However, they do nothing to prevent wiring density building up in congested areas, consequently leading to poor routing results.

The second class of metrics include, for example, measures such as the number of nets that cross a "cut" line. The well known rats nest display shown in Figure 20 gives a quick qualitative measure of the effectiveness of a placement by allowing visual identification of potential areas of routing congestion.

![Figure 20: The rats nest display identifies congested areas.](image)

5.2.2 Net-to-wire partitioning

Because placement must position components in a way that optimises board routability, it will be sensitive to the way in which the signal nets are partitioned into wires. Thus placement and net-to-wire partitioning are interwoven problems. Several net partitioning strategies are shown in Figure 21.
In order to reduce the computational complexity of the problem (which in principle requires nets to be repartitioned afresh for each trial placement evaluation), many systems use either the complete interconnection graph, or one particular partitioning for all trial placements. After a placement has been chosen, the nets may be repartitioned into the minimum spanning tree, or minimum length chain, by solving the travelling salesman problem [35]. After partitioning nets into wires, further placement optimisation may be profitable; in general the layout expert iterates between placement optimisation and net partitioning until he is satisfied that the board can be routed.

Some systems allow steiner tree routing, where a track can join onto an existing track belonging to the same net by making a T-junction. It turns out that the minimum spanning tree is already a near optimal partitioning for finding the minimal Steiner tree [35]. The result of the net partitioning step is a wirelist (not netlist) ready to drive the router.

5.2.3 Initial Placement Algorithms

Initial placement is made using the so called constructive placement algorithms, some of which are outlined below:

1. Random initial placement of components is sometimes used. It relies on the iterative placement improvement algorithms to converge to a good placement.

2. The cluster development class of algorithms choose the next component to be placed by selecting the one with the strongest coupling to already placed components. It is placed at the position that gives the best score using one of the metrics described above. Note that it does not use all the interconnect information available because as yet unplaced components do not contribute to the current component placement decision.

3. The min-cut class of algorithms considers all interconnections in parallel by dividing the set of components into two subsets such that the number of interconnections between the two subsets is minimised and the area of components assigned to each
subset is approximately equal. The partitioning process is repeated on the subsets until each partition contains only one component. These algorithms have the advantage of using the interconnection information globally, top-down, deferring local decisions to last. They produce good results but are compute intensive.

4. Another type of global placement method places all components simultaneously by arranging them in a way that minimises the sum over all components of the directed force vectors representing the strength of inter-component coupling (the force is the number of connections between the pair multiplied by the distance between their centres). The components are then moved to the nearest available placement grid point.

5.2.4 Placement Improvement Algorithms

After initial placement the nets will partition into wires in a different sequence. Thus the initial placement will no longer be optimal for the new wiring sequence. Another reason why the initial placement is less than optimal is that components are placed either by taking into account only the information about already placed components, or by ignoring effects of different sized components, or the constraint that all components must fall on the user defined placement grid, etc. Thus initial placement is usually followed by an iterative placement improvement phase in which components are moved to new trial positions and the placement score is re-calculated. If the trial placement shows an improvement it is kept, otherwise the components are returned to their previous positions and a new trial placement is tried.

Some of the algorithms used for placement improvement are briefly described below:

1. The Pairwise Interchange algorithm selects a each component in turn and successively interchanges it with all other components. Each interchange is evaluated by calculating its score using the total wire length metric. The interchange with the lowest score is retained.

2. The Force Directed Interchange method uses the mechanical analogy of attaching a spring between components for each connection between them. The tension in the spring is proportional to the distance between the components. The resultant of adding all the force vectors defines a direction in which the component should move in order to reduce the total wire length. The pairwise interchange method is then applied uniquely with the neighbouring components in the direction of the resultant force.

3. In the Force Directed Relaxation technique, a component is selected and moved to the placement grid position nearest to its "zero force" equilibrium position. Any other component that was occupying this grid position is itself moved to a new zero force position. The process is repeated until the chain of displacements is broken when an unoccupied zero force position is found. The new trial placement is evaluated by calculating its score and is retained if the score improved over the previous placement's score. All components are selected in turn for relaxation.

4. The Simulated Annealing method avoids a problem that is common to all the other optimisation methods described before. All the other methods only accept a trial placement if its score is better than the current placement; they usually do not converge to the global minimum, but get stuck in some local minimum. The simulated annealing algorithm allows the minimisation to climb out of a local minimum by
accepting poorer scoring trial placements with a probability $e^{-\Delta S/T}$ (where $\Delta S$ is the (positive) change in score between the trial placement and the current placement, and $T$ is the "temperature"). Starting from an initial placement and an initial temperature $T_0$, components are moved at random and the temperature is slowly reduced. If the temperature is not reduced too quickly, the placement will gradually move towards the global minimum. Since the process is probabilistic, convergence to the global minimum is only guaranteed as the number of iterations tends to infinity. In practice good results are obtained for a tractable number of iterations, although some claim that they are not superior to those achieved by other methods. Despite relatively long run times, this method is very popular.

5.3 Package and Pin Assignment Optimisation

The initial packaging process described in section 5.1 assigns schematic elements (i.e. gates, flip-flops, etc.) to packages (each package may contain multiple identical elements) and assigns package pin numbers to the ports of the elements. These assignments can be optimised to improve the routability of the board. Elements can be iteratively swapped between placed packages of the same component type, or they can be swapped with other elements within the same package. Permuting logically equivalent pins (e.g. the two input pins of an AND gate) can also improve board routability. Thus, because swapping element or pin assignments results in changes in the interconnection topology, just as it does with package swapping, the assignments are usually optimised using techniques similar to iterative placement optimisation. In fact, placement and assignment decisions are all related and ideally should all be minimised together. In practice, in order to reduce the computational complexity of the problem, they are treated as separate optimisation problems.

As with package placement optimisation, after assignment optimisation the nets will need to be repartitioned into wires (this is not usually done dynamically as part of the optimisation process). Several iterations of assignment optimisation and net partitioning may be necessary before the board is judged to be routable.

5.4 Routing

After placement, assignment optimisation and net partitioning have been iterated several times the designer will start routing the connections. Routing can either be made interactively, or by invoking automatic routing software. Many advanced PCB CAD tools are equipped with several different routing algorithms, some being fast but less effective at finding routes, others being slower but more effective, and yet others being optimised for routing particular topologies efficiently. Results from many of these algorithms can be controlled by defining cost functions, limits or other parameters as described in this section and for example in section 5.4.4 An algorithm can make several passes at the problem, each pass using different cost functions or control parameters. The operation of a routing algorithm can often be restricted to certain areas of the board, chosen so that they contain topologies for which the algorithm is optimised, or for which the pass parameters have been specially chosen. Most autorouters route on a pair of routing layers, using one layer for vertically oriented tracks and the other for horizontally oriented tracks. However, some routers approach a 3-dimensional routing capability by allowing routing on an arbitrary number of layers simultaneously (in practice the number of simultaneous routing layers is limited by run time and memory requirements and is typically not more than 4 or 6).
Critical routes can be autorouted first, or they can be interactively prerouted. Some autorouters run in batch mode and give the user little or no feedback as to how the routing is progressing. More user friendly autorouters display routes graphically as they are found and allow the user to interact with them by interrupting them, changing parameters or modifying their results and then allowing them to continue. These re-entrant routers should give better results and higher productivity than a batch router by combining the speed and accuracy of the computer with the experience and intuition of a layout specialist.

For batch autorouters, the user plans a routing strategy for tackling the particular routing problems of each board, in which he defines the order in which the algorithms will be applied, the routing parameters for each pass, etc. Setting up effective routing strategies requires much previous experience with the autorouter. However, most systems provide default strategies that have been found to give reasonable results on typical designs.

One of the choices systems usually give to the layout technician is on the order in which the router will attempt to make the connections. Typical ordering options are:

1. Short connections first.
2. Long connections first.
3. Connections from a selected component, or group, first.
4. Connections to components selected by order of their placement:
   a. from left to right
   b. from right to left
   c. from top to bottom
   d. from bottom to top
5. By order of local estimated routing density (route to "surrounded" pins before they are boxed in by other tracks).
6. Autorouting one net at a time (using interactive graphics to select a net to be "one shot" routed).
7. etc.

Ordering strategies can effect routing results, but there appears to be no preferred ordering that is good for all types of layout. The main types of routing algorithm encountered in PCB CAD systems are discussed next.

5.4.1 Pattern Routers

The pattern router is used to route regular topologies of the kind found in memory arrays or bus structures, as shown in Figure 22.
The router stores a certain number of standard routing templates suitable for applying in these cases. The list of templates is searched until one is found that matches the point to point connection to be made and bypasses any intervening obstructions. This algorithm is very fast. However, there is relatively little advantage in using this type of algorithm when the interactive routing software allows step and repeat operations. In step and repeat methods, the pattern is manually routed once and can then be copied many times over between different pin pairs (by selecting a new start point to repeat the pattern with a simple click of the mouse button). Arrays can be interactively routed even quicker when step and repeat can be applied with a group of tracks (e.g. to replicate all the connections between two ICs of the array in one operation).

5.4.2 Line Probe Routers

The line probe router shown in Figure 23 sends out a probe line from the source pin (S) along the longest side of the smallest rectangle enclosing the source and target (T) pins. The probe line either reaches the far side of the rectangle, or it encounters an obstacle (e.g. a pad, a previously routed track, a user-defined “keep out” area, etc.). It then proceeds by selecting one of the following options:

1. Add a via and continue on another layer.
2. Change direction on the current layer.
3. Backtrack and probe in another direction.

This strategy is applied repeatedly until the probe hits the target pin, or is terminated by some other criteria (too many bends or vias, CPU time allotted to each connection search exceeded, etc.). Each of the possible actions that can be taken at a decision point can be assigned a priority (either hard wired in the code, or under user control).

There are many variations on this basic algorithm (for example allowing the probe line to overshoot the target pin in order to find possible paths that approach the target from behind, or starting the search simultaneously from both pins and continuing until the two probe lines intersect.

The quality of the results will depend on the way the possible actions are prioritised at the search decision points. The line probe algorithm executes rapidly because checking for intersection with
obstructions can be accelerated by using a binary search technique on lists ordered by co-ordinate values [35]. The major shortcoming with this class of algorithms is that the decisions made at each point where the probe line is blocked are based on a set of predetermined rules; there is no comparison of possible alternatives with an eventual choice of the best candidate. The algorithm tends to produce routes that unnecessarily obstruct routing channels and box in other pins.

5.4.3 Maze Routers

In this approach, originally due to Lee [36], each routing layer of the board is divided into a large number of cells, as shown in Figure 24. Cells are marked as "occupied" at those positions where there are plated-through holes for component pins, tracks that have already been routed, or user-defined "keep out" areas. The simple maze search algorithm then searches for a path through the maze in two steps:

1. Starting from the source pin (S), adjacent, unoccupied cells in the vertical or horizontal direction are provisionally marked as being occupied by the current search. This step is then repeated using each cell that was marked in the previous step as a source for the second step. By repeating the operation a 'wave front' is made to flood through all accessible parts of the maze. Each newly marked cell is stamped with the iteration number of the wave front expansion process. The shortest possible path is found when the wave front first reaches the target pin. However this path is not necessarily the "best" route because length is not the only criteria for judging its quality.

2. The router now enters a back tracing phase, in which it traces a path backwards from the target pin to the source pin, laying down copper as it does so and marking the corresponding cells as definitively occupied. Backtracing is made by looking for an adjacent cell having a search iteration number one less than that of the current cell. When more than one candidate is available, a choice is made by applying simple rules (e.g. continue in the same direction as the last step).

The maze router can be adapted to find diagonal tracks by allowing the wave front to expand to diagonally adjacent neighbours. It can be applied to routing on several layers simultaneously by defining a 3-dimensional maze and allowing the wave front to expand to neighbouring cells in the third dimen-
Figure 24: The Maze search algorithm floods a wave front from source to target pin.

Wave expansion is first made from the source to the target pin. In the example the target is reached on the 12th step. The track is laid down by retracing from the target using simple rules. In the example, track A is found by starting the retracing downwards and changing direction only when blocked. Track B would be found if the retracing made its first step to the left instead of downwards.

sion in order to explore paths which change from one layer to another.

Search times can be very long, depending on the search area and the size of the cells. Search times can be reduced by starting the search from the pin that is furthest from the centre of the search area (wave front expansion will be cutoff by the search area boundaries in some directions sooner than when the wave front starts from the centre of the area). Another technique starts the search simultaneously from both pins and stops when the two wave fronts collide. Yet another technique limits the search area to the smallest rectangle enclosing the source and target pins, and only increases the search area when a path cannot be found within the rectangle. Execution times may be reduced and routing completion rates increased by allowing the search to connect to any previously routed track belonging to the same net. Steiner tree routing is simply implemented by defining all cells on the previously routed track(s) as targets to be searched for in parallel. The first target hit makes the shortest possible T-junction connection.

Although the simple maze search algorithm will find a path if one exists, results are unsatisfactory either because they include too many bends, or because horizontal track segments block vertical routing channels (and vice-a-versa), or because too many vias are inserted to change layer (each via blocks routing channels on all layers of a multi-layer board and increases board manufacturing costs).

5.4.4 Costed Maze Routers

The costed maze search algorithm is a development of the simple maze search algorithm due to Rubin [37]. Each step in the wave front expansion process adds a cost increment to a running cost that is propagated from cell to cell. Wave front expansion is only done from the cell on the wave front that has the current minimum cost. If several wave front cells have the same minimum cost, they are
expanded in reverse order (i.e. the last one that was assigned a cost is expanded first). In this way the wave front spreads out quickest in directions of lowest cost gradient, and the minimum cost (but not necessarily the shortest) path reaches the target first. In order to facilitate back tracing of the path, each cell remembers the direction from which the wave front entered it.

By assigning a small cost increment for steps in one direction and a large cost increment for steps in the orthogonal direction one can bias wiring on a given layer to run in a preferred direction. A pair of layers will be assigned orthogonal preferred wiring directions and routed simultaneously, with paths changing from one layer to the other when a change of direction is necessary. In this way channel blocking by wires running orthogonal to the preferred wiring direction is eliminated. By balancing the cost of changing layer against the cost of stepping orthogonal to the preferred wiring direction, one can trade the number of routing channels lost through inserting a via against the number lost by allowing short wire lengths orthogonal to a layer’s preferred wiring direction.

Most commercially available systems use a costed maze search algorithm because of its flexibility. They offer a large number of cost factors that can be set by the user in order to control routing results. When the cost factors are well chosen, the costed maze router gives better results than the line probe class of algorithms. Note that, by appropriately choosing the cell sizes, track-to-track and track-to-pad clearances are automatically guaranteed and no design rule checking is required after autorouting. Runtimes can be long (typically tens of hours of CPU time for large, dense boards with a fine cell grid). The host machine will also require a large real memory (typically in excess of 8MB) in order to accommodate the cell array without catastrophic loss of performance due to virtual memory paging to/from mass storage. As with compute intensive simulation tasks, attempts have been made to construct special purpose hardware for acceleration of maze routing algorithms (see for example references [38] and [39]).

5.4.5 Completing Routing by Rip-up and Re-route

Autorouters usually lay down new paths without taking into account their effect on the routability of connections still to be routed. Once routing channel occupancy has risen to the point where router success rates start to drop off dramatically, it may be worthwhile reviewing previous routes and perhaps modifying them in order to open up channels for routing the remaining connections. Figure 25 shows an example where the blocked route AB can be routed after the blocking route CD is re-routed along an alternative path.

Figure 25: Rip-up and re-route of track CD allows AB to be routed.
The rip-up and re-route process can be performed interactively by the layout technician, or semi-automatically with a re-entrant router by manually moving the obstructing route CD and then invoking the autorouter to "one-shot" route the connection AB. Some systems offer automatic rip up and re-route algorithms that identify blocking tracks, rip them up and re-route them along a different path by using different cost functions than those used in the initial router pass. Blocking tracks are ripped up and re-routed until the blocked track can be successfully routed. Sometimes the ripped up track cannot be re-routed and the rip-up and re-route algorithm is called recursively in order to route the ripped up blocking track. Recursive rip up and re-route does not always converge, so one can end up increasing the number of unfinished routes.

Batch routers that use multi-pass rip-up and re-route techniques, use the results of one pass as input to the next pass. Each pass reworks the previous one using different routing parameter values. Although individual passes may rip up more tracks than they re-route, the tendency is to converge. Run times for large boards are measured in days. If the process does not achieve 100 percent routing success, it can be very difficult to route the few remaining connections by hand.

An alternative approach to completing the routes left over by the autorouter is to let the autorouter route tracks as close to the target pin as it can get. It may be easier for the operator to interactively finish the dangling track than to route the complete connection. If this proves to be too difficult the operator may try modifying component placement and re-routing the board. A third option for squeezing in the remaining tracks is to re-route the layer using finer tracks and smaller clearances so that 2 or 3 tracks can be routed between pins of an IC instead of 1 or 2 tracks. If all else fails, extra routing layers must be used.

5.4.6 Choice of Routing Grid

A consequence of developments in packaging technology is that many board designs now contain a mixture of packages with different lead spacings\(^7\). In these cases it is difficult to find a uniform grid on which all component pins fall, and which guarantees sufficient clearance between tracks and pins in all cases and at the same time does not result in the router using too much memory or processor time.

Figure 26 shows how the use of non-uniform grids allows high density routing (2 or 3 tracks between pins with 100 mil centre-to-centre spacing) without the memory space and processing time penalties that would be incurred by using a uniform high density grid.

The variable grid technique uses different grid sizes in different parts of the board, each grid being chosen optimally for the local routing problem. Other systems can switch from the normal grid to a finer routing grid whenever the router is working near an off-grid pin.

Gridless routers which check design rules (clearances with neighbouring tracks, pads etc.) as they route have also been tried. However, the many complex geometry calculations involved cause them to be very compute intensive and slow in practice. In addition the absence of a grid makes interactive routing without violating design rules very difficult.

---

\(^7\) e.g. dual in-line packages (DIPs) usually have 100 mil lead spacing, whereas surface mounted devices (SMDs) usually have 50 mil spacing and some connectors have metric pin spacings.
5.5 Postprocessing, Manufacturing Data and Documentation

Once the board has been placed and routed there follows a postprocessing step that cleans up the results of the autorouter in order to improve board manufacturing yield or reduce manufacturing costs. This is either done by a batch program, interactively, or by using a mixture of batch and interactive processing. Typical postprocessing actions are:

1. Replace "staircase" routing by smooth diagonal tracks.
2. Move tracks to increase clearances between tracks or between tracks and pads.
3. Suppress unnecessary vias (drilling vias is one of the most expensive operations in manufacturing a board; they are also often the source of manufacturing defects).

Often a final pass through a design rule checking program is made in order to catch any design rule violations introduced in postprocessing (or by bugs in the autorouter). The next step is to produce data and documentation for manufacturing:

1. Numerical control (NC) tapes for drilling machines.
2. NC tapes for photoplotting:
   a. Artwork
   b. Solder resist masks
   c. Silk-screen printing, etc.
3. Data for driving Automatic Component Insertion Equipment.
4. Data for set up of board test equipment.
5. Pen plots, parts lists, etc.
The production of NC tapes is done by programs that first optimise the sequence of machine operations in order to minimise machine operation time and wear, and then produce data in the required format. In principle all the information exists in the design database, thus the documentation can be produced with little manual effort and without errors.

Finally, if gate or pin swapping were made during the layout of the board, the changed package reference identifiers and pin numbers must be back-annotated to the logic schematics. The better integrated schematic capture and layout systems automate the back-annotation step, but many of the cheaper systems leave this job to be done manually. One disadvantage of working with schematic capture and layout systems from different vendors is that in many cases there is no support for automatic back-annotation.

5.6 Software for other board technologies

Although printed circuit board technology is used for the production of the vast majority of boards, there are other board manufacturing technologies that have their place. The wire wrap technique is used at CERN where large dense boards are prototyped or only made in very small quantities. As the wire wrap technique uses a machine to make point to point connections using insulated wire there is no difficult routing problem to be solved. The NC tape for driving the wire wrap machine is produced by a relatively simple netlist driven program [22]. Although printed circuit board manufacturing costs are lower, for very small volumes the overall cost (layout design and manufacturing costs) is smaller for the wire wrap technique. The wire wrap technique has the advantage that rewiring during prototype debugging is very quick, easy and reliable.

The Multiwire technology uses insulated wire that is routed between obstructing pins and laid down by a special machine. Component placement, routing, analysis and production of manufacturing and test data are supported by a special Multiwire design software package [40]. As the insulated wires can cross over each other no vias are required for changing from one wiring layer to another layer. These two characteristics mean that a layer of Multiwire routing can be much denser than a typical layer of printed circuit board routing. Like the wire wrap technique, Multiwire technology is suitable for small volumes. The high density wiring also minimises the total number of wiring layers and, in the case of controled impedance ECL designs (where each layer has to be spaced at a certain distance from a ground plane in order to provide the correct transmission line impedance) helps to contain the finished board thickness within allowed limits.

5.7 Practical experience with Board Layout CAD

At CERN we have experience with two different types of PCB layout system; one uses a re-entrant autorouter providing powerful interaction and guidance possibilities, the other uses a multi-pass, rip-up and re-route, batch autorouter. Users of both types of system agree that a good component placement is the key to successful routing. The placement task is approached in a similar fashion by users of both types of system. A typical placement strategy is as follows:

1. Place and lock in place connectors and other critical components.
2. Automatic initial placement of the remaining components.
3. Interactive adjustment of placement in critical areas (e.g. near connectors or around busses) to improve routability. The interactive use of the rats nest display or force
*vectors* display gives dynamic, real-time, graphical feedback on the placement adjustments.

4. The adjusted components are locked in place, and automatic placement optimisation is used to improve the placement of the remaining components. One or more further iterations through steps 3 and 4 may be made.

5. Obvious candidates for manual gate swapping are made on the basis of the rats nest display.

6. The "well swapped" gates are locked and optimisation by automatic gate swapping is invoked.

7. Optimisation by pin swapping is made manually and/or automatically.

8. Steps 5 through 7 are repeated one or more times if required.

The job is now ready for routing. In the case of the re-entrant router a typical strategy is:

1. Decide on the number of routing layers for the job. A rule of thumb allows this to be predicted from the average number of pins per unit area and the board size.

2. Route power and ground connections by hand.

3. Manually route the most critical areas (near connectors or in bus structures).

4. Use the autorouter in re-entrant mode to route the remaining critical areas with operator guidance and interaction where necessary.

5. Manual clean up of tracks (shorten long tracks, display tracks that are likely to block the remaining connections from being routed).

6. Batch autoroute the remainder of the board using multiple passes with each pass gradually relaxing the constraints used in the previous pass.

If only a small number of connections fail to be routed, the layout technician will interactively finish the routing; otherwise he chooses one of the following options:

1. Review the placement and re-route the board.

2. Unroute part of the board and re-route it using a finer grid.

3. Add an extra pair of routing layers to the job.

The strategy is that the technician recognises regular structure in the layout and is better able to place and route those parts manually with assistance from the machine, while on the other hand the machine is more efficient in optimising layout in those areas with little regularity of structure.

---

8 assistance in the form of quick graphical feedback of changes, graphical presentation of several possible routes found by the autorouter for final selection by the technician, on-line design rule checking etc.
In the case of the multi-pass, rip-up and re-route batch autorouter a batch job is submitted to a dedicated routing server. This may run for several hours, overnight, or even for several days. During the batch autorouting process the operator works on other jobs. If the batch autorouter does not achieve 100 per cent routing completion it will be extremely difficult for the technician to interactively finish the routing (the powerful rip-up-and-re-route algorithm leaves very little routing space unused); in this case the technician will choose one of the following options:

1. Improve the placement and try to re-route the board.
2. Route with different design rules.
3. Add an extra pair of routing layers to the job.

In our experience autorouters are useful for non-critical digital TTL designs with random structured layouts. Their results are easily surpassed in quality by an experienced human operator whenever the layout problem contains some degree of regularity. Furthermore, they do not (yet) give satisfactory results for high speed ECL layouts where transmission line characteristics are important. They are also unsuitable for most analogue designs. However, the autorouter is only one aspect of the layout system, and although many jobs may not make much use of the autorouter, they do benefit from other aspects of the CAD system.

One should not forget that layout systems are very sophisticated and complex. Installation of a new system will be followed by a fairly long period where technicians learn to use it proficiently (especially if they have no previous experience with CAD systems) and it is tailored to the users' environment by capturing in technology definition files the local design rules and other technology related parameters reflecting local design practices. Component libraries have to be built up, or adapted to local standards.

Another area that usually requires considerable effort to set up initially is the manufacturing interface. The plethora of incompatible machinery in use world-wide requires the CAD vendor to support interfaces to many different brands of machinery. Inevitably the quality of the average interface package suffers; they are often user-unfriendly, error prone to operate, and contain residual bugs that need to be identified and corrected before a smoothly operating manufacturing interface is established.

Users that mix schematic capture and layout systems from different vendors will probably encounter a number of additional problems related to the fact that the two packages work from different component libraries. This is especially true when the interface supports automatic back annotation. The component library data will need to be modified so that all component data (e.g. gate and pin swapping rules) are compatible between the two sets of libraries. In order to preserve library compatibility, guidelines for creation of new library components will need to be worked out and put into practice. Ideally the responsibility for library development and maintenance should be concentrated into the hands of a library manager or small group of experts.

A large number of problems have to be solved initially and important organisational and management changes will need to be carried out. For this reason it typically takes between 6 and 12 months before a new system comes up to full productivity.

---

9 Examples of other benefits of layout CAD systems are: The autoplace and interactive placement functions, connectivity driven layout eliminating wiring errors, design rule checking ensuring manufacturability, automatic production of error-free manufacturing data and documentation, ease of modifications to the design, etc.

Lack of space precludes a detailed discussion of layout tools for integrated circuits. Reference [2] can be consulted for more details. The semi-custom IC layout problem is in principle similar to the PCB layout problem, but nevertheless differs from it in several important aspects.

Figure 27 shows three common architectures used for semi-custom ASIC layouts. In semi-custom gate array design the macros blocks (the logical building blocks, or components, supplied in the vendor’s library) can be placed only at predefined positions corresponding to the underlying prediffused, uncommitted transistor array. The gate-array is customised by designing metal layer masks that define the inter-transistor connections that turn an uncommitted array of transistors into a specific functional macro block (this pattern is supplied in the vendor’s macro library) and at the same time define the metal interconnect paths between the macro blocks. In the case of standard cell designs prediffused wafers are not used, so cells (i.e. pre-designed functional blocks) can be placed with greater, but not complete, freedom. Usually all the cells have the same height and have to be laid out in rows as shown in Figure 27(a), each cell abutting its neighbour, and sometimes alternate rows having their cells mirrored about the row axis with respect to the neighbouring rows.

For channeled gate arrays (Figure 27(b)) routing is restricted to run in channels of fixed width. For standard cell designs the channel width is determined by the user or automatically by the autorouting software, the cell rows being placed so that sufficient routing space is available to guarantee the successful routing of the design. Small and medium sized gate arrays use 1 or 2 (sometimes 3) metal layers for routing, while standard cell layouts use a metal layer to route along the length of the chan-

---

10 When these layout constraints are observed, power is automatically distributed to all cells in the row without need for explicit power routing, and the danger of CMOS latch up is avoided.
nel, and route across the channel on the polysilicon layer\textsuperscript{11}. Polysilicon has a much higher resistance than metal, so routing lengths in polysilicon must be kept as small as possible. The placement optimisation and autorouting algorithms are optimised for solving the channel routing problem efficiently.

The relative slowness of global communication on silicon (see section 1.4) makes good placement essential. A class of software tools known as floor planners can be used to approach the initial placement problem hierarchically. First the major functional blocks are placed and their shapes modified in a way that optimises connections between blocks. Each functional block can then be laid out internally, using a hierarchy of one or more placement steps.

The existence of fixed routing channels in gate arrays leads some systems to adopt a hierarchical approach to routing. Connections are first assigned to run in specific channels, then the assignment of connections to channels is optimised globally before the detailed routing within channels is attempted. Global routing prevents inefficient use of channels and increases the effective gate utilisation (the fraction of available gates that can actually be used to implement logic and yet still be routed with the fixed routing channel resources).

In general one cannot ignore the effect on system performance of the capacitive load (and for polysilicon the resistance) of the interconnections. Most layout suites therefore include an extractor that calculates the capacitance (and sometimes the resistance) of each interconnection, and feeds this data back into the logic simulator as an effective extra delay. Post-layout simulation gives a more realistic estimate of system performance.

For large gate arrays the restrictions of the channeled array architecture make layout very difficult. For this reason the large arrays have abandoned the channeled architecture in favour of the so called sea-of-gates architecture shown in Figure 27(c). This consists of a uniform array of prediffused cells with no space explicitly reserved for routing. Cells can be used for implementing logic, or they can be "sacrificed" for routing.

Layout generators exist for the automatic generation of specific types of functional module (e.g. PLA generators, RAM generators, etc.). Silicon compilers exist for the generation of layouts from high level descriptions, but these topics are outside the scope of this paper (see reference [1]).

7. Conclusions

Software tools for assisting the design process are now powerful, user-friendly and even becoming affordable. In some areas they are already mandatory (e.g. the moderately priced and quite sophisticated packages available on personal computers for the design of programmable logic devices, or logic cell arrays are effectively the only path to the use of these important devices). In the future CAE and CAD will be the key to the use of the most advanced technologies for applications in high energy physics.

On the other hand there are still a number of areas that are not well served by existing tools. In addition their efficient use requires some changes in design methods, work styles and management techniques. Reference [41] gives an excellent discussion of these issues as they apply to the electronics

\textsuperscript{11} The polysilicon layer cannot be used to route over a cell because it is used inside the cell to build transistors. Connections that must pass over a cell can do so by using a special feed-through cell explicitly placed in the row by the designer.
activities characteristic of high energy physics laboratories. We can however be quite confident that the rapid pace of improvement in design tools and methods will continue into the future, and that most of the existing gaps will be soon filled.

8. Acknowledgements

I am grateful to the many colleagues who have contributed to the environment that stimulated the acquisition of the information presented in this paper and have shared with me their practical experience with the tools. Special thanks go to Hans Anders, Francois Bourgeois, Serge Broecker, Endre Futo, Burkhard Heck, Joop Joosten, Bert van Koningenfeld, Sandro Marchioro, Alain Monfort, Paulo Moreira, Ludwig Pregernig, Karl Zumbrock and Rudi Zurbuchen.

9. References


15. SABER Reference Manual, Analogy Inc., P.O. Box 1669, Beaverton, OR 97075-1669, USA.


24. F. Bourgeois, EF Division, CERN, private communication.


40. PCK Technology Division, Kollmorgen Corporation, the MDS and DWIRS Multiwire design software.