# Transcending Static Deployment of Circuits: Dynamic Run-Time Systems and Mobile Hardware Processes for FPGAs

**A THESIS SUBMITTED TO**
**THE UNIVERSITY OF KENT AT CANTERBURY**
**IN THE SUBJECT OF COMPUTER SCIENCE**
**FOR THE DEGREE**
**OF DOCTOR OF PHILOSOPHY.**

By
Ralph Moseley

September 2002

# Abstract

The usefulness of reconfigurable hardware has been shown in research and commercial applications. Unquestionably, this has and will lead to, unique avenues of thought within computer science being explored. The interest by researchers in some specific areas has led to manufacturers developing devices which were enhanced in their ability to dynamically be configured within a run-time context. These improvements are on-going and rapid progress is being made, producing high density, system-on-a-chip capable devices, with fast run-time reconfiguration.

The advancements in this technology have particularly led to a convergence between software and hardware domains, in the sense that algorithms can be implemented in either; the choice being dependent only in terms of efficiency within the medium itself. Older methods for development with these devices have become rapidly dated and inflexible. Very few suitable tools exist, for example, which are capable of fully utilising the inherent capabilities of such hardware.

The approach taken here allows the division between hardware and software to be diminished. Component designs, which may be hardware description language (HDL) based or synthesised algorithms, become easily manipulated and interacted with through a run-time engine, that can deploy elements at will to local or distributed devices. Such entities are essentially hybrid in nature, possessing both hardware and software functionality. These processes are sufficiently self-supporting, to be capable of being used outside of the run-time system within a normal Java development environment.

This work explores how hardware entities can become as dynamic as memory based algorithms within conventional von Neumann based systems, providing the means for extending the software programming paradigm into such areas. It analyses the possibilities of applying object-oriented and occam/Communicating Sequential Processes (CSP) based concurrency philosophies to these highly mobile, hardware processes.

# Contents

# List of Tables

# List of Figures

## Trademarks

# Acknowledgements

Thanks to Xilinx for producing the Virtex hardware and most of all to the JBits team who have been very helpful in my endeavours. Thanks also to my supervisor, David Wood and the University of Kent at Canterbury Computing Laboratory, for supporting this project.

# 1. Introduction

The versatility of field programmable gate array (FPGA) technology has led to its widespread use in consumer goods and appliances, as well as industry. Its use spans from communications equipment and computer cards, through to space and military applications. Research utilising FPGAs has flourished in many avenues, ranging from evolutionary computing [44], fault tolerant systems, to internet reconfigurability.

FPGAs have greatly improved over past years. This has been in both capacity and functionality. Capacity, for example, now stands at around 10 million gates, with additional block RAM also included. Internal clocks within such devices can now be as high as 500 MHz.

One of the main features of FPGA technology, its ability to be totally reconfigured over its life cycle many times, has also improved. Certain FPGAs can be reconfigured dynamically, that is, areas of the device can be changed while currently running, without disturbing any adjacent circuitry.

Hardware engineering design previously was a static venture, concerned with building in module form, prior to a single download. This modular format could take in many different approaches, from graphical tools, through to languages such as VHDL and Verilog. More recently, behavioural languages have become present, such as Handel-C. All these methods of development can be used singularly or together, forming one design which is synthesised and finally compiled down to a single bitstream[1] for download to FPGA.

Software engineering paradigms are being applied to hardware design in this area. Behavioural languages, as mentioned above, relate more to a software development cycle and the actual language itself is aimed at a software engineer's viewpoint and experience. There is a convergence of fields, both at design and run-time.

The reconfigurability of the FPGA and the ubiquity of the internet, has led to devices being able to be reprogrammed, while deployed in the field. A piece of hardware can thus be updated to the latest specifications or changes made while the customer is actually still using the equipment. Yet another use is the remote configuration of equipment which cannot be reached by a human engineer, in hostile or far off locations, such as deep space or in orbiting satellites. For extreme environments, hardened versions of FPGAs exist, resilient to electromagnetic radiation, which may corrupt configuration data.

There is a slow progression to a much more dynamic approach to hardware development, which captures much of what has been learnt in software engineering. FPGA technology allows ever greater expansion along this route, with its static based memory, which can be updated in real time – analogous to computer memory that can be accessed at any given point[2].

The work related here focuses on improving the flexibility and manipulation of programming FPGAs at a high-level and introduces ideas concerning mobility of design components.

This thesis is divided into eight sections. Section 1 provides a broad introduction to the areas discussed and an overview of concepts, finally framing the problem untaken. Section 2 reviews previous work in this area, giving examples and the extent of other researcher's aspirations, ending with an analysis of the material. The third section

[1] The name given to the single binary file which makes up a configurable image in a FPGA.

[2] A few comprehensive surveys exist of reconfigurable technology [7, 10] and possible futures [25].

extends the analysis, providing an exploration and possible philosophy of approach towards a solution. Section 4 provides detail of the implementation, together with introductory material on less known areas. A user perspective of the system is offered in section 5, together with a tutorial and examples. In section 6 there is assessment and evaluation of the system, providing feedback on performance. Future possibilities are then drawn from this, including possible improvements. Finally, section 8 places the results in context with the initial goals for analysis and conclusion. Other material in the form of a glossary and appendices are added for support of the main work and the reader's understanding.

## 1.1 Hardware Description Languages

This section serves as an introduction to the broad areas involved in this work, that is, the design process, particularly the use of languages and the reconfigurable hardware itself.

### 1.1.1 Introduction

A HDL is a programming language used to model the intended operation of a piece of hardware. There are two aspects of the description of hardware that the language can facilitate; abstract behaviour and hardware structure modelling. Behaviour modelling provides a way of simulating hardware without necessarily being structurally descriptive. Hardware structure modelling represents an accurate model of the physical architecture. The behaviour of hardware may be visualised at various levels of abstraction. Higher level models describe the operation of hardware abstractly, while lower levels include more detail, such as inferred hardware structure.

### 1.1.2 History

The US government was the main motive force behind the development of VHDL [3, 42]. In 1980 the decision was made to try and make circuit design self-documenting, following a common design methodology and be reusable with new technologies. There was a need for a standard programming language for describing the function and structure of digital circuits for the design of integrated circuits (IC's). The US Department of Defence (DoD) funded a project under the Very High Speed Integrated

Circuit (VHSIC) program to create a standard HDL. The result was VHSIC hardware description language or VHDL. It was ratified by the Institute of Electrical and Electronic Engineers (IEEE) in 1987 as IEEE Standard 1076.

The F-22 advanced tactical fighter aircraft was the first major program to make use of VHDL descriptions for all electronic subsystems in the project. Different subcontractors designed various subsystems and so the interfaces between them were crucial and tightly coupled. The VHDL code was self-documenting and formed the basis of the top-down strategy.

Another popular HDL is Verilog, developed in 1983 by a Computer Aided Engineering (CAE) company called Gateway Design Automation. This has had some commercial success. For example, of all the designs submitted to ASIC foundries in 1993, 85% were designed and submitted using Verilog[3]. In December 1995 the Verilog language was reviewed and adopted by the IEEE as IEEE standard 1364.

There is very little difference, except perhaps stylistically, between the two languages. The choice of using either usually comes down to personal preference and the availability of tools to do a specific task – such as modelling, or simulating the chosen language.

Yet another HDL is ABEL (Advanced Boolean Equation Language)[4], which was invented to allow designers to specify logic functions for realisation in programmable logic devices (PLD). ABEL is modular in form and is device independent, that is, the target could be any device. Originally, this would have been small scale PLDs but now larger devices can be programmed through other methods i.e. through fuse patterns.

More recently, behavioural languages based on concurrent versions of C have been introduced for design. The basic idea being to describe the functionality, rather than the structure of a circuit. This more closely resembles the software design process

[3] Source EE Times.

[4] ABEL is a trademark of Data I/O Corporation.

than hardware engineering, although usually it is necessary to include some structural references or physical properties within such code.

Software used to assist in the design of hardware come under one of two categories: Computer Aided Design (CAD) and Computer Aided Engineering (CAE). Tools used to design circuit board related hardware come under the category of CAD, while others used for chip design come under the category of CAE. However, the distinction is not that fine, as a simulator could be used to model both boards and ICs.

### 1.1.3 Design Methodology

The basis of electronic design is essentially top down [43]. Ideally this would mean describing a complete system at an abstract level, using a HDL and automated tools, for example, partitioners and synthesisers. This would create the design in implementation on PCBs or multi-chip modules (MCM) which contain standard ICs, ASICS, FPGA, PLDs and full-custom ICs. This dream has not been fulfilled totally but is rather the goal to which EDA tools strive.

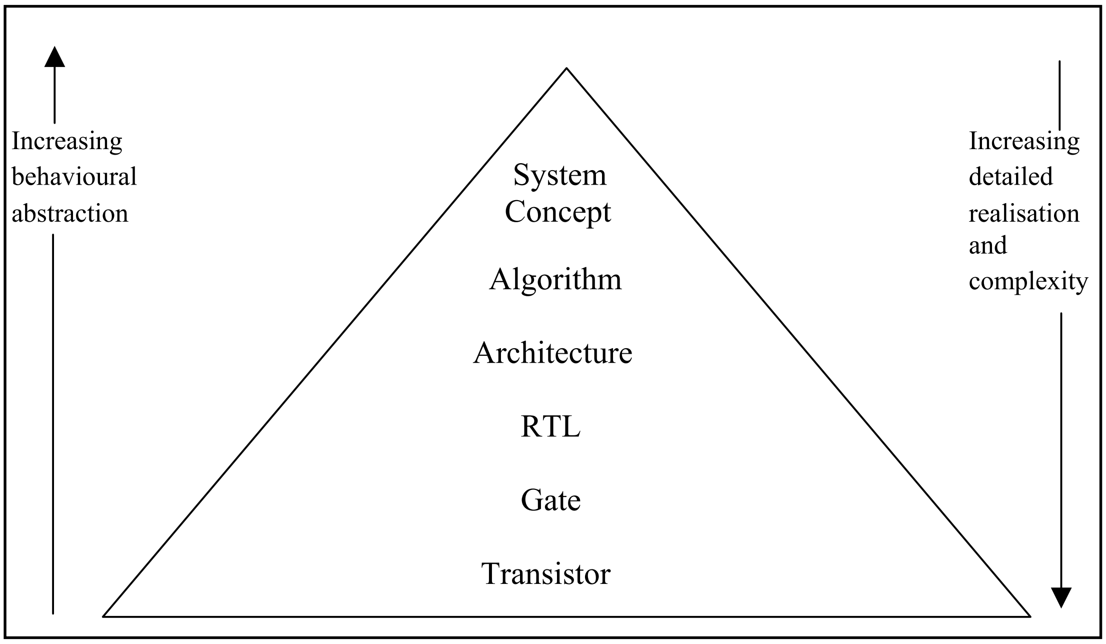

Increasing behavioural abstraction
System Concept
Algorithm
Architecture
RTL
Gate
Transistor
Increasing detailed realisation and complexity

**Figure 1. Behavioural level of abstraction**

### 1.1.4 Simulation

Simulation is a fundamental and essential part of the design process for all kinds of electronic based products. For ASIC and FPGA devices, simulation is the process of verifying the functional characteristics at any level of behaviour, that is, from high-levels of abstraction to low-levels [33]. The basic format of a simulation is:

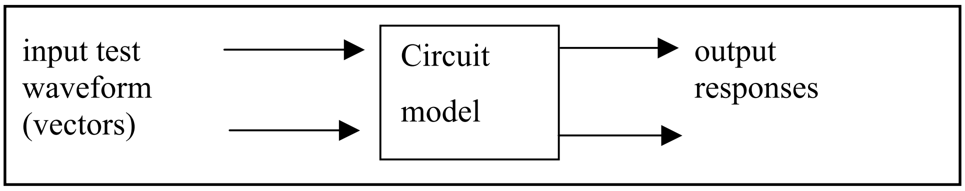


**Figure 2. Black box model**

Simulators use the timing defined in an HDL model before synthesis, or the timing from the cells of the target technology library after synthesis. A simulator can model only the basic behaviour of a circuit, or be a detailed dynamic timing analysis, or both. Static timing analysis is used during optimisation, extracting delays from the cells of the technology library but has several difficulties, such as multiple clocks, complex clocking schemes, asynchronous circuits, transparent latches and identifying and ignoring false paths.

There is a particular type of simulation known as fault simulation, which has particular input stimuli (vectors) and typical manufacturing faults injected into the model. The idea here is to:

- Identify areas of a circuit that are not being functionally tested by the functional test vectors.

- Check the quality of test vectors and their ability to detect potential manufacturing defects.

- To perform board and in-circuit chip testing for both production and repair testing.

This area has become particularly important for several reasons, such as the vast increase in the number of gates on a chip, the increased gate to pin ratio and the reduced timing of sub-micron transistor technology. Early fault simulation has, because of these factors, the ability to reduce costs considerably.

### 1.1.5 Register Transfer Level Synthesis

Register Transfer Level Synthesis is the process of translating a register transfer model of hardware, written in a HDL at the register transfer level, into an optimised technology-specific gate level implementation. A RTL synthesis tool automates this part of ASIC and FPGA design process. Synthesis is by far the quickest and most effective means of designing and generating circuits.

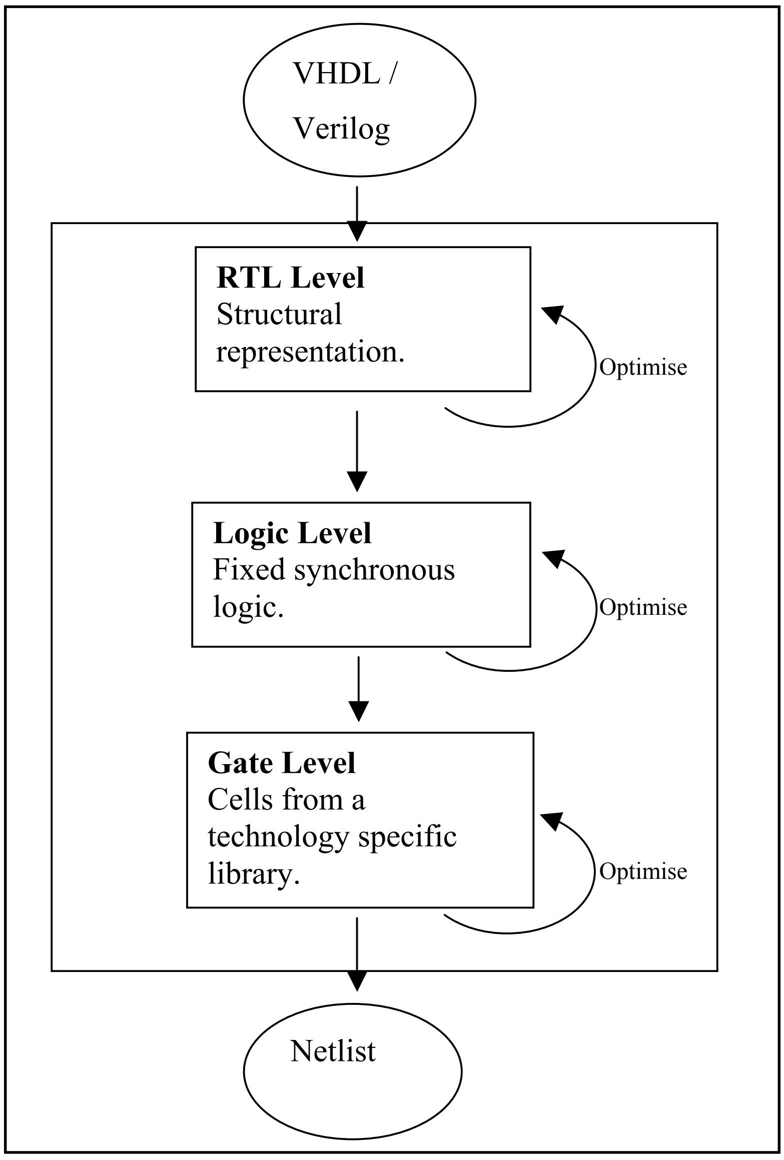


**Figure 3. VHDL to netlist process**

### 1.1.6 HDL Support for synthesis

Certain constructs in a HDL are either ignored or are not supported by synthesis tools. The reason here is that some constructs have no direct hardware correlation, or the hardware intent is extremely abstract. For example, timing related constructs are ignored, as the timing should come from the cells of the technology specific library. Constructs that are not supported typically include; floating point arithmetic operators, loop statements without a globally static range and file manipulation related constructs. What is and what is not supported varies between vendors, although a VHDL working group has been set up to formalise an industry standard subset of constructs that should be supported by synthesis tools, with the intention of making designs portable.

## 1.2 Reconfigurable Systems

### 1.2.1 Introduction

The boundary that once existed between hardware and software has drawn rapidly smaller. This is not only true of our expectations in terms of execution speed but also the ability of what was once solid state physical circuitry to become mouldable to the requirements of the software and the problem domain itself. It would seem that the hardware is becoming as flexible and as expressive as the underlying software, determined only by the constraints of the description language and development tools.

These reconfigurable systems are based on programmable logic, such as the recent large capacity FPGAs, the newly available analogue equivalent – the Field Programmable Analogue Array (FPAA) and to a lesser extent, application specific integrated circuits (ASIC).

### 1.2.2 History

These devices are capable of being programmed to perform any possible logic function. They actually evolved from the programmable logic array (PLA) devices of the early 1970s. The basic structure of the PLA was a matrix of AND, OR and inverter gates, together with a series of programmable switch arrays, which allowed connections to be formed through the chip. A simplified version was also produced, called programmable array logic (PAL), which cut down on the number of gates needed to form functions, by introducing a form of feedback.

Both of these types of circuit were particularly good for combinational circuits (a circuit whose outputs only depend on its current inputs) but needed additional external flip-flops for use in sequential designs (a circuit with memory, whose outputs depend on the current input *and* the sequence of past inputs). In response to this need SPLDs (simple programmable logic device) were developed, which included the flip-flops and multiplexers for selection purposes. Each section including the logic gates, flip-flops and multiplexers that drive each output became known as a macrocell. We see now the development into larger devices with sub-divided structures. Larger chips were developed which combined many SPLDs, these became known as CPLDs (complex programmable logic devices); an example of this type is the Xilinx XC9500 series. The XC95108 contains six configurable function blocks (CFBs) which is equivalent to an 18 macrocell SPLD with 36 inputs and 18 outputs. The outputs of the macrocells exit the I/O pins but also feed back into a global interconnection matrix. By using feedback signals, very complex multi-level logic functions can be built by programming the individual logic functions of the macrocells in each CFB and then connecting them through the switching matrix.

### 1.2.3 FPGA Architecture

An alternative architecture is found in FPGAs [26, 47]. The basic building block here is the look-up table (LUT). A typical LUT consists of four inputs and a small memory of 16 bits. Applying a binary combination to the inputs will match the address of a particular memory bit and make it output its value. Any four-input logic function can be built by programming the look-up table with the appropriate bits. For example, a four–input AND gate is built by loading the entire memory with 0 bits, except for a 1 bit that is placed in the cell that is activated when all bits are 1. In the Xilinx XC4000 series there are three LUTs combined with two flip-flops and some additional steering circuitry, to form a configurable logic block (CLB). The CLBs are arranged in an array with programmable switch matrices (PSMs) between. The PSMs form a similar function to the switching matrix in the CPLD, in that they route outputs from neighbouring CLBs to the inputs of a CLB.

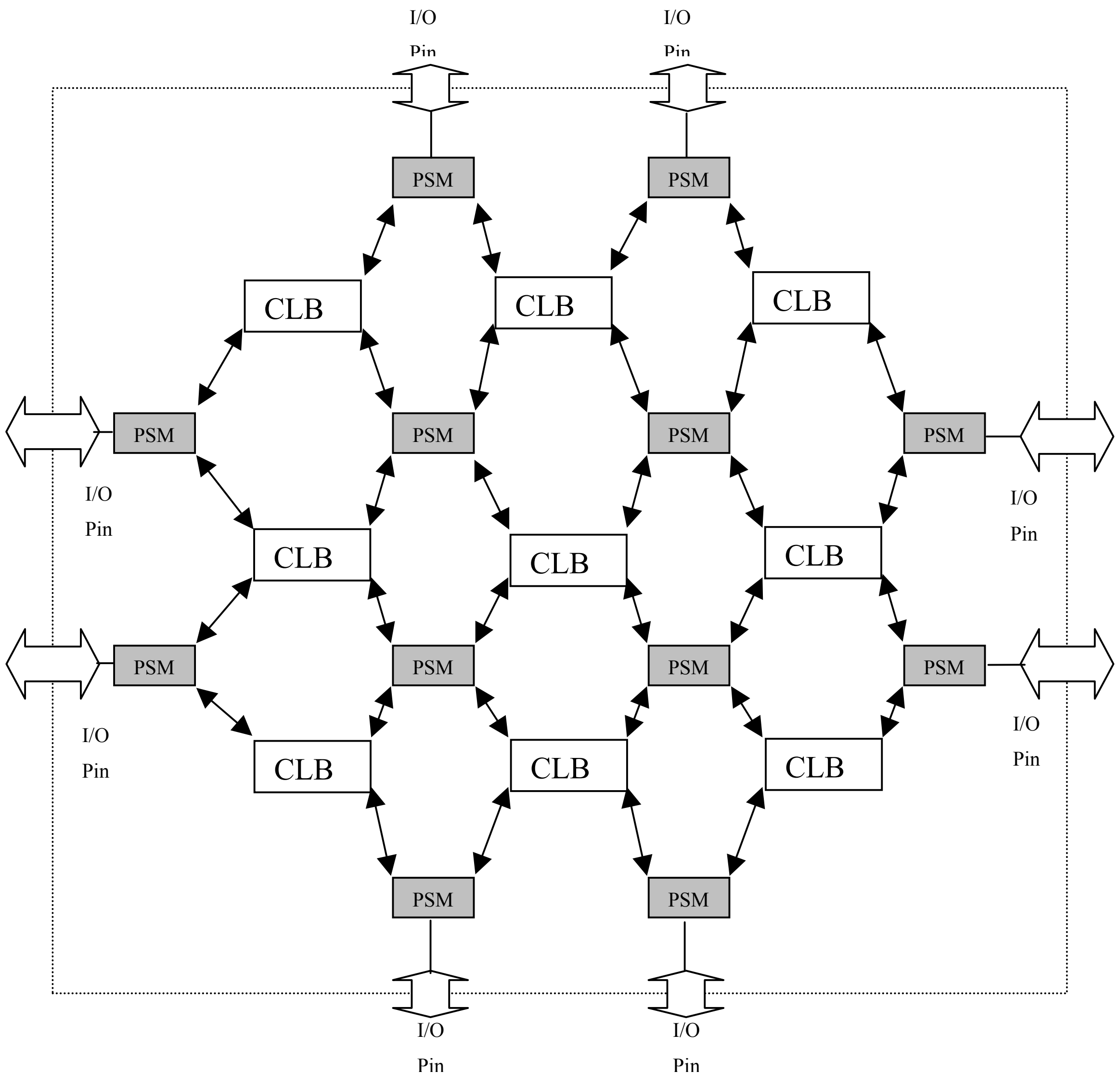


**Figure 4. Internal architecture of an FPGA**

The main question that arises for these devices is how these switches are set. Initially, in the case of early programmable devices, the switch arrays were manufactured with fuses at every cross point, such that every input was connected to each logic gate. Using special circuitry it was possible to burn out fuses at the cross-points to unwanted connections, eventually only leaving the desired logic functions. The disadvantage here was that once the fuses were blown, they could not be reset, so if there was a mistake, the entire device had to be thrown out and a new device programmed. In XC9500, XC4000 and Virtex families, the fuses are replaced with programmable switches. Each switch is controlled by a storage element that records its state. Changing the storage element's value changes the state of the switch and alters the functions of the device. This can be done repeatedly to implement new designs, or alter faulty ones, eliminating the need to buy a new device for each design.

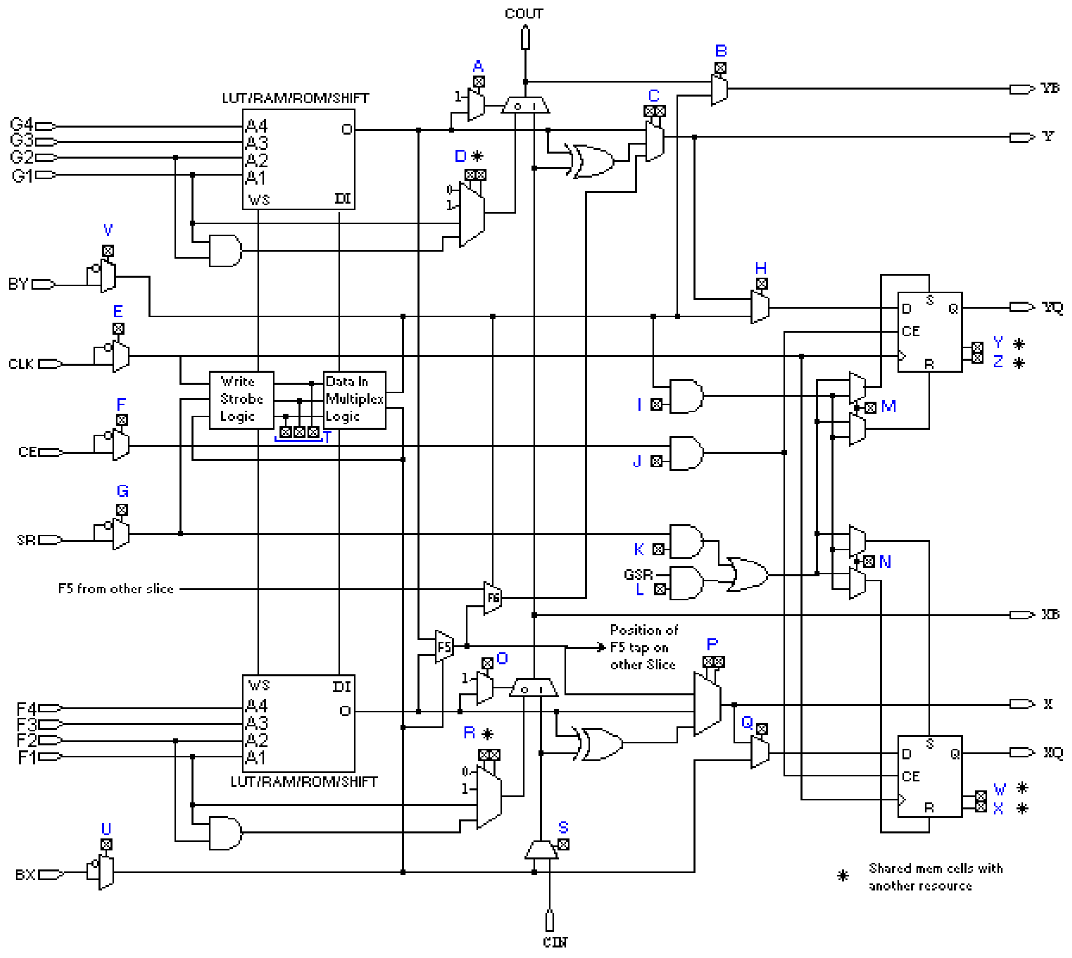


**Figure 5. Virtex internal slice**[5]

The XC9500 CPLDs use a non-volatile FLASH-based storage cell which retains its program even if there is no power. The XC4000 and Virtex families, along with most other common FPGAs, use random access memory (RAM), which needs to be reprogrammed each time the power is interrupted. This can either be done via download from a computer, or more likely, automatically at power up from an EPROM containing the design.

[5] Schematic from JBits user documentation.

### 1.2.4 Comparison to Alternatives

ASIC devices are partially manufactured by a vendor in generic form. This initial manufacturing process is the most complex, time consuming and expensive part of the whole process. The result is silicon chips with an array of unconnected transistors. The final manufacturing process of connecting the transistors together is then completed when a chip designer has a specific design for implementation. A vendor can usually do this in a couple of weeks. FPGAs cannot compare to ASIC technology in terms of speed and density but provide much greater flexibility, cost and rapid turnover of designs. Hardware compilation into FPGAs enables realistic working hardware-software systems in hours or even minutes.

FPGAs are completely manufactured but remain design independent. The vendor manufactures devices to a proprietary architecture, however, this includes a number of programmable logic blocks that are connected to programmable switching matrices. To configure a device involves downloading netlists into the logic blocks and programming the switching matrices to route signals between selected areas. One of the main advantages to using FPGAs over ASIC devices is that a designer can do this himself without sending out the design. Also, the FPGA can be reprogrammed at any time.

### 1.2.5 Utilising Extra Capabilities

While the current tools are capable of adequately programming and developing these reconfigurable systems, there exists much scope for both new design methods and dynamic functionality, which draws on the extra capabilities inherent in such flexible systems.

The advantages of reconfigurable systems are fairly obvious; they have use in emulating systems, developing easily alterable systems and quick development in rapid prototyping. It is possible, for example, to find the design libraries for most logic components, such as those within the 74 series. Cores are also available for most processors, such as the 68000 or 808X. These designs, once downloaded, act in exactly the same way as the original functionally and in some cases better, allowing faster clock rates to be achieved. Core designs can be found and downloaded – free – over the internet and there is a trend toward this.

| Algorithm | FPGA System | Comparison CPU | Speedup |
|---|---|---|---|
| DNA matching | SPLASH 2 | SPARC 10 | 43000 |
| RSA Crypto | PAM | Alpha 150MHz | 17.6 |
| Ray Casting | RIPP-10 | Pentium 75 MHz | 33.8 |
| FIR Filter | Xilinx FPGA | DSP 50MHz | 17.9 |
| Hidden Markov Model | Xilinx FPGA | SPARC 10 | 24.4 |
| Spec92 | MIPS+RC | MIPS | 1.12 |

**Table 1. Comparison of reconfigurable computing to conventional CPU**[6]

FPGAs have contributed significantly in logic design by stimulating new areas of application. PLAs, PALs and other forms of programmable sequencers have mainly been used for traditional logic replacement, which may reduce overall chip count and board complexity. FPGAs can be seen as an alternative medium in which to compile algorithms, particularly for differing systems, not based on the von Neumann architecture. They are suitable for experimentation in many areas, such as compiler design, hardware verification, automated VLSI synthesis, systolic processing[7] and general algorithm acceleration. They are particularly suited to various kinds of parallel processing experimentation – pipelining, vector computing and systolic arrangements. Initially, small FPGAs were used as programmable interface controllers between equipment but as their size continued to increase, they quickly outgrew this role. Eventually, groups of FPGAs could be used as a programmable co-processor to accelerate those tasks that were better for novel architectures, rather than those of the von Neumann processor.

[6] Data from Carnegie-Mellon University, http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15828s98/lectures/0112/index.htm.

[7] A type of computation which utilises a number of processors in an array. Each element takes in information from one direction and outputs it in another having applied some algorithm to the data.

### 1.2.6 Programming Reconfigurable ICs

What follows is a brief discussion of how a typical design flow is utilised in the programming of a FPGA. The Xilinx Foundation series is used as an example system.

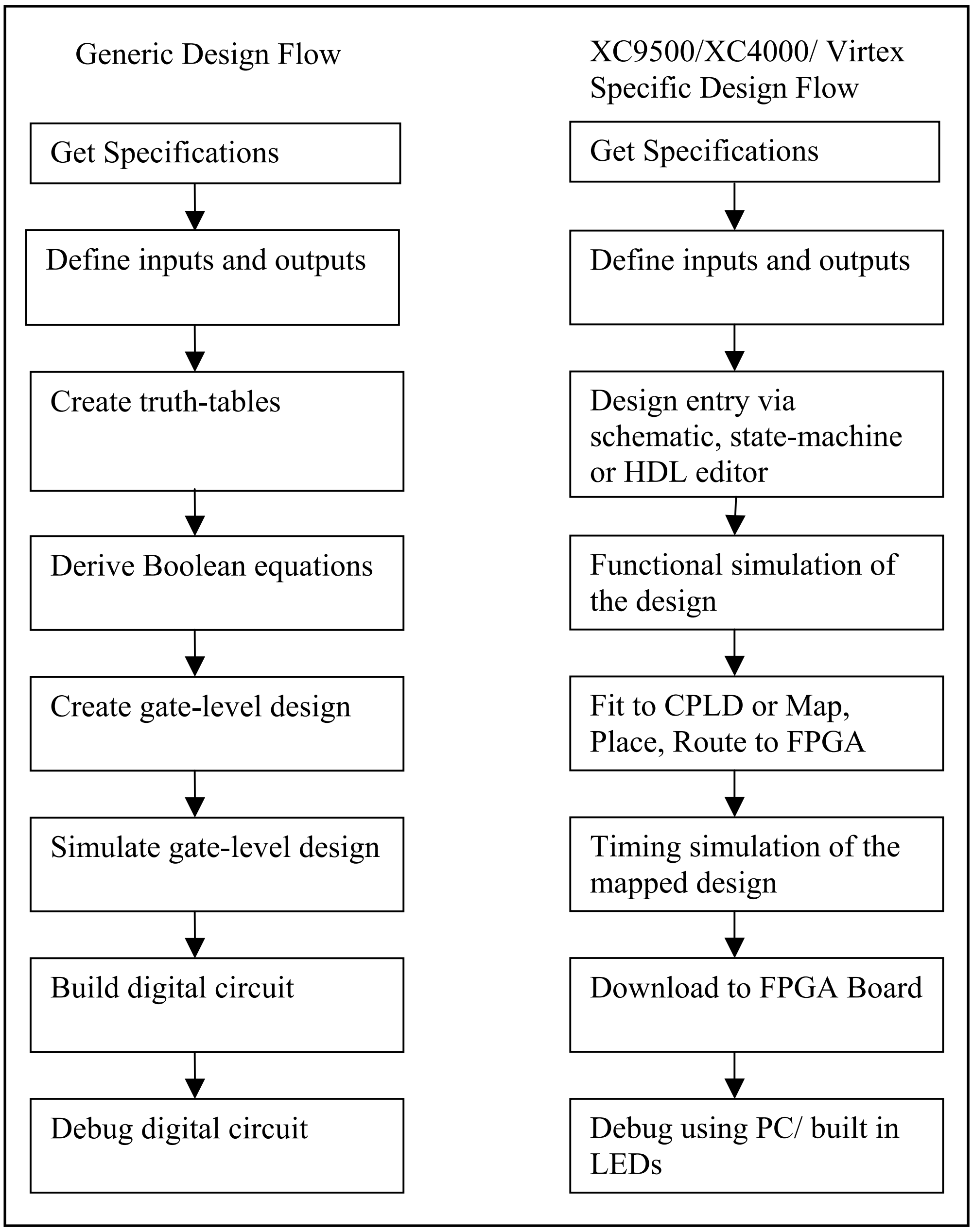

Generic Design Flow
Get Specifications
Define inputs and outputs
Create truth-tables
Derive Boolean equations
Create gate-level design
Simulate gate-level design
Build digital circuit
Debug digital circuit
XC9500/XC4000/ Virtex Specific Design Flow
Get Specifications
Define inputs and outputs
Design entry via schematic, state-machine or HDL editor
Functional simulation of the design
Fit to CPLD or Map, Place, Route to FPGA
Timing simulation of the mapped design
Download to FPGA Board
Debug using PC/ built in LEDs

**Figure 6. Comparison of generic and FPGA design**

The main flow of FPGA design [5] (here using an environment such as Xilinx Foundation 3.1i) to implementation is:

- Design entry: The design can be entered using several different methods – Schematic, HDL (ABEL, Verilog, VHDL), Finite State Diagram, CORE generator or LogiBLOX.

- Verification: This stage allows functional and behavioural simulation, for the identification of logic and other errors. Timing at this stage is acquired from standard delay units.

- Synthesis: The design, in a HDL, or as a schematic, is input to the appropriate netlist generator and a netlist (EDIF Netlist, EDN) is output.

- Translate: The input point to this stage is an EDIF (which may be named EDN, or EDIF Netlist) or Xilinx netlist file (XNF), output is a native generic database (NGD) file. In Foundation series software this translation is handled by NGDBuild. This stage merges all of the input netlists, the type of which is determined by a specially invoked program known as the Netlist Launcher. This starts the appropriate netlist reader program. All the various components and references are resolved to form one design. For example, a user constraint file (UCF) may be created that holds timing and location constraints. Through this it is possible to map certain points on the design to certain pins on the chip, as I/O points. The design is then checked using a Logical DRC (Design Rule Check) before finally being written. The NGD output file contains a logical description of the design reduced to Xilinx NGD primitives and a description in terms of the original hierarchy, expressed in the input netlist. This generic file can then be mapped to any desired device family.

- Map: This stage optimises the gates and trims unused logic in the merged NGD netlist. The program in the Foundation series is known as “Map”. This step also maps the design’s logic resources; logic in the design is mapped to resources in the silicon and a physical design rule check is again performed. The inputs to this stage are the NGD file, the outputs are a native circuit description (NCD) and a physical constraints (PCF) file. The PCF file contains constraint information usually as a consequence of the conversion of logical constraints from the NGD file which has been input.

- Place and Route: After mapping, the Flow Engine places and routes the design. Foundation series software does this with the Par program. In the Place stage, all logic blocks, including the configurable logic blocks (CLB) and input/output block (IOB) structures, are assigned to specific locations on the die. If timing constraints have been placed on particular logic components, the placer tries to meet those constraints by moving the corresponding logic blocks closer together. As well as timing driven methods, placement can also be governed additionally by cost-based considerations. Cost tables are used which assign weighted values to relevant factors, such as constraints, length of connection and available routing resources. The NCD file and PCF file from the preceding stage are input and a new NCD file is output which contains placement and routing information.

- Configure: The Flow Engine then translates the physical implementation into a configuration file (bit) that is used to program the FPGA. In Foundation software the BitGen executable creates the configuration file. This bitstream can then be downloaded to the chip by several standard programs. The NCD file is input at this point and a final bitstream (.bit) is output. The bit file contains all the configuration information from the NCD file, defining the internal logic and interconnections of the FPGA, plus device-specific information from the other files associated with the target device. The binary data in the bit file can then be downloaded into the FPGA's memory cells.

Several stages can also be added for verification, testing and tracing between each point. There exists ways of retracing steps backward to facilitate both reverse engineering towards another goal (such as conversion to another type of design or target device) and for ways of optimising under constraints, usually by introducing a form of feedback. This allows information on timing to re-enter the design at earlier stages as physical constraints (through PCF files) and therefore lead to improvements.

Most stages also produce reports and log files which trace the progress of a design. Further work can be done on a design if so desired, via editors and debuggers. These can analyse either the design or the actual hardware implementation, allowing visualisation of the implemented design. For example, it is possible to view the design built on the chip and zoom in and view the configuration of the CLBs, IOBs and net distribution. It is also possible to correct problems or alter the layout to a preferred design if part of the silicon is to be used for other reasons (for example, in partial reconfiguration, discussed later). The editor can then re-write a place and routing file, which can be re-loaded into the project for re-synthesis and implementation.

### 1.2.7 Reconfigurability

It is possible to view FPGAs in terms of their programmability. All devices of this type are programmable, a subset of this can be re-programmed by terminating the current operation and re-loading a complete device configuration. Another subset of this can be reconfigured partially by stopping operation (making inactive) and loading information only into the area required, without disturbing the rest of the logic circuit. Finally, dynamically reconfigurable FPGAs can be reconfigured while the circuit is still operational and a new design can be implemented in part or fully. It is possible to define the term “compile-time” configurable as referring to reconfigurable FPGAs and dynamically reconfigurable FPGAs as “run-time” reconfigurable.

### 1.2.8 Partial and Dynamic Reconfiguration

One of the most tantalising aspects of the FPGA devices is their ability to be reprogrammed while in circuit (this is also known as ISP – in system programmability)[8]. The usual method for the programming of such devices is on power-up but there is an additional capability in the form of causing a reconfiguration to occur at any point by taking a pin, usually called “PROGRAM” to low status. When this is done on a device which is not dynamically reconfigurable, the internal configuration is cleared and re-initialisation occurs. This begins the transfer of the configuration from external memory to the FPGA. On an FPGA which is partially reconfigurable, an inactive chip may have a portion of its area re-loaded.

[8] A technical description of active and dynamic reconfiguration is available in the application notes from Xilinx, Dynamic Reconfiguration XAPP093 [53].

On some devices, such as the Xilinx XC6200 (now only available for research) and the large density Xilinx Virtex family, a dynamic reconfiguration can occur. This allows part of the circuit on chip to be still functioning normally while a reconfiguration of the logic happens elsewhere. The mapping of logic occurs in frames; some data may actually fall upon an active area but only changed patterns will be affected. The main overhead for such dynamic reconfiguration is the time it takes for a transition to take place between the old logic pattern and the new. There are two ways of dealing with this, the first is to use a reconfiguration as mentioned above. The second is to use multiple-context configuration memory maps, which basically "switch-in" another area of the configuration memory to the active context. This context swap can be performed quickly across the entire configurable array, so these devices have the shortest dynamic configuration times. The actual multiple-context configuration memory, however, can occupy large silicon areas.

An FPGA may cause its own reconfiguration by connecting one of its output pins to the program pin. It may also effectively access its own reconfiguration data or have that data mapped into the memory space of  an external microprocessor . The external CPU can then alter the data as simply as adjusting variables. Yet another method of controlling reconfiguration is to employ multiple reconfigurable FPGAs, with one or more devices assuming responsibility for loading the configuration data into others.

In devices where the suspension of operation is necessary to implement functional changes, it must be decided whether the algorithm in use is best implemented in a different way if the overhead becomes too great. Any acceleration achieved by using such devices may be lost by this reconfiguration latency. Where different tasks are being performed concurrently, some may complete before others, making optimisation of the complete system difficult. The problem is increased if the device only supports a compile-time static type reconfiguration.

A problem in terms of memory space may occur for large scale devices if only a static reconfiguration is available. Without the ability to partially reconfigure specific areas, IC size bitmaps have to be stored, resulting in enormous areas of memory being required. There may be ways to manipulate the bitmap externally and exploit data dependencies, by using special procedures to reduce such storage requirements. However, even partial compile-time reconfigurability is probably better than this.

### 1.2.9 Alternatives to Conventional Systems Architecture

The ability of dynamic reconfiguration to selectively reconfigure active devices leads to the idea that a new system model is possible. By removing the von Neumann processor from the system, its operation can be either absorbed by a large enough FPGA, or replaced by some novel architecture, possibly better suited, or more efficient, to the task in hand. By giving the FPGA access to a RAM block, the system becomes as generic a model for computation as a similar system composed of von Neumann architecture. It would also be possible to include a von Neumann co-processor where a FPGA may find certain functionality hard to synthesise, for example, in the case of floating point arithmetic. In this instance, the co-processor becomes a arithmetic and logic unit (ALU), any other unrequired functionality can still be absorbed by the FPGA.

The self-reconfiguring system has many advantages to similar microprocessor-based designs. This is in terms of cost and speed (it is not based on the von Neumann interpretive cycle – fetch, decode, execute). It has been found in various practical situations it is simply better to get rid of the processor and rely instead on novel architecture, not based on the interpretive von Neumann model. A benefit of this being that the bottleneck in processing is taken out. This can have cost benefits. Designs can also be realised in what has become conventional, that is, the building block architecture. The array logic used in FPGAs has the speed of hardware with the flexibility of software. The speed factor increase comes from high clock frequencies, inherent parallelism of hardwiring and concurrency that comes from being able to execute multiple tasks on a single array simultaneously.

FPGAs can also be thought of as some extension of the concept of modern day memory. This perspective allows the viewpoint of treating it in a similar way to that of a memory cache in a conventional system. It is possible, in a similar way to a memory cache, to load only those circuits/processes which are currently active. The remainder reside in external RAM and load when required. This technique has been labelled “logic caching” by some.[9] Logic caching can be applied when the FPGA is either self-controlling, or when reconfiguration is controlled externally by the use of a processor.

### 1.2.10 Software Design Tools

There is a current need for better tools that realise the flexibility of reconfigurable devices, for example, there is no way to program such systems in a general purpose manner. Current tools are slow and hardware-oriented. Other software tools are also required which suit reconfigurable systems, such as:

- Simulation Models for dynamic reconfiguration.
- Automatic design partitioning, based on temporal specifications.
- Support for generation of relocatable bitstreams.
- A simulation package for modelling new FPGA architectures.
- Debugging tools [29].

### 1.2.11 Co-Synthesis

One of the main aspects of reconfigurable systems is the allowance of the medium to act in a similar way to memory for the implementation of algorithms. It is through this that makes it possible, in theory, to program it as software but with all the inherent capabilities that accompany a hardwired design. Additionally, there are also other factors which encourage its use, such as cost, flexibility and dynamic reconfiguration.

### 1.2.12 Convergence

The overall theme of this points to a closing of the gap between digital electronic design and the study and practice of computing. The increasing complexity and size of computer systems, in terms of both hardware and software, calls for the development of more comprehensive design tools. Such tools can help view the system as a whole through a unifying model.

It is at the boundaries of systems, in designs and understanding, where most errors are found, usually at sub-system interfaces. This is particularly true of systems which

[9] Lysaght 1995 [22].

have hardware and software components, due to the nature of the split in disciplines – allowing for misunderstanding, bad co-ordination and misuse of resources.

Current design methods in both areas are inadequate in working in each other's domains but represent a dichotomy in overall systems design and implementation. When considering the present state-of-the-art, with all its additional functionality inherent, in terms of reconfigurability, concurrency and flexibility, the present design and debugging tools are far from coming close to satisfactory. Hardware description languages are totally inadequate in their description of software; whereas software lacks some of the necessary features but could be adapted to meet the requirements. It is interesting to view the progression of hardware description languages over the last twenty years, which have tended toward becoming programming languages.

It is obvious that the answer to this is the development of a single descriptive language which encompasses the two domains. This may be done in several ways, for example, the language may attempt to describe an algorithm in neutral manner and then be targeted toward either hardware or software. Another way is to embed hardware units, independently developed, within code that is called from the software and executed.

The speed gained from the use of this kind of hardware comes from the inherent parallelism, resulting in extremely fast execution. This parallelism must be described by the programmer for synthesis, which results in a final hardware description. The software itself must describe the appropriate parallelism, or it must be extracted automatically. Given the correct language and tools, it is reasonable to assume that programmers can provide this level of description.

A problem arises in the shape of the two domains, that is; software and digital electronics, being split in such a way. Any programmer, at least in the short term, needs to be able to describe and synthesise hardware and software systems and needs to combine both areas of knowledge, currently belonging to different fields of specialism. Designer-programmers of the future will need to have a greater understanding of how such combined systems operate in terms of parallelism and the temporal behaviour of their designs. This temporal-spatial element in designs is probably more known to digital engineers than programmers at the moment. It represents not a particularly great challenge but rather a shift in perspective toward training with awareness of spatial and temporal implications in programming.

## 1. 3 Current Research and Development in Reconfigurable Systems

### 1.3.1 Main areas

There are several main streams of work in this area:

- *Hardware and Software Co-Design*

  This mainly concerns the convergence of the two areas into one; unifying system design. Languages developed to this end tend toward being for specific systems, with some amount of hardware knowledge still being required in their usage [21, 35, 36].
- *Fault Tolerance*

  Through adaptive and reconfigurable technologies it is feasible to suggest that it would be a relatively easy matter to create systems which are adaptable to circumstances and through this, robust in the face of hostile or disruptive environments. This is particularly of interest to military use and currently a heavily funded area by DARPA.
- *Rapid Prototyping*

  The nature of having a "soft" substrate in which to describe hardware components – the so called sea-of-gates allows rapid turnover of new designs.
- *Evolvable and Artificial Intelligence*

  The main ideas here revolve around allowing various weighted stochastic or learning programs generate solutions to problems, particularly based around genetic algorithms (GAs).
- *Novel Architectures*

  Experimentation with new kinds of architecture is possible mainly through the ease in which designs can be implemented. Parallel architectures involving pipelines for systolic and vector processing are relatively easy to develop.

### 1.3.2 Defining the problem tackled here

A great deal of past work with FPGAs has been held back by tools which rely on older methods of development and those which are based around static deployment of designs within the devices. Recent work has shown the potential available in this technology, particularly in light of the rapid expansion and ubiquity of the internet. This interconnectedness creates the perfect medium for communication with nodes that can use reconfigurable technology to assume whatever function is required.

The work discussed here impacts upon the entire cycle of development, envisaging a design period which extends into run-time. Its main objectives are to explore a fluid, dynamic approach to hardware, utilising knowledge gained in software engineering from various philosophies, such as object-oriented and CSP paradigms. In this way there is a dialogue between the two engineering spheres which inform this project.

A design methodology is required which embraces the mixed module approaches of software and hardware, utilising libraries and tools which express the flexibility of the medium. While there is a movement towards a design methodology similar to that in software engineering, there are many factors which necessarily differ and in some ways, improve, projects developed in this way.

As it is now possible to access state and configuration information at runtime from a device, there are many techniques that can be explored. Objects could be manipulated singularly, for example, for movement and placement. Interrogation of state and configuration is possible and runtime routing would allow additional circuitry to be added at this point.

Built on the above premise, it should be possible to extend the concept of software libraries into runtime, creating a dynamic system, changing content in a similar way to conventional microcomputer memory.

The infrastructure exists for such devices to be configured over network links; it exists in the sense of the appropriate hardware and software servers. The above mobility of entities should not be limited to placement within a device but have network mobility. This allows for remote, dynamic configuration and communication.

The project described here therefore aims to promote a fluid, dynamic approach to hardware manipulation in several key areas:

- At design-time, through mixed methods of hardware design input.
- A means of encapsulating hardware entities in the software domain, for transport and software presence/intelligence.
- At delivery time, through mobility of component objects.
- At run-time, through communication and interaction.

While these are the main aspects of the work, the above raises wider implications in several areas, such as language and design synthesis.

# 2. Context

## 2.1 Background

In this section, I will show how research within this area has grown, from initial grappling with definitions, taxonomies and requirements, through to actual systems which deal with the practicalities of implementing such concepts. I will consider these various theories and research, developing from this a set of criteria on which to build the basis of my own approach.

From the early beginnings of development with programmable electronics, the goal has always been to capture some of the flexibility of logic and memory we have in conventional computer software based systems. PLAs, PALs, programmable sequencers and FPGA technology that draw on architecture based on PLDs, are generally still used for logic replacement. FPGAs were originally targeted at traditional "glue logic" applications, replacing other types of PLDs. They were used to perform such functions as address decoding, wait state generation, or bus demultiplexing. The FPGA then evolved into primarily a memory mapped programmable device for implementing custom hardware interfaces. As the size and speed of FPGAs continued to increase, they quickly out-grew this role. It was realised that the devices, or groups of them, could be used as a programmable co-processor, to accelerate tasks better suited to its architecture, than that of the von Neumann processor.

Certain advanced FPGA technology took this a step further by being able to implement complex circuits, logic and memory, in ways that were easily modifiable. The devices soon became viewed as an exciting new medium on which to compile algorithms. They offer an alternative to the traditional von Neumann architecture and are sufficiently complex and cost effective enough to have become the focus for experiments in various areas such as; logic design, timeshared hardware, compiler design, hardware verification, automated VLSI synthesis, systolic processing and general algorithm acceleration. Recent developments allow configuration of these devices to take place while active, modifying only specific areas.

There is still a large gap between the design capabilities of FPGA hardware and the software which supports it. Designs are still, in the main, downloaded as one large design. Even though the general view of the engineer is component or module based, this is relegated to a "static" design-time.

Previous work in this area has realised both the difficulties and possible applications of such devices to be configured on-the-fly [28]. The work has primarily focused on developing a means of mapping designs at run-time, introducing ideas such as logic caching [22] and dynamic synthesis [30].

### 2.1.1 Taxonomy and Defining the Area

Initial work in this area was carefully explored by Patrick Lysaght and John Dunlop in the paper "Dynamic Reconfiguration of Field Programmable Gate Arrays" [22]. This work outlines the fundamental concepts and definitions for the dynamic and partial reconfiguration of FPGAs. It establishes the consistency of these definitions with terminology used to describe reconfiguration techniques elsewhere in digital electronics[10].

The paper defines the nomenclature of FPGAs and classification. They may be classified for example, by their configurability. All devices may be, by definition, configurable. A smaller subset may be reconfigured by terminating operation and re-

[10] Other notable works included DeHon's Reconfigurable Architectures for General-Purpose Computing, 1996 [9].

loading a complete device configuration (others may have only one download available similar to EPROM style chips). In general, any subset inherits the properties of its parent set, so, any reconfigurable device is programmable but the reverse does not hold.

A device is classed as dynamically reconfigurable if its embedded configuration storage circuitry can be updated selectively, i.e. nominated configuration storage circuits (and the corresponding logic functions and interconnections they control) can be changed without disturbing the operation of the remaining logic. These devices can be therefore configured while still active. This kind of circuitry is referred to as Dynamically Reconfigurable Logic (DRL). Such FPGAs share common features: they are cellular arrays of relatively fine-grained cells, whose configuration is controlled by writing data to static memory locations. The definition of dynamic reconfiguration also implies that a particular FPGA must be capable of partial reconfiguration while active. A device is defined as partially reconfigurable if it is possible to selectively reconfigure it, while the rest of the device remains inactive but retains its configuration information.

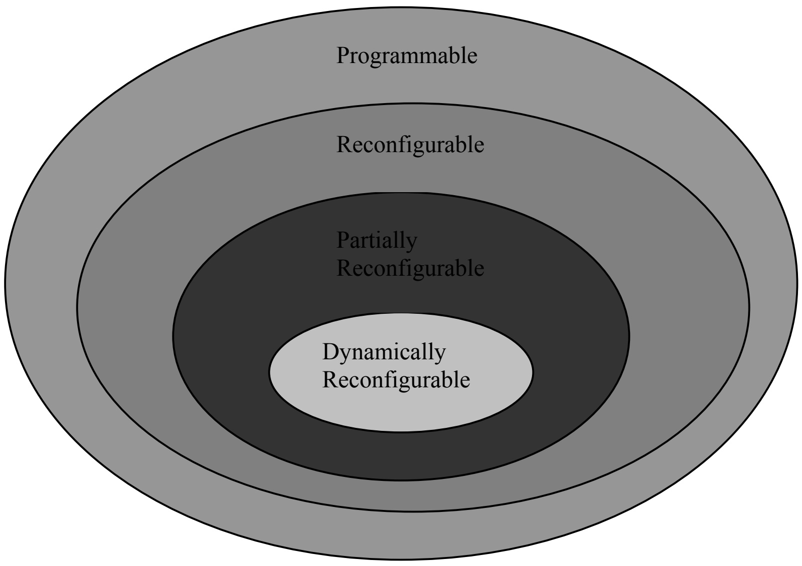


**Figure 7. Classification of FPGAs according to configurability**

It is also noted in this paper that there must be a consistent terminology used. Reconfiguration techniques are widely used in digital electronics for fault and defect tolerance. They are applied in parallel processing, particularly in applications which use processor arrays, such as systolic or wave front processors. Kung [58, 17] introduces the terms “run-time” and “compile-time” to qualify the reconfiguration of VLSI processor arrays. According to these definitions, reconfigurable FPGAs are compile-time reconfigurable and dynamically reconfigurable FPGAs are run-time reconfigurable. It follows from this that the FPGA that is reconfigurable at run-time is also reconfigurable at compile-time.

Other terms have also been defined by numerous authors, including; “static-reconfiguration” to refer to compile-time reconfiguration; “on-line reconfiguration” and “real-time reconfiguration”, in place of dynamic reconfiguration.

FPGAs that are not dynamically reconfigurable must be off-line (not active) before a reconfiguration cycle, partial or full, can begin. If such a suspension is required then this limits what can be actually done efficiently on the device. Usage, for example, as a programmable algorithm accelerator, is limited in this case, as even small changes to functionality become critical to its whole operation. Certain algorithms, therefore, become undesirable in this kind of set-up. Such devices also have problems with task organisation, when multiple tasks are being performed concurrently, as certain tasks are likely to complete before others, thus making it difficult to optimise the whole design for maximum speed advantage.

If a reconfigurable FPGA only supports complete reconfiguration the disadvantages multiply. As such devices increase in density and do not include the ability to partially reconfigure, this is likely to become more problematic. There would be, for example, a large overhead resulting from the need to store unique bit maps for every computational permutation. The Xilinx 4008 device (which is relatively small by latest metrics) requires 140,597 bits of programming data for each configuration. The amount of storage required for such a device may be manageable but there are some with hundreds of times the size of such a bit map. Time and storage penalties become inhibiting. The reconfiguration latency may offset any acceleration achieved by the array.

There may be ways to exploit data dependencies within the bit maps by using special procedures, performed externally, to reduce the storage requirements, while at the same time maintaining transparent access for the FPGA. Partial, compile-time, reconfigurability would be far better than this situation. Compile-time reconfiguration was at one time restricted to the reconfiguration of the entire device, as most devices, such as the Xilinx LCA, were reset prior to reconfiguration. The previous data was therefore lost and partial reconfiguration to implement only a small design change was not possible.

The Lysaght paper also details how such reconfigurations may take place. FPGAs would need, it is posited, a means of being reconfigured if many functional changes are required to its run-time design. For example, a microprocessor could take on the task of loading new designs, or multiple FPGAs could be used with one or more devices assuming responsibility for loading the others. Lysaght also develops the idea of the FPGA being a stand-alone host, replacing the von Neumann computer and where necessary, its functionality. The FPGA in this way becomes an example of a self-controlling, dynamically reconfigurable system. A system is detailed which is set up in a similar way to a conventional computer model i.e. connected to read/write memory. In this way it represents as generic a model of computation as the von Neumann system. The RAM block is critical here, as without it the model is simply that of the FPGA in "master mode". This model can be further altered to include a von Neumann co-processor, which completes tasks not easily implementable or difficult to synthesise efficiently on FPGAs. Such tasks include floating point arithmetic, where the von Neumann co-processor would essentially become an ALU (Arithmetic and Logic Unit); other functions performed by a CPU would be absorbed by the FPGA.

In such scenarios where the FPGA is the centre of a fully programmable system it is a necessity that the device is dynamically reconfigurable. The task in such a case, which controls and initiates the reconfiguration, itself resides on the target, thus making it impossible if the device must be stopped for reconfiguration.

In many applications where there is no need for a microprocessor in the normal sense of the word, self-reconfiguring systems have an immediate cost advantage [39]. The actual way in which FPGAs work also means that the execution of the von Neumann interpretive, fetch, decode and execute does not have to take place, since the reconfiguration process occurs much less frequently. This represents a significant speed advantage. The actual design of the configuration can also remain in the format popular with digital designers, that is, composed of building blocks.

Lysaght explores how Dynamically Reconfigurable Logic combines properties normally associated with software OR hardware. It offers, for example, the flexibility of programmable solutions and fast operating speeds, which are similar to those of hardware systems. Density on such devices can now reach over 10 million gates, with clock speeds of above 500MHz. The logic is faster than software (when executed on a microprocessor implementing comparable technology) for many applications. This advantage comes from the high clock frequencies, the inherent parallelism of hardwired solutions and the concurrency offered by executing multiple tasks on a single array simultaneously.

### 2.1.2 Logic Caching

Another area investigated and relevant here is logic caching. FPGAs have in the past been described as "programmable active memories". The operation of a self-controlling dynamically reconfigurable system can be seen to be analogous to that of a memory cache in a conventional computer system. When operating as a cache, only those circuits that are currently active are held in the memory. The remainder can reside in external RAM and be loaded as and when required. The operation of the logic cache optimises the ratio of active logic, with respect to silicon area. In other words it is possible to store many more circuits in external memory than could be possibly placed in the FPGA's SRAM at any one time. There is generally a greater amount of resources taken up by an implemented design in an SRAM FPGA than the equivalent anti-fuse based programmable device. This is due to the programmable nature of the resources, rather than the logic and spacing itself.

This logic caching in itself does not imply that the underlying system must be self-controlling, it is equally valid to use this particular approach in systems where reconfiguration is controlled by an external device, such as another FPGA, microcontroller or processor.

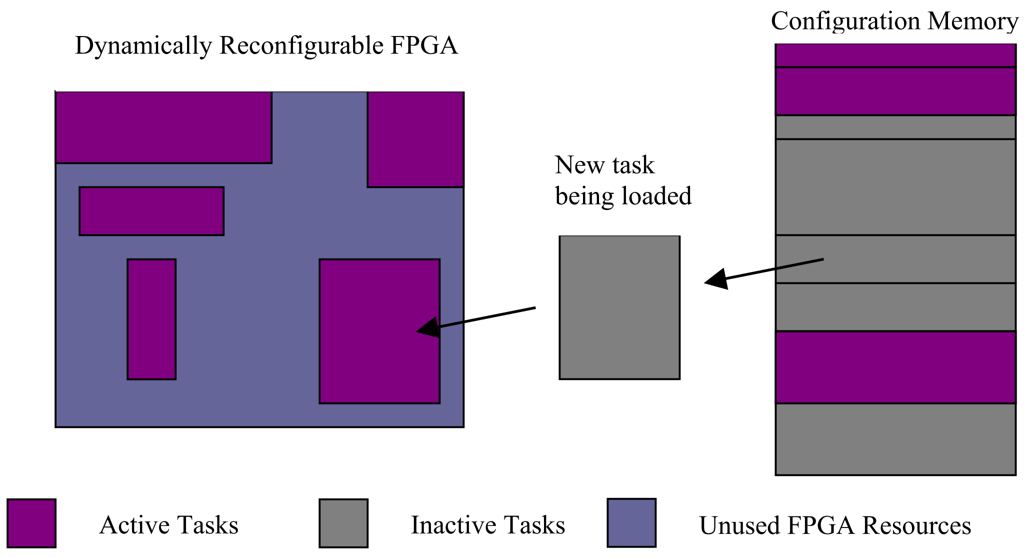


**Figure 8. Logic caching**

An interesting example is given in the paper, which utilises logic caching. The example employs dynamically reconfigurable logic to monitor in real-time a 64 kbps serial communications link. The instrument employs a multi-phase algorithm which begins with a parallel search of the incoming data for one of five possible framing structures.

The array is programmed to commence operation at the point when time is equal to zero. This compile time loads in six separate tasks T1-T6 and is usually the longest configuration period. There are five frame sequence detectors, T1-T5 and one further task, T6, which performs partial, run-time reconfiguration of the array, when required. Only one of the five frame sequence detectors can achieve frame synchronisation with any given data stream. Once one of the circuits gains frame lock, the other four become redundant. At this point, T6, activates to perform a partial reconfiguration of the array for phase two of the algorithm, while the frame sequence detector which achieved synchronisation, continues to maintain frame lock. The reconfiguration circuit and the synched task survive through a reconfiguration period, the length of which is proportional to the amount of new circuitry being loaded into the array. It is possible at this stage that extra, functionally-redundant data may be needed to be loaded on to the array to make redundant cells inert, by re-initialising them to the default state. Circuits which have been left in a powered up and semi-functional state have the potential to oscillate, cause cross-talk, or consume power.

The new circuits loaded into the array, called T7 and T8, extract a particular channel from the incoming stream and check the channel contents to whether it contains live traffic or control data. If the most significant bit of the channel byte is set, the channel contains control data and additional circuitry (T9, T10, T11) replace T8 (the data or control detection circuitry) to decode the control information. The new tasks take whatever appropriate action is required, such as the activation of local loop-back, or comparison of incoming control data to a locally generated reference, such as a CCITT 16 pseudo random binary random sequence. In the event of loss of frame synchronisation, the data test set suspends current activity and the operation reverts to phase one, via a dynamic reconfiguration, controlled by T6.

In this case a self-controlling system would have to be controlled and reconfigured by tasks T2 and T6. After this first configuration, tasks T2, T6, T7 and T8 are resident on the array. A partial reconfiguration then takes place with tasks T9, T10 and T11 substituted for task T8. At any given time, the maximum number of tasks which may be present simultaneously on the array is determined by a potentially complex relationship. This involves such factors as array size, the combined task sizes and any special requirements, such as input/output pin access, that may be needed if certain tasks are to run.

The example given by Lysaght, illustrates many key points about the algorithms needed in dynamically reconfigurable logic devices which use logic caching[11]. Characteristics shown in this case, such as the algorithm being multi-phase and each phase having several component tasks, are representative of the solutions to various problems utilising this approach.

In the first phase each frame synchronisation detector must execute in parallel for maximum speed of operation, until one circuit detects frame lock. When this happens

[11] Logic caching is also explored more recently by K.Compton and S. Hauck, Configuration Caching Techniques for FPGA [8].

all other frame synchronisation circuitry is redundant and the next phase must be implemented. This type of search algorithm is ideal for use in hardware multitasking systems such as this. For any data stream, n-1 of the n frame synchronisation circuits and the tasks that are needed in subsequent phases are mutually exclusive to each other, with respect to time.

The system detailed above is a real time application that requires several tasks to be executed simultaneously, which is ideally suited to the speed, concurrency and parallelism of a dynamically reconfigurable system. The delay involved in reconfiguring the FPGA is minimal; for example, in this cited case the next stage after frame lock is the user entering information which is several magnitudes greater than the latency involved in the hardware dynamically reconfiguring. The delay involved can be important to take into consideration in most cases. In this case, the application did not need any numerically intensive computations. FPGAs are generally not the best choice for such mathematical calculations as multiplication and square roots, due to the inherent delay characteristics of multi-level logic. Certain applications which are more suited to FPGA synthesis include string matching, error coding and decoding, bit level signal processing and bit level systolic algorithms. It can be difficult to match up applications to the available features and capabilities – programmability and exhibiting the multi-phase, multi-tasking characteristics with mutual-exclusion between tasks over time. Complex digital systems exhibit these features more often.

As FPGAs have increased in density and complexity, it has become evident that these features are present. Some areas of such large designs are not in use for long periods. Implementing complex designs without dynamic reconfiguration would no doubt lead to more than one FPGA required, which complicates functionality by introducing interfaces for inter-FPGA linkage, as well as how to partition the design efficiently enough, without a reduction in system speed. Logic caching in such cases would reduce the number of packages required, saving space, power and money[12].

[12] A rigorous mathematical example of context switching/logic caching with FPGAs is provided by S. Scalea et al, A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing [38].

There are three principal factors cited for the task of converting programs to parallel processing environments which interact in complex ways, these are: the application, the algorithm chosen for implementing it and the architecture of the target system. It is a relatively easy matter to develop a parallel processing system from scratch but it can be difficult to exploit the available parallelism by converting an existing application. Another factor involved is that the FPGA has an entirely different basis, as it is not a von Neumann architecture.

Lysaght offers a model of abstraction which views the programming elements as being mirrored in a memory plane which lies above the logic and routing plane but controls the latter's operation. He suggests that the overhead imposed by this layer is large, given that, without dynamic reconfiguration, its function is entirely static. Various manufacturers, for example, have developed FPGAs which recoup the functionality from the logic in the memory plane, by introducing modes of operation which allow such blocks to be directly accessible and can be used for data storage. This is further helped by the introduction of quantities (kbits worth) being available in RAM blocks for easy access. Using dynamically reconfigurable logic with the memory model allows a more general method to achieving more "functionality" from the memory plane in SRAM based FPGAs. It is suggested that by extending this model it would be possible to allow logic on the array to directly access the configuration data of other cells by introducing a more radical coupling between the logic and memory planes. This would in effect by-pass the normal loading mechanism for configuration data.

This particular paper concludes with some interesting observations which have not yet been fully realised. Many CAD tools have still to fully implement ways of dealing with reconfigurable and dynamically reconfigurable devices. While such tools support the basic device, they are yet to implement an interface to exploit the available power fully. The features for such systems are also noted:

- FPGAs optimised for speed of reconfiguration.
- A small viable unit of reconfigurability.
- Access to reconfiguration control circuitry from inside the array.

- Orthogonal (i.e. regular in two dimensions) routing and logic resources.
- Course-grained dynamically reconfigurable FPGAs.
- Co-ordinate addressing of individual array cells.
- Write to and read back from the storage resources of individual elements of any given cells.

Many of these features are now available in such devices as the Virtex family. For example they are orthogonal with regard to routing and logic resources, have co-ordinate addressing of individual array cells and there is a capability for reading and storing elements of individual cells. The access to configuration data by the chip itself is still somewhat cumbersome. The JBits model although extrinsic (the host running it is external to the device), employs a similar model to the one proposed by Lysaght and others in his paper. For example, there is a memory layer which is written to and read back from in the host, providing a kind of buffer. This buffer is updated on both sides and kept up-to-date should an event take place. On a read back from the device any changes are copied to the memory. From the host side, any changes, such as the adding or alteration or circuits or logic cells, results in such areas being marked as "dirty" until a synchronisation has taken place which brings the device in line with the memory. As mentioned, this differs in that it is not a task on the device accessing the memory plane but an application on the host.

In software tools he also notes the following possibilities:

- Simulation models for dynamic reconfiguration.
- Automatic design partitioning based on temporal specifications.
- Support for the generation of relocatable bitstreams.
- Deterministic APR optimised for rectangular areas.
- A simulation package for modelling new FPGA architectures.
- Debugging tools.

Many of these points have been met by recent software. For example, the Xilinx JBits package provides simulation and debugging tools. Relocatable bitstreams will be addressed later in this work.

### 2.1.3 Early Prototyping

In “Prototyping Environment for Dynamically Reconfigurable Logic” by Lysaght et al., 1996 [23], the idea of the universal dynamically configurable system is developed. This extends the above paper by implementing a system which takes the above factors into account. It is “universal” because it aims to be able to work with a wide range of FPGAs, no matter what physical interfaces are provided.

A detail of the prototyping system is provided here. The environment has two modes, standalone and with host. The actual layout and hardware used is based on utilising a wirewrap area for connecting a variety of FPGA devices; a socket for an Atmel AT6005 FPGA; several ispLSI-1032 CPLDs; two T805 transputers plus B430 Tram and a large amount of EERAM. The system is connected as follows. The transputers provide the interfacing capability with the host computer (usually a PC) and function as an embedded microprocessor[13]. The wirewrap area and Atmel are connected to the CPLDs. These provide the interface and memory access control between the target device and the memory and between the memory and the host, via the transputer link.

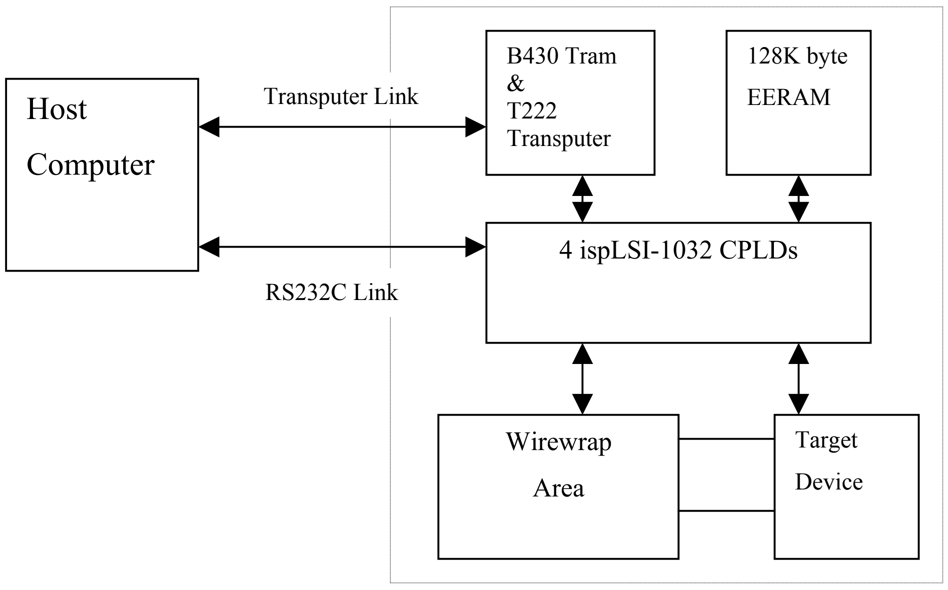


**Figure 9. Diagram of prototyping system**

[13] Others develop the idea of augmenting a microprocessor with reconfigurable hardware [14].

The CPLDs are non-volatile, can be programmed up to 1,000 times and are electrically erasable integrated circuits, which can be configured in-circuit via a further RS232C link to a host PC. These devices can then be customised to provide the appropriate interface for the specific FPGA which is plugged in. The main purpose of the CPLDS in this instance is to function as a hardware device driver, similar to that which is found in operating systems. They act to conceal any FPGA device-specific interfacing details, such as configuration control signal generation and routing of data pins, from the software part of the prototyping system. This encapsulation of the device-specific functions allows the interface to the computer to be standardised and kept independent of the particular FPGA in use. Some modification is of course needed for power connections, which vary between devices but this is minimal and fairly easy to implement using the wirewrap board. Such a system allows a fairly quick universal method of utilising new FPGA devices, which is universal in application. The fact that it exhibits universality in the target FPGAs it can be implemented upon, allows many new devices to be used and experimented with relatively easily.

The main configuration modes are standalone and host controlled. The extrinsic host mode allows debugging and development. Data can be retrieved and analysed. In stand-alone mode the FPGA can access the RAM for configuration and so applications can be developed which utilise dynamic self-reconfiguring techniques. The CPLDs are configured to provide a fast DMA mechanism, allowing the FPGA to be programmed directly from the EERAM. This stored configuration data must initially be downloaded from the host, via the transputer link but once this is done, the data is retained through the RAM being non-volatile, so no further microprocessor intervention is required. The transputer can also function as a completely separate, independent local processor.

Such a system can be used for dynamic reconfiguration, new FPGA and system architectures, system analysis and performance measurement, design methodologies and design automation.

At the time of development of the prototyping environment there were many limitations for such a system. There were few FPGAs, for example, that could be dynamically reconfigured; such configuration was relatively slow and the density of such devices was fairly low, compared to those available now. Many FPGAs could only be partially reconfigured while offline and inactive. Software tools for work with FPGAs were still in their infancy, mainly being CAD tools adapted from less flexible devices.

The development board was planned to have a more powerful transputer added with greater system memory. Another plan was to absorb the programming interface for the CPLDs into the new embedded transputer, reducing the need for the RS232C connection, which was currently the main way of programming them from the host. A problem which developed with the initial prototyping system was that the transputer link with the host had limited bandwidth, proving unsuitable for some data-intensive applications, which required post-processing with the PC. By including a more powerful transputer much more of the processing could be done on the actual system board, reducing the need for a higher bandwidth link.

One of the main problems of such a system was the amount of software support for the actual synthesis of designs. Any new FPGA added required its own design tools on the host system for development, reducing the ability to automate the process of synthesis and implementation. This still exists to some extent today; there are few tools which will develop for many manufacturers, as there are proprietary drivers and software involved. Usually the best way to deal with such matters is to stay with one manufacturer and develop within one family of devices catered for by one tool-set.

This was possibly not the answer here, as the main idea for the described system was one of universality, where the only choice may have been utilising batch driven synthesis, which included a manual element of input for specifics. The inclusion of more than one FPGA on the system would have further compounded the problem, especially if they were from different manufacturing sources.

### 2.1.4 Languages – Java

Development of reconfigurable systems has led to the requirement of such features to be harnessed in the sense of programming language. In "JHDL – An HDL for Reconfigurable Systems" 1999 [4], Peter Bellows and Brad Hutchings develop the idea of a language which is closer to software ideology than hardware engineering[14]. This follows the idea of using a Java-like language to describe and build circuits. A library is available for each type of FPGA programmed, this library extends the capabilities of Java so resources can be utilised. Circuits are treated as objects in a similar way to software object-oriented programming, with construction and destruction occurring in a similar way. The way this is done is by describing circuits within the code itself, integrating the possibility of change over time within the programming language. Normally, the designing is done at a static point prior to run-time, any changes occurring on a reconfigurable system being done by a host or embedded microprocessor.

The primary objective of the research described here is to develop a tool-suite/design environment for describing circuit organisations that dynamically change their structure over time. They set out the following requirements; that it must use an existing language with no extensions; the level of abstraction should be high; it must support run-time and partial configuration; that it should be able to simulate the developed program. Other languages that have been used to describe circuits are mentioned here such as Perle (C++ based), Spyder and Lola.

JHDL is a design tool which attempts to hide details of configuration from the user. It does not attempt to automatically identify partial configurations; nor does it address the run-time physical transformation of circuits so that they will fit within available FPGA resources. It is primarily a manual design tool that combines some control aspects and circuit design into a single integrated description. This description allows the designer to express circuit organisations that dynamically change over time. Their intention here is to remove the detail normally required to perform control over such actions in a dynamically reconfigurable system. For example, the need for a

[14] A similar idea to this is presented by N.Weaver et al, Object-Oriented Circuit Generators in Java [48].

partial configuration may be explicitly stated by the user's control programming. Rather than invent a new language or method of doing this, a software object-oriented approach was taken. This uses the object-instance construction/ destruction mechanism found in such paradigms. The languages which use this approach manage memory through object constructors and destructors. At the time of use the object is invoked and memory allocated from the heap or stack, setting object variables to initial values. Once such memory has been allocated and the object instantiated in this way, it is necessary to use a destructor to reclaim resources taken by it. A similar method is developed in JHDL, to reclaim FPGA resources. The language used also parallels the equivalent software semantics but applied to the FPGA rather than system memory.

When used the objects can be applied to either a simulation or hardware. In the case of a simulation, a kernel is used in place of the hardware which models a device providing a clock-by-clock simulation of the user circuit. In the case of a hardware execution the constructors/destructors communicate directly with the FPGA resources so circuits can be built and destroyed. While constructors build the circuit using details from a library, analogously destructors replace existing circuits with blank configurations, similar to a FPGA reset state.

At the time of publishing their paper, the system was at an experimental stage, the main goal being the demonstration of feasibility regarding the constructor/destructor mechanism as a means of controlling the instantiation of circuits. At this point JHDL implemented a circuit simulator and the control API for the Hotworks board from Virtual Computer Corporation (VCC). An actual Netlist format was not yet determined. All circuits were therefore designed manually and the simulation models written using the JHDL primitives. Most circuits were also hand placed.

The JDHL system is implemented as a set of Java class libraries with functionality having a dual role: circuit simulation and runtime support. The circuit simulation classes allow the designer to produce models that can be simulated at clock level with the JHDL kernel. The runtime support classes provide transparent access to control functions for the constructors and destructors, mentioned earlier.

Designers develop circuits in JHDL by selecting from a set of synchronous and combinational elements and wiring these together to form any arbitrary synchronous circuit. There are three different classes that can be used to implement a circuit: `CL` (combinational), `Synchronous` (clocked) and `Structural` (interconnection of elements). The designers' first task is to decide whether the outputs of the circuit are updated continuously, i.e., a combinational circuit (a `CL` object), or updated on the clock edge (a `Synchronous` object), or if it is a structural circuit (`Structural` object) i.e. a circuit which is a set of existing synchronous or combinational circuit elements interconnected together. The designer defines a new class that inherits from the appropriate class and implements the required functionality in the constructor and other methods. Circuits can be wired together by passing `Wire` objects as arguments to the object constructors.

The actual behaviour of a defined class is specified differently depending on whether it is a `CL` or `Synchronous` object. The `CL` object requires a `propagate()` method that will generate a new set of outputs based on the current inputs each time it is called. This method is called automatically by the simulation kernel each time at least one of the input wires connected to the circuit object registers a change during simulation. A `Synchronous` object requires a `behaviour()` method that will generate a new set of outputs each time it is called. The `behaviour()` method is invoked automatically each time a new clock cycle is issued by the simulation kernel. Structural designs only require a class to be derived from the `Structural` class and the writing of a constructor that will wire up the appropriate library elements to gain the required functionality. `Propagate()` and `behaviour()` methods are not used in `Structural` circuits as their behaviour is derived completely from the behaviours of interconnected sub-circuits. A degree of flexibility is available for organisation of the circuits and an arbitrary number of hierarchical levels are supported.

The simulation system is limited to synchronous, globally clocked circuits, although it is possible to alter the simulation kernel to support multiple clocks if necessary. A problem with the system is that it requires that clocks are stopped before relevant changes are made to circuitry; this means that although it is dynamically reconfigurable it is not actively reconfigurable – operation must be suspended while changes are made.

The user circuit is encapsulated in a top level class called `HWSystem`. The `HWSystem` provides all the functionality for simulation and device control. This class also provides the functionality required for the system to communicate with the outside world or host computer system. Communication is provided through “wires” called `ports`. These points of exchange synchronise input and output with the global clock used in JHDL. There may be many objects in the circuit but the top level object is always encapsulated by the `HWSystem`. It provides the essential link between the simulation and hardware execution, by implementing the simulation kernel and invoking behavioural descriptions of the circuit objects during software imitation. The `HWSystem` also implements the API to device drivers, coordinating computation on the FPGA by configuring circuits; loading data and wiring resources. The `HWSystem` object tracks all the wire and logic objects it encapsulates. As each object is constructed it registers itself on a wire list or clock fanout list in the `HWSystem`. The `Inport` and `Outport` classes are the main points of access to the top level circuit via the `HWSystem`; these can be simulated behaviourally, as software buffers that are synchronised to the global clock, or executed on the device as actual hardware ports that are completely functional as communication points to a host system, for example. This duality implies that any circuit described in JHDL can be run interchangeably in the simulation kernel, or on a hardware platform without any modification of the source code. The `HWSystem` simulation runs the following algorithm:

1) Cause each `Inport` to drive its wires with the next data in its buffer.
2) Issue a “clock” to all synchronous logic (by calling the `behaviour()` method for every `Synchronous` circuit object).
3) Propagate all wires that were updated by the synchronous logic.
4) Propagate all affected combinational logic (by invoking the `propagate()` methods of all `CL` objects in the wires’ fan-outs).
5) Repeat 3-4 above until the network is stable.
6) Cause each `OutPort` to write the state of its associated wire into its buffer.
7) Repeat 1-6 above until the requested number of clocks have been issued.

The actual hardware execution and computation is performed differently to this. On the construction of the `HWSystem` there is a call to the device driver to load the initial configuration bitstream into the device. The user must provide the name of the file to the `HWSystem` object but is to be replaced when netlisting capabilities are added. On receiving a request to clock the circuit, the `HWSystem` makes a device driver call to clock the hardware device the desired number of times. Both the `InPort` and `OutPort` objects also call the device driver separately from the `HWSystem` to communicate or exchange values with the device. The hardware execution cycle is therefore:

1) The user passes input data to the `InPort` buffer. The `InPort` sends data to the device via a driver call and the driver buffers the data.
2) The user requests a sequence of clocks. The `HWSystem` makes a driver call to issue the clocks.
3) The driver issues the clocks and buffers the data for each `OutPort`.
4) The `HWSystem` waits for the driver call to complete. When it does, it requests the data for each `OutPort` and loads the data into the appropriate objects.

An attempt has been made at a degree of "platform independence", or universality, as to what devices can be used in the JHDL system. The JHDL API must be supported by the hardware device. The driver is compiled to native code and the driver calls linked to JHDL as native methods. The driver is then loaded at run-time as a shared library; changing the hardware platform is a question of changing the library file. The first hardware to be implemented was the VCC HotWorks board, which is based on a Xilinx XC6216 series FPGA. The driver is a wrapper around an already existing device driver, developed at BYU, which adds buffering capability and exports the appropriate API.

As already stated, the main method of reconfiguration is via constructor and destructor objects. Java supports only explicit object construction and not destruction, as this is the task of the garbage collection system, when references to an object are no longer present in a application or program. All memory resources in such a case are reclaimed automatically. JHDL therefore needs to provide a `delete()` method for each of the base circuit classes. When `delete()` is invoked on an object, its internal mapping on a netlist graph is removed (dereferenced so they can be garbage collected) and marked as deleted. Invoking the `delete()` method also causes the hardware at the specific circuit locations to be "blanked".

Partial reconfiguration is supported through an additional interface class, the `PRSocket` (Partial Reconfiguration Socket). This is used to describe parts of the circuit which need partial reconfiguration. It is this class that maintains the multiple configuration information and automatically provides the transparent switching between simulation and hardware execution. The `PRSocket` provides a model for applying partial reconfiguration to an existing circuit. This mnemonic acts as a kind of place holder in the internal netlist, modelled as a discrete chip socket that allows any chip with the correct pinout to be plugged in. Any static logic in the circuit is connected to this socket. Any circuit with the correct interface may be "plugged in". When the simulator is in operation, the `PRSocket` simply dereferences all pointers to the underlying JHDL object and creates an instance of the newly configured object. In a hardware execution the `PRSocket` communicates with the hardware drivers to load the new partial configuration at the correct chip position.

When a partial reconfiguration is to take place, the `PRSocket` must be told in advance which configurations it will encapsulate. The `PRSocket`, using this information as a constraint, develops a netlist and also utilises this information for hardware execution. `ConfigGroup` encapsulates a list of configurations defining a similar function to this:

```
class myConfigGroup extends ConfigGroup {
...
      /* A Node is the base class for all JHDL logic
      Node getNewCircuit(int id, PRSocket sock) {
            switch(id) {
                  case 1:
                        return new Circuit1(...);
                  case 2:
                        return new Circuit2(...);
                  case 3:
                        return new Circuit3(...);

                  ...
            }
      }

}
```

The `ConfigGroup` object is passed to the `PRSocket` when it is constructed and the user connects it up to the static logic, as per any other logic structure. When circuit construction occurs it acts as a placeholder in the JHDL internal netlist. The `PRSocket` instructs the device driver to keep a pointer to all partial configurations, as well as loading the static configuration. A `Reconfigure()` method of the `PRSocket` is invoked with the ID number of the partial configuration to identify the specific circuit to be updated. It references the partial configuration and calls the device driver to load the appropriate configuration into the corresponding part of the circuit. This, of course, is only feasible with an FPGA device which supports partial reconfiguration, such as the Xilinx XC6200. The VCC HotWorks device driver had to be augmented to support partial reconfiguration.

Several examples of usage were given in the paper. These have been implemented on the Xilinx 6200 using partial reconfiguration. These were initially developed and placed manually. The first example given is the "shapesum" and "correlation" functions of the Chunky-SLD Automatic Target Recognition (ATR) problem [37]. Using the system a single description can be built of the control and circuit, avoiding the need for Tcl/Tk scripting program for controlling the device loading. The correspondence between the JHDL circuit object and configuration bitstream is managed manually by the user through the ConfigGroup, as already mentioned above. Practically, this means that the designer is responsible for generating bitstreams with a tool and "telling" JHDL where these files are and what circuit objects they are associated with. The entire run can also be simulated correctly.

A slightly more simple example is given by way of demonstrating coding and functionality. A FIR filter is given as the main example, assuming tap-weights are compiled directly into the circuit and partial configuration is used to modify the weights at run-time. The top level is a user-written Java program to provide the interface, gather input data and so forth. The following program wires it into the top-level system as follows:

```
class myJavaCode {
     SomeMethod(...) {
```

```
/* Create a new system */
HWSystem system = new HWSystem();

/* Tell the system how to compute –
this time, we’ll simulate */
system.setSWMode();

/* Create new wires to pass to the filter,
 8 bits wide each
 The ‘system’ reference helps the Wire class build
 the circuit graph (“system” is the parent) */

Wire Input = new Wire(system, 8);
Wire Output = new Wire(system, 8);

/* Create a new filter object;
   pass in pointers to I/O wires.
   Note that this configures when
   in hardware mode. */
FirFilter filter =
     new FirFilter(system, Input Output);

/* Encapsulate the I/O wires with ports. */
InPort inport = system.newInPort(Input);
OutPort outport = system.newOutPort(Output);

/* The object is now constructed
   and appropriately encapsulated.
   Now, reconfigure the tap constants;
   this method is user-defined. */
filter.Reconfigure(GetNewTapConstants());

/* Now, initialise the input buffer with data. */
inport.writeBuffer(InputData);

/* Allocate a new output buffer, same size
   as the data array */
outport.newBuffer(InputData.length);

/* Now clock the circuit and get the results. */
System.Clock(some_number);
int Results[] = outport.getBuffer();

...
```

The actual FIR filter circuit could be built:

```
/* This is just a structural circuit – all behaviour is
described in the fir taps */

class FirFilter extends Structural {

     Wire[] data_wire_array, mac_wire_array;
     Wire fir_zero, data_input, data_output, fir_output;
     PRSocket FirTaps[];
     int tapCount;
```

```
    /* This manages the partial reconfigurations
       for each fir tap */
    static FirConfigGroup config = new FirConfigGroup();

    FirFilter(Node parent, Wire in, Wire out) {
        /* Every object that inherits from Node
           must do this first to build
           the netlist graph */
        super(parent);

        /* Now, declare my inputs/ outputs, 8 bits each */
        inPort(in, 8);
        outPort(out, 8);

        fir_zero = new Wire(this, 8);
        ... [initialise other wires in this manner]

        for (i=1; i<TapCount; i++) {
            FirTaps = new PRSocket(this, config);
            /* Now we must declare the static interface
               to each PRSocket. Each wire is assigned a
               “port number” for reference. */
            FirTaps[i].inPort(data_wire_array[i-1],0);
            FirTaps[i].inPort(data_wire_array[i-1],1);
            FirTaps[i].inPort(data_wire_array[i-1],2);
            FirTaps[i].inPort(data_wire_array[i-1],3);
        }
    }

    /* The user defines this to export to a top-level
       Reconfigure method to the outside world. However,
       the actual reconfiguration of each individual
       circuit is handled by the PRSocket. */
    public void Reconfigure(int tap_constants[]) {
        for (int i=0; i<TapCount; i++) {
            FirTaps[i].Reconfigure(tap_constants[i]);
        }
    }
}
```

The user must also define the `FirConfigGroup` so that it returns the kind of object he wants:

```
class FirConfigGroup extends ConfigGroup {

    /* Tell the superclass how many ports(4)
       and potential configurations
       8 bit fir tap constant => 256 are allowed) */

    public FirConfigGroup(HWSystem s) { super(s, 4, 256);  }



    /* Here, the reconfiguration is completely
      described by returning pointers
       to new objects of the desired type.
      In our case, the id represents the tap
```

```
        constant and associated name of the
        file containing the configuration
         to be loaded. */

     public Node getPRObject(int id, PRSocket sock) {
          return new FirTap(sock, id);
     }
}
```

The user must also describe the behaviour of the fir tap. In this case a Synchronous object is the best choice of object type:

```
class FirTap extends Synchronous {
     int tap_constant;
     Wire  fir_input,  data_input,  mac_output,  data_output,
d0_d1;

     public FirTap extends Synchronous {
          int tap_constant;
          Wire d0_d1;

          /* The PRSocket is a Node, so it is
             the parent object. */
          super(p);

          tap_constant = constant;

          /* Now, get a pointer to all the wires
             that the interface to the static
             logic. The data_input wire was assigned
             port #0; etc. */
          data_input = p.inConnect(0);
          fir_input = p.inConnect(1);
          data_output = p.outConnect(2);
          fir_output = p.outConnect(3);

          inPort(data_input, 8); ...
     }

     /* Here, we describe the actual computation of each tap.
        This is executed by the HWSystem once per clock. */

     public void behaviour() {

          /* Multiply-accumulate the input value,
           and delay the input value. Wire values
           are read and written using the get() and
           put() methods, respectively. We pass a
           pointer to “this” with every access, which
           is used to help enforce ports directions
             (necessary for netlisting). */




          fir_output.put(this, tap_constant * do_d1.get(this)
                    + fir_input.get(this));
          data_out.put(this, d0_d1(this));
          d0_d1.put(this, data_in.get(this));
        }
}
```

JHDL provides many useful ideas for the development of both software based hardware description and reconfiguration integration within a software context. The constructor/destructor mechanism provides an useful technique to control configuration on the FPGA device. The actual usage of a current programming language is productive in the sense of the programmer/designer being already probably acquainted with the underlying syntax and basic conceptual framework behind the object-oriented paradigm implemented. All standard language features such as I/O are accessible to the programmer throughout the design process, both to the console and to files during software simulation. This contrasts remarkably with VHDL, for example. Some of these features are only available within simulation mode in JHDL but this is largely when such things would be necessary. It is useful, for instance, to be able to print the state of various objects during a run, for debugging purposes and the general status of an application at any given point. Graphical User Interfaces (GUI) can also be added without the need for any particularly complex programming or linking, utilising the Java API for such things. The only addition a programmer needs is the API and the specific library for the hardware used. The low-level interface with the device is masked by the native library, providing a generic interface for the higher level abstracted classes, which remains the same no matter what device is used.

These Java programs run and function while a hardware process is active, as JHDL allows the application to be divided into those parts that will run in software and those parts which will be hardware-oriented. Only those parts of the program describing circuits, in hardware execution mode, will be executed on the board. All other parts of the application remain on the host, operating essentially as a separate program, that is communicating with the user-designed circuitry, via the hardware device driver. JHDL therefore allows the software and hardware descriptions to both co-exist and co-execute.

The system improvements that are listed have better integration with utilising netlists to allow JHDL to function as a complete structural design tool and behavioural synthesis to allow circuits to be expressed at a higher level. A netlist format is required and a way of utilising further FPGA objects, such as commonly used circuits.

Some work has already been done to produce behavioural synthesis using another circuit class, `HWProcess`. The designer inherits from this class when behavioural synthesis is required, in a similar way to circuits that are defined using the `CL` and `Structural` classes. The `HWProcess` provides an additional method, `waitUntilClock()`, that the designer inserts into their behaviour methods. This is similar to a `wait()` in a VHDL process. In this way a designer can behaviourally express the circuits using a sub-set of Java statements and developing a circuit description that uses the common wait until clock idiom present in VHDL. The sub-set of Java statements will form the basis of providing a way back to VHDL via translation and further processing by VHDL synthesis tools. This can then be simulated with other circuits, which may be based on VHDL, in the normal way. This allows commonly available tools to be used, rather than specialised software developed for this purpose.

The lack of adequate communication with normal synthesis tools and also no way for access to utilise legacy circuits, is in some ways disabling. One possible way around this would be to develop a means of full netlist integration in some form which is commonly output by software tools. A requirement that was stated was that the system must use an existing language with no extensions. This makes the tool accessible to a wider range of programmers, in the sense that they only need the common Java compiler.

One of the main problems is the necessity to provide a new driver for each possible board in use, which may be difficult for the average user, bearing in mind that that the driver package must supply a generic point of communication at the interface. The requirements for this are quite daunting, as the code must obviously be based on native code with knowledge of low-level access to hardware, both on the host and separate board.

JHDL supports partial and global reconfiguration but not dynamically while the device is in “run-mode”. This means in practice that the device must be stopped before a circuit is downloaded, increasing the latency which occurs between configurations. These configurations become well-defined discrete “slices” rather than continuous interlaced events, which occur concurrently while processes are active.

A very useful aspect of this system is the ability to easily swap between hardware execution and simulation modes. No modifications to code are required and the switching itself between the two modes at run-time requires only the setting of a single boolean variable.

Normally design visualisation is an integral part of design methodologies in hardware engineering. For example, in VLSI computer-aided design, this visualisation provides circuit designers with an abstract view of design characteristics in a form appropriate for a particular design abstraction. The high-level representation hides the actual complexity in order to provide a model which is more manageable. This may be in the form of a data flow, or finite state machine graph.

### 2.1.5 Visualisation and Manipulation with FPGA Tools

Visualisation tools, such as schematic or state diagram editors, floor planners and layout editors, allow engineers to define a structure, connectivity and behaviour of a design and also to analyse design functionality, performance, feasibility, reliability and other characteristics. Designers of reconfigurable systems are required to analyse numerous temporal and spatial design properties, including; design latency, throughput, configuration time, spatial conflicts, sharing of reconfigurable resources in different configurations, impact of placement on FPGA configuration time, size of configuration data, power consumption, etc.

Most research into design techniques for reconfigurable systems focuses on algorithmic, methodological or design entry issues, where little work has been devoted to reconfigurable design space visualisation. Luk and Guo [20] describe a visualisation environment for reconfigurable libraries, which allows designers to track execution of pipelined computations. Some user interfaces of commercial FPGA tools, such as Atmel FPGA layout or Xilinx ChipScope, allow consideration of logic and routing resources in multiple configurations.

The way in which an FPGA design is developed, in both temporal and spatial dimensions, is explored in “Design Visualisation for Dynamically Reconfigurable Systems” by Milan Vasilko [45]. This is particularly interesting here because of the way in which multiple configurations, as frames in time, are composed and utilised.

The system they have developed named “DYNASTY” is a CAD environment to support research into novel design techniques for Dynamic Reconfigurable Logic (DRL) systems. The system, in effect, implements a temporal floorplanning based design methodology.

DYNASTY uses an integral design representation which combines models from behavioural, architectural and physical levels. The representation, which also has available design manipulation and visualisation tools, allows simultaneous DRL design-space exploration in both temporal and spatial dimensions.

The design tools included allow manipulation at multiple abstraction levels. The Design Browser tool allows a designer to analyse the design structure at behavioural, architectural (register transfer) and layout levels. A specific design model can be viewed graphically, using the respective model viewer. A variety of algorithms can be invoked from the Browser to perform transformations and calculations on the selected design object.

The Library Server Browser tool provides an interface between library servers and the designer. Each library server can include: cell and parametric module libraries, technology-specific design or estimation algorithms, target technology device models and other technology-specific characteristics. The Library Browser allows the contents of the library servers to be selected from the options available for the target technology.

In order to avoid unnecessary design iterations, the effects of design manipulation have to be indicated to the designer during the manipulation process. This system utilises visualisation to present all the necessary design information to the designer. Static design systems provide visualisation of many common design characteristics (e.g. structural information, overall timing and others).

In DYNASTY there is an attempt to visualise factors which are pertinent to dynamic reconfiguration such as: configuration partitioning, reconfiguration overhead effects, design execution and configuration schedule and finally, spatial conflicts (overlaps) between blocks in different configurations. DYNASTY, as far as was able, was made to be not technology-specific, in other words, various DRL systems were to be supported.

Configuration partitioning allows an input behavioural model to be split into sets representing individual configurations. By manipulating the design at this level, the relationship between the behavioural model elements and the resulting configuration partitions can be easily established.

Another factor which must be taken into account is the execution latency, which is dependent on the time necessary for the configuration of all design blocks. This is largely bound to the interface, which allows configuration to take place on the target reconfigurable device.

To allow for both of these, that is, behavioural and temporal perspectives, there are two alternative views in the DYNASTY Floorplanner; the configuration view and the system clock view. The configuration view represents partitioning of the design into configurations. Such a view is useful in the early design stages, when a designer needs to perform this partitioning on a behavioural design model. At this stage, only the configuration sequence with a design block granularity is determined, while the actual cycle-accurate execution schedule can be calculated at a later stage. The system clock view is a cycle-true display of the design activity. This view includes visualisation of both execution and configuration processes for all design blocks. The cycle-true schedule is recalculated by the library server as the design is manipulated.

Reconfiguration overheads are calculated by a technology-specific algorithm in the library server. The period of reconfiguration is actually displayed in the Schedule Editor window using a red bar-type element. The Floorplanner indicates the configuration of individual blocks in either a 3D or 2D view. The number of graphics representing a block in a vertical direction indicates the configuration latency as a number of clock cycles. This allows a designer to assess the configuration overheads for the current placement, partitioning, execution schedule and system/configuration clock period in a dynamically reconfigurable design.

Spatial conflicts (overlaps) are visualised in two ways. If the conflict was caused by manipulation outside the Floorplanner tool (e.g. by changing a library module allocated to a behavioural model element), the conflicting floorplan blocks are highlighted. The second way is by rejecting placements within the Floorplanner via on-line checking of data dependencies and block positions, which generate a spatial conflict while a designer is manipulating the design.

As there are interdepencies between the execution and configuration design scheduling, the visualisation of both schedules have been merged into a single Schedule Editor tool. This tool displays the overall execution schedule which combines the execution and the configuration latencies of the individual design blocks. The actual Schedule steps are identical to system clock cycles. If the configuration clock is different to the system clock, the configuration latencies are scaled to the system clock units. Data dependency conflicts between blocks are emphasised in the Schedule Editor window.

There are two viewing options; 3 dimensions and 2 dimensions. The 3D option provides a viewpoint which is better to represent overall design characteristics. The 2D Floorplanner view allows designers to examine each of the layers individually, also allowing detail to be seen in places which are hard to see in the 3D view. The 2D Floorplanner is also better suited for exploration of desired sharing between configuration layers, as a designer can display multiple layers to compare and examine their similarities.

An example is used to explore the functionality of the system. This example is a simple Laplace Filter operator used in image processing. The 3 x 3 Laplace operator implements the following computation:

$$gm_{(1,1)} = 4 \times i_{(1,1)} - (i_{(1,2)} + i_{(2,1)} + i_{(2,3)} + i_{(3,2)})$$

where (r,c) represent row and column pixel coordinates in the operator mask.

The operator is implemented on a resource limited FPGA architecture, the Xilinx XC6200. The size is so limited it does not allow the entire Laplace operator to implemented in a single configuration. The designer can instead, if limited to such a device – as in this example – choose to “fold” the data-flow computation over several configurations. Blocks derived from the behavioural model of the design are used to construct a 3D floorplan. The main objective at this point being to minimise the design execution latency. This is determined by the module execution latency and the configuration latency. While module execution latency is fixed for a given module type, the configuration latency can be reduced if module resources are to be shared among configurations. The designer needs to identify those design solutions where the configuration latency is minimised.

The design modules are firstly partitioned into individual configurations, the initial solution of which can be viewed using the 3D Floorplanner. Configuration overhead can then be minimised by the designer with a module placement, which would maximise resource sharing after the initial partitioning is achieved. Using the Schedule Editor, the actual execution latency can be measured and seen in the 3D floorplan, using the system clock view.

Another example given is a Pattern Matcher, showing how a designer can observe the reduction of the configuration time as a result of sharing reconfigurable logic resources. A comparator circuit is used as a simple pattern matcher. The varying match patterns for the comparator are analysed by the configuration estimation algorithm and indicated via the system clock view of the floorplanner that only one floorplan block needs to be reconfigured. This immediately provides the designer with information about the qualities of a placement. The example in question required only the change of one unit during such a reconfiguration; the addition of a NOT gate to the output of a flip-flop.

Using an open architecture of DYNASTY library servers, custom DRL architecture models can be implemented. Various DRL architectures can be visualised in combination with the visualisation capabilities, for a given set of behavioural problems. This will aid in the development of future application-specific reconfigurable architectures.

The DYNASTY system offers several tools and techniques for the designer of dynamically reconfigurable logic, with very useful and interesting features. The visualisation and immediate feedback of a design, particularly temporal, provide a means of optimising a device's resources. Rich visual design presentation allows easier analysis and understanding of a design's characteristics. This contributes to the reduction of the overall number of design iteration needed to find a suitable design solution. Using a visual method and interface of manipulating such a design allows direct and intuitive modification of design properties. By providing on-going analysis while manipulation takes place, various design decisions can be calculated and visualised instantly. The combination of DRL design visualisation and temporal floorplanning design methodology provides a rapid development route for such systems. The use of library server estimation algorithms, together with visualisation, means that numerous design alternatives can be evaluated in the early design stages, avoiding the need for time consuming iterations, through FPGA place and route tools that are commonly used.

This design tool holds much promise for the future by recognising the various dynamic elements of design connected with the FPGA. The paper also recognises future potential in using 3D abstraction within a virtual reality context and other interactive technologies. The main immediate benefit is the recognition of tools which allow an FPGAs configuration to be seen in terms of time and the use of shared resources by subsequent iterations or partial configurations.

### 2.1.6 Further Development of Tools

The method of floorplanning and the automatic design synthesis technique used in DYNASTY is outlined in “Automatic Temporal Floorplanning with Guaranteed Solution Feasibility” by Vasilko and Benyon-Tinker [46]. This paper describes a synthesis algorithm that, when given an input behavioural algorithm, a target technology library server and a set of design constraints, will generate a DRL design solution in the form of a 3D floorplan and a design schedule. The technique optimises the design solution in a multiple-objective design search space, while making realistic assumptions about the implementation reconfiguration overheads. It will also consider the use of partial reconfiguration if such a mechanism exists in the device being used. Multiple design objectives at various abstraction levels can be simultaneously considered. A realistic estimation of the reconfiguration overheads also guarantees the feasibility of the automatically generated solutions. While other methods of developing designs simplify the model of the target DRL technology, or ignore partial reconfiguration, this system utilises methods to explore the available design search space. An oversimplified approach that ignores the flexibility of such devices requires post-processing steps or numerous iterations to produce even a single feasible design solution. There are many problems involved with this: placement conflicts, routing problems, data dependency, violations due to prohibitive reconfiguration overheads and other such restrictions.

The configuration model used as an example is that of a reconfigurable system composed of several parts: the reconfigurable logic, a reconfigurable controller (which can be another FPGA or microprocessor) and a configuration data storage (such as a block of SRAM). While this model is a fairly standard architecture for the run-time system, the actual approach that is used is worth examining. Their assumption for the system are that: an input design problem is fully specified prior to the temporal floorplanning; a system (or its part) designed using the proposed method is non-reactive, i.e. its behaviour cannot be changed through external control signals – such an operation is typical for signal-processing applications; the target reconfigurable system is synchronous with one common system clock signal; the configuration controller is external to the dynamically reconfigurable logic array and all array resources are available for storage of the configuration data and the data shared between the configurations; data between blocks in different configurations is transferred either via registers, shared between configurations, or via the external memory. In the latter case, the data is stored and retrieved from the external memory as a part of the configuration process. A behavioural model of the design problem is input to the synthesis algorithm in the form of a Control/Data Flow Graph (CDFG). After the synthesis process is complete a design solution is generated which contains a 3D floorplan and an execution schedule.

The reconfiguration schedule is executed by the external reconfiguration controller, which controls the system execution and reconfiguration process itself. The 3D floorplan can be used to generate the configuration data needed for the implementation of the design in the specific DRL technology. The input design model has several tasks which are performed during the automatic temporal floorplanning: module allocation, scheduling, 3D floorplanning and cost evaluation. Module allocation consists of each node in a CDFG being assigned to a module selected from one of the libraries available in the technology library server. The scheduling task organises the execution period for each CDFG node within the overall design schedule. The 3D floorplanning stage allows design modules to be positioned in a 3D floorplan. Coordinates in the horizontal plane represent spatial positions of design modules. The vertical module coordinates provide a representation over time, which is measured in schedule control steps. A cost evaluation function is used to evaluate the overall quality of the generated solutions. This forces rejection of solutions violating either design constraints (overall latency, solution spatial size, or data-dependency violations), or target technology constraints (module spatial overlaps in a 3D floorplan, configuration dependency violation). Configuration latency is also recalculated as a part of the cost evaluation.

The search mechanism that is used to resolve the problem of the temporal, has to produce a design which has multiple design and technological constraints. A multiple-objective search space like this requires specific techniques. Genetic algorithms have proved successful for a variety of problems such as this, including those of traditional ASIC synthesis. Genetic algorithms model the natural process of evolution using mathematical models for natural selection, reproduction, cross-over and mutation. A genetic algorithm was adapted to the problem of temporal floorplanning. The main core of which was provided by the MIT Galib library. The approach taken was to use a steady state algorithm with tournament selection. A problem-specific genome, genetic operators, cost function and evolution control routine were developed for this problem.

The 3D floorplan is captured in a design solution represented as a data structure. This genome data structure provides links to the design data structures at a behavioural (CDFG), an architectural (register transfer) and a 3D layout level (netlist of blocks positioned in a 3D floorplan). Normally, simple mutation and cross-over operators are used in the generation of a solution using a GA. However, because there are various design characteristics over the entire solution, a number of problem-specific genetic operators have been developed, to manipulate different design characteristics, at different stages during the evolutionary process. The probability of the application of the operators is controlled by an evolution control strategy designed specifically for the algorithm.

A greedy "first come – first serve" allocation algorithm is performed on a CDFG model when a design solution is initialised, followed by a random placement of allocated design modules in a 3D floorplan. To guarantee that the pool of initial solutions is feasible, the initialisation procedure checks for data and configuration dependency violations.

The evolutionary process utilises two genetic operators: cross-over and mutation. Cross-over simulates mating between two parent solutions which produces two children solutions. In this problem scenario there were three cross-over operators in use: module allocation cross-over, random-sized 2D floorplan and random-sized 3D floorplan. The module allocation cross-over exchanges modules allocated to identical CDFG nodes in parent solutions. Random-sized 2D floorplan cross-over exchanges X-Y (i.e. horizontal only) positions between the two module groups in one floorplan layer. Random-sized 3D floorplan cross-over exchanges a randomly-sized group of modules between the two parent 3D floorplans. This copies the X-Y-Z positions of the selected modules (both horizontal and vertical placement).

The mutation operator simulates random changes to one or more individual solutions. There were several mutation operators designed: module allocation mutation, 2D floorplan mutation, 3D floorplan mutation and 3D floorplan "shaking". Module allocation mutation changes the module allocated to a given CDFG node to a module of a different type but with the same functionality (e.g. a ripple-carry adder can be swapped for a carry-lookahead adder). 2D Floorplan mutation changes X-Y coordinates of the selected module in a 2D floorplan layer. 3D floorplan mutation changes X-Y-Z coordinates of the selected module in a 3D floorplan layer. Finally, 3D floorplan "shaking" produces an effect of randomised "shaking" of the entire 3D floorplan. This is an essentially greedy algorithm which generates new X-Y coordinates for all randomly selected modules in the solution. This may lead to further compaction of the 3D floorplan.

The actual evolution control strategy is controlled by a core steady-state generic algorithm. There is also a monitoring algorithm implemented which controls the frequency of application of the selected genetic operators. These probabilities change dynamically in response to the population divergence during the course of the evolution. The control function and the individual probabilities can be altered by the designer.

The overall strategy is to apply operators which produce big changes in the design solution early during the evolution (when the population divergence is large). As the confidence in the generated solutions increases (observed as decreasing population divergence), the probability of the fine-tuning operators (e.g. 2D floorplan mutation) increases at the expense of operators producing big changes. The evolution terminates once a solution satisfying the design objectives is generated, or at a request from the designer.

A solution correction algorithm is invoked on a design solution after application of each genetic operator. This checks for data dependencies, evaluates configuration overhead and calculates the overall design latency. Dependency violations are resolved by gradual increase of the design latency until all such violations are removed.

The configuration time is calculated using a technology-specific algorithm from the target technology library server. Depending on the capabilities of the target technology, this algorithm may consider partial reconfiguration and resource sharing at fine granularity. A cost (fitness) function is used in the algorithm which evaluates the overall design execution latency. This is a composite metric measured in system clock cycles and includes both execution and reconfiguration delays. Other criteria, such as memory size required for the configuration storage, could also be incorporated in the cost function.

These methods were checked using several small design benchmarks. These had good manually-optimised solutions already known so they could be easily compared. A Xilinx XC6200 was used for the experiments, together with the DYNASTY system and its XC6200 technology library server. The XC6200 library server implemented in the DYNASTY system provides an estimation algorithm suitable for the calculation of the reconfiguration overhead in a given 3D floorplan. The algorithm uses a detailed model of the XC6200 architecture and its configuration interface.

The benchmarks used for the experiments were: HAL differential equation solver, Laplace transform spatial filter mask and the Pattern matcher comparator circuit. The HAL differential equation solver was used to test the algorithm in a scenario when a large problem needs to be mapped on an area-limited reconfigurable architecture. This is achieved by “folding” the algorithm execution across a series of configurations. The Laplace transform was used to test the case when sharing between two configurations has to be maximised in order to minimise the reconfiguration overhead. Two masks with different coefficients were used for this evaluation. Finally, the Pattern matcher comparator circuit was used to test whether the algorithm will be able to identify that configuration of only one logic cell needs to be modified in order to change between two versions of a circuit.

In the case of the Pattern matcher, an identical result to the manually-optimised design was achieved, leading to configuration overhead of only one clock cycle. The “folded” implementation of the Laplace transform and HAL differential equation gave average latency of the automatically generated results which were higher than for the manually-optimised solution by 19% and 27% respectively. Individual results approaching the qualities of the manually-optimised design were observed in many synthesis trials.

This result is similar to those in static systems where automatic tools for behavioural/RTL synthesis are used, in that, it is not possible to achieve results identical to those optimised manually. It is not possible to achieve efficiency identical to manually-optimised solutions in all but very simple cases.

The basis of the accuracy of the synthesis method also depends upon the accuracy of the estimation algorithms provided by the library server. The DYNASTY system allows designers to choose the type of estimation algorithm used for configuration overhead estimation. The designer can therefore select the desired tradeoffs between the algorithm execution time and the accuracy of the generated design solutions.

This automatic design tool can be used to synthesise a DRL design for a given in a very short period of time. Manual optimisation of designs such as the benchmarks above may take long periods of time; several hours or possibly days. The approach given with the DYNASTY framework can produce solutions with about 50% inefficiency within several minutes on a Pentium 450MHz PC. Even at this level the approximate result can be used to estimate the feasibility of the implementation of the design using the selected DRL technology. This gives an ability to perform trade-off analysis using different design constraints. The use of technology library servers rather than a generalised model for the target device, provides the algorithms used with a form of technology independence and realistic configuration overheads to be estimated.

### 2.1.7 Sequencing the Hardware

The Dynamic Instruction Set Processor (DISC) system described in "Sequencing Run-Time Reconfigurable Hardware with Software" by Wirthlin and Huchings, 1997 [50] is another system[15] which relies on the caching of designs and run-time reconfiguration of FPGAs. The idea here being to exploit the advantages of RTR while limiting the disadvantages involved in temporal partitioning and the high time penalty required for reconfiguration at run-time.

[15] There are a few more, such as the Chimaera system [13] and MATRIX [32].

DISC is a run-time reconfigured processor that provides a convenient method of sequencing application-specific hardware. DISC also allows the caching of hardware modules to reduce the overhead of frequently used hardware modules. The central concept behind the system is to provide the simplicity and flexibility of conventional software design and the performance of application-specific hardware.

The way the performance reaches the level of application-specific hardware is by allowing user-defined application specific instructions to supplement a conventional instruction set. The instructions, which are user-defined, are designed in hardware to exploit the low-level parallelism, custom control and specialised I/O of custom hardware circuits. When designed in such a way, a hardware instruction module can execute an application-specific function many times faster than a sequential stream of general purpose instructions. An optimised custom-hardware instruction module replaces the long series of conventional instructions and therefore eliminates the instruction fetch, decode cycle and other such overheads.

When designing application-specific functions as instruction modules, hardware circuits operate under software control. This software control of application-specific instructions allows DISC to retain the programmability of a conventional processor, while preserving the performance of custom hardware. Conventional general purpose instructions can be described and mixed with complex sequences of custom instructions.

Resources on the FPGA can be altered as demanded by run-time conditions, thus removing certain hardware limitations. The DISC system is implemented on partially reconfigurable CLAy FPGAs, allowing the instructions to be paged in and out by partially reconfiguring the FPGA, as demanded by the executing program. An arbitrary number of performance-enhancing application-specific instructions can be implemented by re-using FPGA resources.

The processor core and the custom-instruction space form the two regions within the DISC processor. The processor core handles the instruction sequencing and remains static on the hardware during program execution. The instruction space is reserved for hardware instruction modules and is open to continual reconfiguration as run-time execution demands.

The main part of the system, the processor core, was designed within the FPGA resources to sequence program instructions, interface with external memory, synchronise instruction swapping and control inter-module communication. This core is based on a simple accumulator model. All loaded instruction modules and the processor core itself have access to the global accumulator register and several state signals. Dedicated I/O Interfacing logic, basic sequencing capability and addressing control are all contained within the processor core. Several simple sequencing and internal control commands are available which are statically wired into the processor, which, as already stated, is never reconfigured.

A large proportion of the area within the DISC FPGA is reserved for custom instruction space. Within this space such instructions are continually configured and manipulated as demanded by the application program. Any custom instructions which are in frequent use are kept configured within the custom instruction space to eliminate the configuration overhead when reused.

Custom instruction modules can be placed fairly freely within the instruction space, allowing the placement to be determined at run-time and not at design-time. This maximises hardware efficiency by allowing module placement to be determined by run-time conditions.

To allow instruction modules to operate correctly over the surface of the FPGA they must be designed in a very specific way. The interface between the global circuit and the instruction module must appear the same at every location. This problem is solved by constructing a simple but regular global context throughout the FPGA. The way this is done is by running global processor signals vertically on the FPGA and spreading these signals along the width of the FPGA.

DISC instructions are then designed under the strict constraints of this global context. Instruction modules are designed horizontally across the width of the FPGA. To gain access to each of the processor signals, regardless of their vertical placement, the modules lie perpendicular to the communication signals. An instruction module must span the entire width of the FPGA but can consume any amount of hardware by varying its height.

The initial version of DISC was built and tested on a wirewrap prototype board with a single FPGA. This was not a particularly large device and left little room for both the processor core and unallocated instruction hardware space resources. When first developed in this way, the DISC processor core was 8-bit and provided few built-in instructions, with a limited addressing range. The system board was also unreliable.

A second system was developed utilising the DISC architecture, which answered many problems that the first attempt had highlighted as a "proof-of-concept". The new board was developed on a stable FPGA board provided by National Semiconductor, offering several advantages over the board used earlier. This board provided a dedicated host interface, additional memory and three partially reconfigurable FPGAs. More FPGAs allowed a natural partitioning of system functions to take place. Static resources were moved away from the custom instruction space to maximise dynamic hardware. The three main system units were therefore: the processor core; the Bus interface and Configuration controller; and finally the Custom instruction space. There was also the system memory connected to each, as well as the host bus. The Bus Interface remains static throughout its operation and actively monitors the processor state and configures modules on the custom instruction space, as provided by the host.

The processor core was now separate from the instruction space and sequenced the instruction modules as dictated by the program in memory. As the processor has more resources available the actual design itself was improved with a 16-bit bus to match the bus width available on the board. The new core also took advantage of other available resources and had extra registers to provide more support for function calls, recursion and high-level language addressing.

Applications to be designed for DISC require software and hardware development. The application-specific modules must be developed using hardware tools and environments, while the program that sequences their execution is written using normal software environments. The partitioning of hardware and software is done depending upon the nature of the task. Those operations of an application which can exploit circuit level parallelism, custom I/O interfaces, or hardwired control, are implemented as custom hardware. Whereas Global control, complex control sequences and complex data structures are usually implemented using software.

The instruction modules, which are built in a hardware environment, must be developed with the constraints imposed by the Global context. To make the development process easier structural models of the DISC system were made available to ensure the user defined modules meet the global context protocols. The next stage is then to map the instruction module to DISC using device-specific mapping tools. Various post-processing utilities were written to insure that the module adheres to the global context constraints. The module, once designed and mapped, can be then used at any point by the DISC system.

The sequencing of an application is done through software which organises the instruction modules to execute in a specific order. The complexity of such a program can range from a simple assembly language type to a full blown C program, which allows greater manipulation and references to instruction modules. The way in which the instruction set of the processor changes means that the assembler must allow the user to define new instructions in application programs. A retargetable assembler was made for DISC, which allows the user to do this as they are developed. By specifying the instruction mnemonic, instruction type, opcode and any necessary control parameters, it is possible to define a new instruction.

A retargetable C compiler called LCC was targeted to the standard DISC instruction set and additional syntax was added to the compiler to support custom instruction calls from within the C language. Familiar C control structures and high performance instructions can be mixed within programs for the compiler. The way in which custom instruction modules are referenced within C is by a function call with a native_ prefix as follows, such as:

```
native_lowpass(a);
```

The corresponding instruction name is substituted for all native_ function calls by post-processing after compilation.

| Module | Size in Rows |
|---|---|
| Image inversion | 4 |
| Image move (copy) | 4 |
| Image clear | 5 |
| Image threshold | 7 |
| Histogram generation | 8 |
| Dilate and Erode | 7 |
| Image Difference | 7 |
| Median pixel | 9 |
| Skeletonisation | 25 |
| Low-pass filter | 28 |
| Edge detection filter | 30 |
| High-pass filter | 42 |

**Table 2. Instruction modules**

An example demo of use is given for an image processing using the DISC system. The first step here was to develop an instruction library for image processing. The main aims here are to show how the system works, its improvements over conventional methods and the ability to easily sequence DISC custom instructions. The operations involved all obtain significant speed improvements through parallel computation, custom memory addressing and pipelined arithmetic. Each instruction operates at the DISC execution speed of 8 MHz. Table 2 shows the instruction modules that were available.

By way of demonstration, an object thinning algorithm was implemented on DISC, which uses the above library and high-level language support. The algorithm is based on the following three operations: pre-filtering, thresholding and region thinning. The code is as follows:

```
Main() {
     image *image1, *image2;
     histogram *hist;
     int thresh;
     int skel;

     image2 = native_lowpass(image1);
     image1 = native_clear(image2);

     hist = native_histogram(image2);

     thresh = peakthresh(hist);

     image1 = native_threshold(thresh, image2);

     for (skel = 0; skel != 0; ) {
          skel = native_skeleton1(image1, image2);
          skel += native_skeleton2(image2, image1);
     }
}
```

This algorithm is intended to operate on high-contrast, grey scale images such as news print, hand writing and lettering. Firstly, the image is passed through a simple low-pass filter to remove high-frequency noise that may introduce unwanted regions. The instruction for this operation is the `LOWPASS` instruction, the result of this when applied to an image is that the original loses some definition, softening and somewhat blurring the image. The C code for this is:

```
image2 = native_lowpass(image1);
```

Thresholding is the next stage after high frequency content has been removed. A thresholding algorithm works by pulling an object of interest to the foreground, while reducing the visibility (placing in the background) all other information content. The most simple way of achieving this is to compare each pixel value in the image with a threshold value. Pixels with an intensity greater than the threshold value are placed in the foreground, while those less defined are placed in the background.

This is a simple process but in practice the actual threshold value can be difficult to ascertain. The process of choosing a threshold value involves initially obtaining a histogram of the image. Simply finding the mean or median of the histogram is enough. These approaches are inadequate for high contrast images with a dominant foreground or background. A better way of doing this is to detect peaks within the histogram, a good threshold value may be obtained by calculating the mid-point between the foreground peak and the background peak.

The library of instruction modules can be used for the thresholding process and therefore completed with high speed custom hardware. A histogram table can be made for an input image using `CLEAR` and `HISTOGRAM` instructions. The `CLEAR` instruction sets up and initialises the histogram memory and the `HISTOGRAM` instruction efficiently builds the image histogram. No instruction is available for determining histogram peaks or finding the mid-peak threshold value, this can be instead written in software as C or assembly language. The `THRESHOLD` instruction module will convert the original source image to a new binary image using any value for the threshold that has been arrived at using the software routine. The following C code is used to further complete the process of thresholding of an image:

```
image2 = native_lowpass(image1);
image2 = native_clear(image2);
hist = native_histogram(image2);

thresh = peakthresh(hist);

image1 = native_threshold(thresh, image2);
```

This code uses four custom hardware modules and one software routine (`peakthresh`).

An object-thinning algorithm is then applied to the image. Object-thinning is the ability to derive the basic shape of an image, losing all redundant shape information. This, in practice, leaves a simplified image which represents the basic shape of an object within the picture as a “skeleton” of single pixel lines.

The algorithm employed for this thinning process uses successive removal of the outer edges of an object until only the skeleton remains. This is similar to another operation, known as erosion. However, where this uniformly erodes the outer edges of an image object, leaving sometimes nothing, the thinning routine will never completely destroy an object or disconnect object regions. The algorithm works within the following constraints:

- Connected object regions must thin to connected line structures.
- Approximate end-line locations should be maintained.
- The skeleton should be one pixel wide.
- The skeleton should lie at the centre of the object cross-section.
- The skeleton must contain the same number of connected regions as the original image.

There is a degree of trade-off that can be used between skeleton quality and algorithm speed. In this case the Zhang-Suen thinning algorithm was used because of its suitability for implementation with hardware. The algorithm works by determining whether a pixel can be removed by evaluating its neighbourhood.

In this example, a custom instruction module, `skeletonize`, was developed to implement the algorithm with a height of 25 rows in the DISC. Each execution of an instruction module completes one pass of the thinning algorithm and indicates whether an operation is finished. The instruction module must therefore be executed until no more modifications are capable of being made to the image:

```
for (skel = 0; skel !=0; ) {

     Skel = native_skeleton1(image1, image2);
     Skel += native_skeleton2(image2, image1);

}
```

The run-time execution of the thinning algorithm therefore depends upon five custom image processing instructions; `LOWPASS`, `CLEAR`, `HISTOGRAM`, `THRESHOLD` and `SKELETON` and three other instructions that were used in support of the C code; `SHIFT`, `COMPARE` and `ADD`. The amount of resources available does not match the amount required by all of these hardware modules, therefore there must be some swapping of modules at run-time. This must be while the algorithm progresses.

The execution sequence is now described. The clear instruction space is first loaded with the custom instruction modules; `LOWPASS`, `CLEAR` and `HISTOGRAM` which are configured and executed. After the histogram has finished executing the software routine is called, which determines the threshold value. This C code routine uses several instruction modules not built into the global controller; `SHIFT`, `COMPARE` and `ADD`. As the spare resource are all used by modules, the system must make room for the instructions to be used, which help the C code. The least recently used module, `LOWPASS`, is removed and replaced by the required unit. The next stage, once the threshold value is computed, is to produce the binary image. This uses the `THRESHOLD` instruction module which must be loaded into the spare space still available after the large `LOWPASS` module was removed. The process continues with thinning, the skeletonisation instruction. This module will not fit in the space available so the two oldest instructions are removed, `CLEAR` and `HISTOGRAM`. The `SKELETON` module is then loaded and iteratively executed until object thinning is completed.

A comparison is provided between the DISC system and a 66 MHz 486 PC, running the object-thinning algorithm:

| IMAGE | 486 | DISC | Speedup |
|---|---|---|---|
| Silk Screen | 2.17s | .29s | 6.5 |
| Block letter | 3.90s | .53s | 6.4 |
| News print | 9.90s | .89s | 10.1 |

**Table 3. Hardware and software in comparison**

A great deal of the time within the DISC system is actually spent reconfiguring the system and moving the modules. For example, in the silk screen image, over 25% of the execution time is spent configuring DISC and moving the instruction modules. An interesting point also to note here is that the DISC clock is running at approximately $^1\!/_8\,th$ the speed of the 486 PC.

One of the main advantages of the system is the ability to mix C code and hardware as “instruction modules”. This adds a great deal of flexibility to the development process. Designers can utilise the benefits of each domain (software or hardware), depending on the nature of the problem. There is also the aspect of development time, as software can quickly replace or add to a design and then be swapped later with a hardware module, if it is more appropriate. Software can easily be used to replace complex control and functions that are not performance sensitive.

The hardware modules can be reused within the system by any application. This reuse is determined by the ability of the module to fit within the global constraint which is applied to the FPGA. In other words, the module utilises a particular interface in which to "lock" into the surrounding circuitry. A downside here is that the module must be the width of the FPGA, although it can be any number of rows in height. This does stop the need for constant place and routing of such modules but creates inflexibility in placement to some degree. It allows large libraries to be developed for use in applications which can reference them within C programs. The DISC system, together with the global context, take care of all other concerns. FPGA resources are automatically reused after modules are lifted back off the FPGA device.

Another aspect of the DISC system is its extensibility, which allows applications, such as the example object thinning program, to be combined with more complex image processing algorithms. Additional programming, for example, could turn the object thinning application into a complete object recognition system, with as much complexity as required for a given task.

### 2.1.8 A Dynamic Run-Time System

A dynamic reconfiguration run-time system was proposed by Jim Burns et al. in the 1997 paper "A Dynamic Reconfiguration Run-Time System" [6]. The idea here was to provide a system which allowed a non-specialised approach to run-time reconfiguration. Past attempts had involved ad hoc combinations of software and hardware, the software being only viable on that particular system set-up. The paper presents the design of an extensible run-time system for managing the dynamic reconfiguration of FPGAs, motivated centrally by this universal approach. The system detailed is called RAGE (after the group that built it; Reconfigurable Architecture Group) and incorporates operating-system style services that permit sophisticated and high-level operations on circuits.

The system was born out of the need for a common platform to develop upon, rather than, as usually has been the case, the development of a different run-time system when a new application is built. Such applications have highlighted the need for a more complex run-time system. A common set of requirements were extracted from several applications. This formed the basis for the design of a proposed, core, run-time system, able to support any of them. It is easy to see here the parallels between this and the conventional operating system, as the techniques applicable to manage resources, such as memory and the CPU, are equally applicable to FPGA resources.

A number of case studies are provided to describe how the various components of the system function. The first example is the sub-project FPGA/PS, which aims to exploit the dynamic reconfigurability of FPGAs, to accelerate PostScript® rendering. The rendering of graphics for high quality printing (thousands of dots per inch), which is usually time consuming and slow, is accelerated by circuits realised on an Xilinx XC6200-based FPGA. This is a co-processing task, as the bulk of PostScript rendering is still performed on the host computer. When a different rendering operation is required (such as line or arc rendering), it is dynamically swapped onto the FPGA. This technique of swapping circuits on and off the FPGA is labelled here as "virtual hardware". This technique obviously has applications beyond the example given here and is relevant to a much wider class of problems.[16]

A collection of pre-routed and placed circuits reside in the main memory of the host system, ready for rapid download onto the FPGA over the PCI bus. At pre-defined intervals during the rendering, data and operation codes are passed as parameters to the virtual hardware run-time system. The system then swaps an appropriate circuit onto the FPGA and also preloads circuit input registers, with values passed from the application.

When results are available on the FPGA, the application on the host is interrupted, allowing data to be accessed and relevant material to be manipulated and moved. Only one working circuit is on the FPGA at a time.

[16] Other examples exist of utilising co-processing reconfigurable logic with software applications graphical [40, 24, 33] and database related [19].

Memory mapping between the FPGA, the RAGE system and the FPGA/PS application can be eased by organising the circuits in the following ways:

1) The inclusion of some static circuits on the FPGA, which serve as input and output ports for the dynamic circuitry. A circle rendering circuit needs (x, y) coordinates for the centre and a radius value, for example. These static input registers would be re-used when the next circuit is swapped in. This technique reduces the number of cells to be configured but makes circuit placing less flexible.
2) The input and output ports of the circuit representation are tied to the working circuit itself and the circuit is swapped onto a predetermined location on the FPGA.
3) The circuits are placed anywhere within known boundaries and RAGE organises the mapping of circuit registers into host memory transparently. The application is aware of these known addresses.

Where circuits are in a state which needs no run-time place and route, they can be represented simply as address/data pairs which form the FPGA programming information. In this instance there is no need for an internal representation of the FPGA device itself because there is no place and route required. In cases 2 and 3 above, a more complex swapping technique is required and consequently a full representation of the device's resources needs to be maintained. This includes all resources, such as muxes and configuration memory. The actual representation of the circuits needs to be a great deal richer too, as there is a necessity for knowledge of ports, hierarchical circuit blocks etc.

The circuit swapping system has access to a library of circuits and passes the appropriate programming data to the device driver. During the period of swapping it is important that the behaviour of the system is predictable. Methods of achieving this at such an important point involve either isolating the on-chip static circuitry, or by being aware that at this time there is the possibility of spurious results being produced.

By utilising the nature of some circuit inputs to remain static for long periods of time, relative to other inputs, partial evaluation reduces the resource requirements of circuits. Partial evaluation can be best represented by the following example. A circuit, such as the one commonly found in digital signal processing applications, multiples a stream of numbers by a constant. The constant value would normally be given as an input to the circuit, because while it is a constant number for a particular stream, it may change for another. The number therefore needs routing through the circuit and a general multiplier is implemented. The multiplier can be specialised for a particular number and the routing resources required to feed in the constant are freed. The components which supported the static method of supplying the number can then be removed or utilised in some other way.

This technique is already used in software compilation and execution, particularly in functional programming. Within this area the source text is analysed for static expressions, which can be evaluated at compile time. In the case of hardware synthesis and compilation, there is a traditional flow from the hardware description to implementation, which involves going through hardware compilers and generally lengthy place and route phases where the gates of the compiled description are arranged according to the available FPGA resources. This process makes it difficult to do such work at run-time, when a constant value is changed. This necessitates that the work of producing specialised circuits should be moved to the initial compile-time rather than run-time.

A way to do this is by symbolic partial evaluation, where such inputs are tagged as static but their values are not defined. There inputs can then be traced through the circuit to see which components of the circuit are functionally dependant on them. Case analysis of these components can be used to determine the different possibilities for the different inputs. This involves building a tree of possible circuit components. The appropriate leaves of the tree will yield the components of a specialised circuit for any particular values of the partially static inputs. By pre-placing and pre-routing the components at the leaves of the tree, the specialised circuits can be quickly built at run-time. If components are already resident in the appropriate combination, all that needs to be done is routing between them.

The actual generation of such a tree is not a particularly straight forward task. The different branches of the tree must be interchangeable with each other and therefore have similar FPGA footprints. The hardware description language, Ruby, allows not only to encode the behaviour of the circuit but the layout too. Overall layout is retained as much as possible and areas of reconfiguration localised by the symbolic partial evaluation process which transforms the specifications. The transformed specification trees must be placed and routed to produce a tree of FPGA-level components, which have foot prints that allow them to be interchanged. No automatic tools exist that can do such directed place and routes but existing tools which can do partial place and routes can be utilised by using various placement constraints.

A large proportion of reconfigurable circuits can have areas which are functionally static. By making the reconfigurable parts of the circuit well-defined regions and effecting changes solely in these areas, the functionality of the overall circuit is modified within some restricted overall area of operation . An example of this is a multiplier does not turn into a VRAM circuit but a *3 multiplier may become a *7 multiplier. This implies a reconfigurability at a fine grain gate level, as opposed to the level of reconfigurability applied to the FPGA/PS project.

When using such a fine grain viewpoint of reconfiguration, it is important to have within the high-level representation of the circuit a way of symbolising areas of reconfigurability. This allows the appropriate circuits to be available for insertion into the area. These areas can then also be referred to by their symbolic name. By coupling the symbolic reference with the circuit it is to be substituted with, the reconfiguration can be appropriately managed.

A CAL file can be used for the circuit representation. This contains a circuit representation at a low/device programming interface level, which must be given over to the system with a higher level symbolic representation of the circuit itself. The system then generates or imposes a layer to map between these two, effectively fusing them together. This mapping must allow the process to be inverted between the two representations, as the circuit can then be modified if needed. This creates a dataflow which flows counter to the direction of implementation, allowing configuration data and results to flow back to the application. It is possible that the top level application may require, for example, knowledge of the transformed state of its submitted circuit. A circuit may be transformed and applied to the device matrix but in this form may fail to fulfil application specific criteria. The RAGE system does not go this far but could be implemented at other levels.

There is also the possibility that CAL and symbolic representations could be optimised by the application only submitting a symbolic representation of a pre-placed and routed circuit and allow the lower level system to generate the programming stream circuit representation. Such a stream representation can be generated relatively quickly. This approach can be used to reduce the complexity of the application interface from the user's point of view.

The paper presents another example of use through a real world design for a systolic FIR filter. An *n*-tap systolic implementation of a finite impulse response (FIR) filter is considered, the basic structure of which is a series of processing elements, interconnected in a regular, linear fashion. The systolic FIR that is presented utilises Kean's inherently reconfigurable multiplier design. Kean's multiplier relies on the ability to split an *m*-bit multiplication (with a 2*m* bit result) into *m*/2-bit multiplications. Look-up tables are used to perform subsequent multiplications, each LUT configured to produce the multiplied value of the free coefficient by the upper and lower halves of the constant coefficient. The two partial results are reconciled by a 16 bit adder circuit, to produce the 2*m*-bit result.

Such LUTs can be made to easily reconfigure dynamically. Every cell in columns 0, 2, 4 and 6 is a 2 input to 1 output gate, which encodes four columns of the truth table of a single-bit multiplier. The values for these gates can be worked out at run-time by the host for a given 12-bit coefficient. A total of 96 cells need to be reconfigured, 48 cells for each 4-bit by 12-bit multiplier. A single byte defines the functionality of an individual cell. In a single configuration cycle four such elements can be passed, the worst possible case scenario is 24. If a 33MHz programming interface was used, an entire LUT could be configured in 1.45µs, including set-up overheads.

An XC6200 architecture was used to realise the design of the systolic array because of the way in which the LUTs can be configured simply in such a "divide and conquer" *m*-bit multiplication. To facilate the addition of partial results (a 12-bit adder appears in columns 7 and 8), the reconfigurable LUTs are vertically interleaved (columns 0 to 6). The result of the multiplication is accumulated with the processing element's (PE) incoming y value by a 16-bit adder (columns 9 and 10).

The paper further describes how the proposed run-time system will support the inherently reconfigurable systolic array. Template systolic array symbolic circuits are submitted by the application levels. An initial or default set of taps are preconfigured in a design. The system, on receiving such a design, begins the process of making the circuit on the FPGA matrix. It is the job of the Virtual Hardware (VH) Manager to ensure the co-existence of the new circuit with any that are already present. It is possible at this point that the array may be transformed and in effect denied its default configuration, location or both. The system in charge of any corrections, or necessary modifications to the circuit topology, is the Transformation Services, which is informed by the VH Manager. The Transformation Manager has a full range of geometric and replacement or re-routing facilities, which can be used in this venture. Using the configuration manager the array can be made resident on the device, under the instruction of the VH Manager. This rapidly converts the symbolic circuit representation to a configuration stream, which is communicated to the low-level device driver, interacting directly with the hardware.

Areas of the submitted circuit, represented symbolically, can be specified by request from the application for reconfiguration. Transformation services are not required for fine grain elements of reconfiguration, such as resources at gate level, including LUTs. Functionality can be updated in subsequent write cycles. At courser grain levels some degree of transforming of circuit designs may need to be applied, as it would be unusual that a circuit supplied from the application level would be immediately usable for reconfiguration if it has been symbolically transformed in some way.

To continue with the example; if an array has been made resident on the device, all that is left to do is to be able to communicate with it, such as the supply and recovery of data. Information which is toward the array, that is, to fill or alter it, can be treated as a form of reconfiguration. At the time of initial reconfiguration there is default data supplied within the circuit design itself within registers. In other words, as there are topology reconfigurations, so there must be register state reconfigurations. A form of mapping to such registers can be utilised if the symbolic representation within the run-time system is rich enough to allow it. It can, for example, group registers at the symbolic level to form larger units, which may be configured together. This symbolic register identification is interpreted in the VH manager and effects change to the register's state on the device, via the configuration manager.

The other direction of data transfer, from the device to application, can also be done symbolically. An application holds and is aware of a symbolic representation of its transformed design, it may also enquire what the value of a symbolically named register is. Using the VH Manager such requests can be mapped back to the actual configuration bitstream. The VH Manager interprets the symbolic identifier and directs the configuration manager to acquire the current state within the actual device configuration. The value can then be passed up through the VH Manager to the application level, for use by the application.

There is the possibility suggested here that circuits could actively seek communication with the application levels. The means suggested is through a exception mechanism which allows the on-chip circuits to raise, possibly prioritised, exceptions in the application levels. Register regions could be identified as exception flags, which are represented at higher levels, symbolically and therefore functionality at those levels could be added.

In this example the objective was to provide a high-level interface to applications that wish to perform complex reconfiguration and circuit manipulation tasks. In this way it acts similarly to an operating system for the FPGA.

The following basic requirements have been derived from this activity:

- The ability to reconfigure an FPGA and access board-level (e.g. clocks), without directly communicating with a device-driver.

- The ability to reserve a chunk of FPGA resource. This allows one FPGA to be shared amongst several tasks and applications.
- The ability to transform a circuit e.g. change it's orientation or translate it's position.
- A rich high-level symbolic representation for circuits, retaining the ability to quickly effect changes in the representation and transform the representation in device-specific programming data.
- An architectural correct representation of the FPGA device to support core algorithmic operations within various entities of the proposed system.

In overview, the RAGE system consists of several main units. The virtual hardware manager is the central hub of the system, coordinating the execution of the other system components. The device driver hides the programming interface of the FPGA and PCI board and presents a low-level foundation on which to build more complex functionality. Its main tasks are to create a mapping of the connected board's I/O ports and FPGA's configuration into host memory, enforcing mutual exclusion. The configuration memory is also mapped into the host address space. Another major unit is the configuration manager, which acts as a link between the virtual hardware manager and the device driver. The configuration manager can produce a programming stream, which is then mapped transparently onto the memory of the FPGA.

This proposed system is designed initially to work with the Xilinx XC6200 FPGAs. This is somewhat limiting but could be easily expanded to include other types, due to the fairly similar nature of such devices in terms of programming. The nature of the design also allows low-level system-specific detail to be decoupled and other sub-systems have been developed to ensure that the services provided make sense in other SRAM based FPGAs. It is believed that through the practical experience gained with one specific FPGA that this may be transferred to a more generalised version, which could easily be adapted to such devices as the Atmel 6000 series or National Semiconductor CLAy FPGAs.

The XC6200 series of FPGAs are based on a grid of cells (48x48, 64x64, 96x96, or 128x128), each of which can realise routing, any one/two input logic function or a 2-to-1 multiplexor. These functions can be combined with a register. Cells can communicate with neighbours, or to cells further away, using a hierarchical routing system based on blocks of size 4, 16 and 64. Individual bits of configuration memory can be reprogrammed. In one 32-bit write cycle it is possible to program 4 cells, although regularity in a design would allow more to be written. The programming interface runs at 20MHz. A key feature here on the XC6200 series is the ability to read and write directly to cells configured as registers on the FPGA, this is exploited on the system. It is possible to read locations without having to wire up I/O pads. For example, using the system it becomes possible to support a powerful location transparency with which to realise the circuits.

The main interface between the application and the run-time system is the virtual hardware (VH) manager. The circuit is made resident, communicated with and its resources freed by requests to the manager sub-system. A typical sequence is given:

- Application submits circuit in an external representation to the VH manager, which converts this into an internal circuit data structure, to be put into the circuit store.
- Application requests circuit to be made resident.
- VH manager checks current resource utilisation and finds a location for the circuit, possibly after having to pass it to the transformation manager for translation.
- VH manager passes the, possibly transformed, circuit to the configuration manager for conversion to programming data and download to FPGA, via device driver.
- Application provides circuit input data to VH manager and requests execution.
- VH manager downloads the data to the circuit, sets up interrupt handlers for completion and signals the circuit to begin execution (all through the configuration manager/device driver).
- On completion interrupt, the VH manager reads back the result data and passes it to the application.

- Application then expresses that the signal is no longer required.
- VH manager marks circuit resources as free and updates the configuration manager.

While these events occur it is quite possible that the application will make other requests for partial reconfiguration or download of circuits. These requests are all down at a high-level, using symbolic forms for components and circuits. The high-level circuit blocks are used to map resource usage, maintained by the VH manager, while the configuration manager is actually relied upon to map these to actual cells on the FPGA. An example of this is that the VH manager might refer to a register of a circuit as `counter.preload`, whereas the configuration manager may refer to the same area as cells 4-20 of column 12 on the FPGA device.

The circuit supplied to the system may not always be suitable for residence on the reconfigurable device. The device matrix shared between circuits in this way means that sometimes a circuits default position will be taken by another. It is necessary therefore that circuits themselves must become flexible, so the system itself must be able to transform the circuit, so it can become resident in some other section. This is the task of the Transformation Manager. The transformation system allows a circuit to be manipulated in two dimensional space, using a set of primitive operations. It is possible to perform standard 2D geometric transformations, such as rotation, mirroring and scaling. If the shape of an area does not match the shape of the circuit, visualised as simple polygons, then the shape must be altered in some way. By allowing rotation and translation, it is possible that the vacant space can be used. Without this, much space would be wasted on the FPGA surface and it would be impossible to place many circuits at a time. The only other alternative would be to deny service to the application, or allow the system to remove resident circuits from the FPGA. Taking off resident circuits is sometimes actually better in terms of resources than trying to transform a circuit. This must be weighed against whether it is worth the circuit being interrupted from its task and is purely a question of cost/benefit break-even points. Transformations are, however, more attractive in some respects and far less disruptive.

The need for some kind of transformation mechanism within the context of the system has been established. After transformation the symbolic circuit representation is translated into programming information for a specific FPGA device. The transformations must therefore act on the representation, which is a symbolic data structure capable eventually of routing a physical circuit. In this case a form of netlist is used, which is capable of explicitly defining routing and placing positions. The transformations may need some degree of routing and placing information at circuit compile time. The Xilinx XC6200 resources are generally regular, although there are some features which are asymmetric. During compile time the circuit generated relies on a very particular placement of routing and the availability of specific resources, as well as the overall features of the devices. Any transformations that take place at run-time could damage the givens that were taken into consideration at compile-time. The circuit would more than likely fail its intended purpose and functionality if such a situation was not corrected. As well as this, a malfunctioning circuit could quite possibly interfere with the functionality of adjacent circuits, as routing may cross boundaries at certain points. This particular task of ascertaining and constraining the footprint is done by the VH manager.

Re-routing of the circuits at run-time in this system is seen as necessary, mainly due to the nature of the architecture aboard the Xilinx XC6200. If a circuit, for example, of dimensions 3x4 is located at (0,0) on the device, then translated to position (1,1), its internal routing would have to be routed across a series of switching matrices that exist on the boundaries of every group of 4x4 cells. This particular and novel architecture is specific to the XC6200. From this description it is clear that the internal routing details of the circuit would have to be adjusted.

The regularity of the device is also punctuated by certain asymmetric features which must be taken care of. An example of this in the XC6200 is a series of hidden inverting multiplexors. Due to their irregular positioning it would not be possible to reproduce a circuit designed for position (10,20) to be placed at (21,10) as the features available at these two positions differ. The answer in this system was to introduce new components into the circuit, this may have knock-on effects on the placement and availability of resources for other components.

Three possibilities are cited to deal with these problems, relying on the ability of the system to detect when a circuit transformation will invalidate the circuit. The first way of dealing with such problems is simply to deny requests for transformation at a position where its symbolic form is translated, where it could not mirror its original design. This seems to be a rather inflexible way of dealing with the problem but the system could work within this idea if the circuit could only be positioned where it was capable of being routed in its current form, that is, without the use of transformation services. This method would allow the functionality of the circuit to be extended at the cost of losing the granularity of placement and due to this, a degree of flexibility. Any transformation services could be denied in this way or selectively dependent on specific resource requirements and alignment on the device itself.

Another approach is to actively try and re-route the circuit in the new areas. This would involve replacing components too, which may be in the original design but not in the placement area, to overcome any clash in requirements. Within this system set-up described here, such work re-routing and re-placing may be unfeasibly slow. An approach suggested here is to localise changes by a bounded runtime place and route. Such a mechanism localises and minimises change to the circuit and therefore reduces the time taken to place. A suggested way of reducing duration of runtime fitting on the reconfigurable source is to abandon the search for an optimal solution to placement or routing for a specific circuit. This reduction down to only the transformational necessities of placement itself simplifies activities at run-time and results in faster operation. In addition to this it may be possible to gradually evolve a better routed and placed circuit during the course of the application's lifespan itself, rather than when the circuit is first initialised.

In a conventional computer system, programs exhibit locality of execution and a local scope of reference; in a similar way circuits can also show locality of placement and therefore locality of routing. Transformation service overheads can be kept to a relatively acceptable minimum by utilising this localised behaviour with respects to evolutionary runtime placement and routing, preserving the flexibility of the overall system.

The configuration manager acts as the final abstraction layer, providing a device independent interface to the VH manager. It acts in the same way as the software layers above a device driver do, by providing code libraries and final device abstraction. Some of the services provided by the configuration manager include:

- Configuration of a circuit.
- Partial reconfiguration of a circuit.
- Load in data, passed from the application to the VH manager finally down to the FPGA.
- Pass up state information from the device driver to the VH manager. An example of this my be an interrupt message if the FPGA overheats.

The configuration manager preserves a mapping between two representations. The first involves the symbolic representation of circuits converted to the specific FPGA concerned. The second is the data to be loaded into the circuit registers being in the correct format. This mapping allows communication from the VH manager, such as a request to be converted into the correct programming data for the FPGA. In a similar way, state data from the  device driver, such as interrupts, may be passed back to the VH manager. This may be, for example, that the counter circuit had run for a specific requested number of clock ticks.

The configuration manager maintains an image of the FPGA configuration and state, replacing the need for continual querying of the device, which results in much wastage in terms of programming cycles. It provides services for the VH manager, allowing various information exchanges on demand.

At the nearest level to the FPGA is the device driver. This works with the configuration manager to program the device and communicate with a PCI board, which may contain multiple FPGAs. The required functionality for the device driver is:

- Writing and reading the PCI board SRAM.
- Writing and reading the FPGA cell programming data.

- Interrupting the host when certain board generated events occur.
- Setting the FPGA clocking frequency.
- Monitoring the current drawn by the FPGA.
- Clocking FPGA circuits with the individual, continuous, or a preset number of clock cycles.

In the RAGE system there is a mapping on to the host's processor's memory space of the FPGA programming interface and the SRAM. Both the operation of programming the FPGA and writing data to the SRAM are available as programming language assignment operations. The FPGA and the host can use the SRAM programming interface as a fast cache. In this specific set-up there are two SRAM banks. A bank cannot be shared, as such but the device driver allows ownership to be switched to permit data to be shared at any time.

The PCI board, used in this paper, has a total of three interrupts. When the FPGA draws too much current, or exceeds a predefined level, this triggers the first interrupt. The second interrupt is set when a specifically given number of clock cycles has elapsed. The third interrupt is user-defined and can be set by the FPGA for any reason. These interrupts allow the device driver to be interrupted, at which point the manager can either handle the interrupt itself or pass on relevant information to higher levels. This can be propagated all the way up to the application, which can then take appropriate action.

These system entities described are meant to provide a general foundation of functionality, applicable to many general application contexts. They do not cover every eventuality but rather can be built upon and extended by inserting application specific entities as mediators between the proposed basic system mentioned in the paper and the application itself. An example of how such extension could be applied is the partial evaluation already mentioned. Another, more simple, example is the addition of a Specialisation Module effecting constant propagation (this is a standard optimisation performed at compile time) [51]. Known inputs are fed through the circuit and logic optimisation is performed e.g. an AND gate with one input high can be replaced by a wire driven by the second input. In some contexts this kind of optimisation may be best performed at run-time for dynamically reconfigurable devices. If the specialised circuit is to be used many times then there would be recouping on the run-time costs of calculating it. An example can be given where the AND cell has a constant high input and is optimised away, thereby converting what would have been three wires to one.

By propagating values of constant registers (a set of protected input-register values) throughout a circuit, the Specialisation Module working with the symbolic description optimises a circuit. When an asynchronous feedback path, an output register, or flip flop is encountered, propagation stops.

Modules which extend the system, need to provide a simplified interface and yet be able to access detailed information about symbolic, possibly transformed, submitted circuits – as well as having fast access to data structures. The way this is explained is by having a privileged interface, a superset of the application interface, which would allow such extension modules mediated and controlled access to internal run-time system data structures.

The RAGE system has many interesting features. Most similar systems are highly specialised to particular applications, or groups of applications, where RAGE is more generic and is not tied to a specific application. Applications must obviously be designed with the system in mind. The main goal is high-level abstraction of access to the FPGA.

Several sample applications are examined. The Dynamic Instruction Set Computer (DISC) developed at Brigham Young University, utilises FPGA reconfiguration to dynamically supplement its own instruction set. Each instruction or circuit in the system can be swapped on and off the FPGA, which in this case is the CLAy31, as the running application demands. In a similar way to RAGE, DISC maintains a library of circuits, each of which may be positioned at any vertical position on the FPGA. RAGE extends this by being able to position anywhere. DISC reconfiguration is controlled by a reconfiguration controller, which executes on the FPGA – the programming being dependent on a retargetable C compiler. RAGE on the other hand, runs on a host system where debugging is somewhat easier. RAGE provides, in this way, a higher level interface for hardware/software co-design.[17]

Another system mentioned in this paper is the Run-Time Reconfigurable Neural Network (RRANN) which uses dynamic FPGA reconfiguration to implement three stages of the neural network back propagation algorithm. An application swaps on and off the circuits which represent each part of the algorithm. In this system only one circuit module is resident on the hardware at a given time. Static circuitry is also on board the FPGA all the time, which does not change over configurations. This circuitry controls dataflow and the sequencing of execution of the dynamic modules. The RRANN system speeds up execution by simple and minimal usage of dynamic reconfiguration. It maximises the amount of on-device static circuitry and therefore spends less time reconfiguring circuits and system efficiency is sped up. There is also reconfiguration at a much finer granularity. An example of this is the reduction in circuit size; a counter can be shaved from 11 to 8 bits and the rest of the circuit remains the same.

These systems can be both implemented on RAGE. The circuits would have to be developed firstly to work on a Xilinx FPGA, the second step would be to write code in any language that MS Windows™ DLL functions can be called. In this way the application can access the virtual hardware, without resorting to redeveloping the application program.

Yet another possible conversion to use RAGE would be a technology developed at the Department of Electrical and Electronic Engineering in the University of Strathclyde, named Fast Reconfigurable Crossbar Switching. This implements a fast reconfigurable switch on an Atmel AT6005 FPGA, used as part of an ultrasonic imaging system, which is highly time dependent. The usage of the Atmel rather than Xilinx at this time would preclude the conversion of the system but more devices are capable of being added to the proposed RAGE system and therefore its feasibility is not beyond reach. The dynamic reconfiguration would then be handled by the virtual

[17] Others have worked at this aspect of hardware/software co-design [21].

hardware manager using pre-placed and routed alternative circuits and may also utilise the services of the RAGE transformation manager.

The RAGE system is shown in these examples to provide a possible way for applications to have access to FPGA dynamic reconfiguration at a high-level of abstraction. Its usefulness is particularly evident by the usage of the transformation system, providing circuits with a larger area of free resources to be mapped upon. This is far better than most ad hoc methods of producing systems, which are specific to the application, giving a generic approach with simple interface methods. The idea of the system being extensible by providing plug-in modules at various stages adds to the flexibility introduced. There are many times when a specific application may require a particular algorithm to be available at run-time. This may be, for example, optimisation, or for the solving of problems at a symbolic level within the design. It could be the generation of solutions through genetic algorithms, for instance.

The possibility of usage within a hardware/software co-design development environment would allow designs to be developed in tandem with the software component. Partitioning between the areas can be tested and efficiency of solutions immediately assessed.

Virtual hardware as defined here – the swapping of circuits on and off the board – is the main functional mechanism which is available. When backed up by the generic method of access, this becomes a very powerful tool, which is available to a wide range of applications. The down side, of course, is the necessity of such applications to be adapted, albeit minimally, to provide the interface to the RAGE system. As suggested, some existing applications can have requests to hardware channel through a conversion program, operating at the host's system level. This program converts the demands of the application to be in line with what is expected at the RAGE end and vice versa. Some degree of knowledge is required of both systems for this purpose though and may not be an altogether straight forward task.

### 2.1.9 Analysis

The work in this area can be broken down into several basic aspects:

- Basic control and architecture.
- Run-time sequencing.
- Mobility.
- Packaging.
- Partitioning.

The architecture and method of basic control are obviously interlinked. Although FPGAs have still a few ways of interfacing with their surrounding environment, this has become easier through hardware and software mechanisms, which allow data to be downloaded and uploaded quickly and efficiently. For example, in the Virtex FPGAs a state machine exists at a low-level, which allows data to be exchanged using a relatively simple protocol, implemented through minimal wiring. The actual configuration bitstream consists of packets containing commands and data which is targeted at specific resources or areas within the device.

Several methods have been presented from past research work which instantiate objects within a devices matrix and use different approaches in their manipulation.

The term mobility is used here to describe the ability for hardware processes to be positioned anywhere within a device (possibly over several types) and its ability to be transferred over networks. In the research work looked at in this section this has mainly being confined to placement within a specific device type. Circuits have been manipulated within a device to be target both with some constraints, as in the case of DISC and much more flexible, as in the case of the RAGE system. The RAGE system also shows the value in being able to transform a circuit's basic shape to fit within areas which, if not used, would be wasted.

The way in which objects are manipulated relates directly to the way in which they are packaged at run-time. There are several levels at which the actual design could be translated to hardware. For example, a high-level language such as VHDL could be used and synthesised directly to positions on the device matrix. The same applies to netlist formats. A pure bitstream straight from a hardware compiler could be used and in some way transformed to the given locations resources. Recently JBits has allowed direct control over resources at a high-level, so this too could be used. There is a relationship also between the initial design process and the eventual packaging chosen, which will be manipulated during run-time. Certain methods of detailing circuits may lead to more lengthier transformation or synthesis periods, which should be avoided in the case of real time systems, where timing is a priority.

The ability for some manner of mixed functionality between hardware and software components allows a trade off between the best features of each. This can be seen particularly well in the DISC system, with its mix of C programming and hardware objects. Such flexibility allows partitioning possibilities to be explored and compared both quickly and accurately.

Through the above areas there is an obvious impact on the programming language and the initial design stage.

In terms of language, this covers both how sequencing and control occurs but may also apply to adaptation or mutation of designs during the run-time period. The DISC system utilises C successfully but the physical nature of the hardware circuits lends itself toward a language which uses object-oriented expression. This can be seen in JHDL, where circuits are built as objects within constructors. It must be remembered here that we are talking about languages which successfully bridge the gap between hardware and software, being able to describe both domains. This is in contrast to languages such as Handel-C and others, which are HDLs describing only hardware in behavioural terms.

The time before run-time, the design stage, is sometimes impacted upon, depending on the needs of the run-time system. It may be, for example, that it requires the design in a specific form. This may be in terms of the way the circuit is made with specific I/O points, or the format of the circuit as net file or bitstream. There may also be some other constraints such as timing or area, which may have to be considered at this early stage.

There are obvious advantages to being able to utilise already available tools and systems to develop the initial designs. Specialised development tools would need to be learnt, you could probably not utilise existing hardware libraries and finally, would have a limited knowledge base support. There may also be problems with tools that can use common formats as input for debugging or visualisation.

This section has provided reflection on past attempts at producing run-time and hardware/software manipulation systems. The discussion has centred on the ability to supervise FPGAs at run-time, the ways of manipulating hardware components and the language needed to express this in the temporal and spatial dimensions. The main features of such a system have been highlighted and overviewed, the next section will develop these concepts further and outline the philosophy behind this work.

# 3. Design of a Solution

## 3.1 Philosophy of the Approach

The analysis of past attempts at run-time systems yielded several key aspects: basic control and architecture; run-time sequencing; mobility; packaging and partitioning. The philosophy behind building a dynamic system which combines such elements and extends past research into these areas is now explored.

The main vision of a dynamic run-time system described here has several requirements which are based upon the analysis of previous work by other researchers, as detailed in the previous section and through experimentation and development of theories by this author.

It has already been detailed that there are many ways that an architecture of a system can be implemented with FPGAs. A standalone unit can be developed (or purchased) that interfaces to a computer through standard I/O cables, or even through network links. Boards are also available which plug-in through the PCI bus, making the PC itself the host. In such systems it is possible to develop on-board run-time systems that are self-configuring or host configured. As has been pointed out, it is possible to do this in various degrees, with the main FPGA configuring itself; another FPGA or microcontroller; or a host computer doing the task.

The main control mechanism examined here is based on software control, in other words, it must run on a computer system or local microcontroller. On a large enough FPGA and extended memory it may of course be possible to run such a system on one chip eventually.

The basic model for a dynamic run-time FPGA system which contains mobile self contained elements is now determined. In overview the system itself can be broken down to several key areas.

- Entities (the designs)
- Capture system (the means of extracting a design)
- Run-time system

To allow for mobility over networks and within devices, a form needs to be found which will allow it to be easily manipulated, transported and encapsulated. A means must be found to translate the original design into this new form and finally some way must be developed to manipulate its placement, connection and communication.

To control such activities on the run-time system there needs to be a way of executing programs and a language with suitable commands and structures for control, sequencing and repetition.

### 3.1.1 Entities

An entity is envisaged here as an entirely separate object with defined input and output points, much like programs within object-oriented software engineering or modules used in the hardware development. They should exhibit all the properties and good software practices associated with such objects; for example, only allow loose coupling to external programs. These entities are built up to form a design, in other words, they are compositional "black boxes" or building blocks. Part of the entity in this case is actually even closer to true independence than its software counterpart, being a physical circuit, when instantiated, rather than code running in a computer.

Normally in a hardware design a complete configuration is built up at design-time, with the engineer using several tools and languages. Components may come from libraries, or be built from scratch. This may then be tested, synthesised and optimised, finally producing a bitstream for download. This approach does not allow for the mechanisms implied here, where individual entity resolution is required at run-time. Therefore entities need to be extracted separately, for manipulation at a later point.

There is a trend to close the gap between hardware approaches to design and software cycles, especially since behavioural languages such as Handel-C have appeared. The way to achieve this is to see the design in a compositional sense rather than one large object. In other words, a large part of the design process is moved to runtime rather than at a static point beforehand. It can be seen that the system described here brings the hardware/software divide closer.

It is envisaged that entities should be mobile within devices and over networks, possibly being controlled within a client-server topology. To do this the entity must be in a form which can be transmittable from the main server to the device node. It must also be capable of being positioned within such a device. There are three possibilities here; the use of a pre-synthesis format, such as VHDL; a low-level approach, such as the configuration bitstream; or an intermediate form.

Utilising VHDL or other such design language would present too much processing to be done at run-time, the equivalent of re-compiling in fact. It also does not provide a suitable mechanism for certain requirements at run-time, such as communication with software. The configuration bitstream is large and inflexible for transmission between devices. It is also infeasible (and inefficient) to be able to manipulate a design within such a structure that may be very small in relation to the data packet itself. The design must therefore be extracted from such a file and placed in some mediating format. This could be a relatively dumb structure, such as a reduced size bitstream, which can be passed to the device for processing and placement.

A better idea to pursue would be to include this data within a program that encapsulates the circuit and extends the functionality of the entity into the domain of the software itself. It is therefore proposed that such objects are bi-partite – they have a physical and software component. For example, some of this functionality may be the actual ability to build its physical circuit itself into the FPGA. The entity may have the following functions:

- Circuit building and resource capture
- Communication across domains
- Circuit modifications
- Circuit destruction and resource release
- I/O Port handling

Communication across domains is here defined as a means of data exchange between the software and hardware planes. This allows interaction to happen with a instantiated design without the need for physical wires, a “virtual” link is formed where it is possible to have defined variables or registers for data exchange.

Circuits as well as being built, must be destroyed and the resources released in the same way as memory is in a conventional computer system.

Ports in such an entity must be defined at both a circuit level and for the software representation. Functionality can be added at the software level to provide not only a communication between circuits but also a means for exchange of structural data or information on the object itself. Another entity may, for example, require information on port structure, such as number, names, direction etc.

The entity may also hold state information such as RUNNING, FROZEN, READY_FOR_CAPTURE etc. Other information could also be stored and passed back, such as circuit positioning, size and resource use. This information would be useful for any run-time system that endeavours to communicate with or manipulate the object.

This translates to object-oriented practice, although certain aspects of the entity, such as the physical circuit it contains, are better represented from a CSP process viewpoint [15]. The hardware part of the entity is more self-contained in the sense of control, it does not rely on the time-sharing of a processor to operate, or a thread which gives it a focus for an amount of time [18]. CSP is based on channel communication which translates easily to the wire-based communication available within electronic processes.

The viewpoint here has been largely that the software wrapping the hardware acts as both a mediator and vehicle of support for the physical circuit, where it is possible to actually have entities which are equally functional on both levels. There is no reason, for example, why an object could not contain code which has functionality beyond the support of its physical part. This may be to process some incoming data from another object before being passed to the circuit itself, or may support its presence at a software level. The entity becomes hybrid, having structure and functionality on both levels.

The physical circuit need not have any particular constraints placed upon it; it would be possible, in theory, to use legacy designs converted to the new encapsulated format. However, it would be wise to conserve layout space on the device matrix. It would be possible to manipulate such a design at capture to the new format to reduce size of layout, although care would have to be taken to preserve factors that have been optimised at synthesis time. It is also possible to configure synthesis for compact layout at implementation time, as has been seen before in previous attempts using geometric transformation.

A circuit must also have well defined I/O ports. Such ports must be able to be detected by any capture system and matched with named labels to preserve programming semantics. It would be useful if  a port's names persisted for the user of such entities, manual naming later could prove to be laborious and unnecessary if such details already existed. The data for I/O ports is usually contained in a separate file in any case, known as the User Constraints File (UCF). This data could be matched to any ports found.

It may be an interesting idea to propose entities which are purely software, that is, they are carrying no hardware information. These units can be simply entities which supply software functionality to the run-time system, which may not be available and could be thought of as any other object-oriented class, with the advantage of being able to load and unload at run-time. It would be interesting to compare algorithms implemented as hardware, with a similar unit which is purely software. This allows flexibility at design-time, especially if hardware design-time is greater, as software can be implemented to functionally “get the system up and running”, or to test possibilities.

### 3.1.2 Capture

A capture system is required to produce an individual entity. The entity could be produced, in theory, from a pre-synthesised design in a language such as VHDL and another program generated from this. However, if the entity is developed from the synthesised configuration bitstream, the initial choice of the engineer, in terms of tools and language is not important, thereby creating flexibility. Legacy designs can be, with minimal adjustment, converted to any new system.

A bitstream contains the two-dimensional format of the device in a linear file. The idea here is that the capture system generates a program from this which can rebuild a circuit on the device at any point. As well as this, the entity must also be built with support software and any extra functionality (such as communication) on that level.

The actual capture of the circuit can be seen in various ways, being careful to remember that the aim is to produce a faithful, hi-fidelity replication at the point of delivery, in terms of its functionality. It must maintain the correct gate propagation and time specific aspects held within the circuit’s form, which was probably optimised to various constraints and criteria.  To maintain these aspects it is important to capture a circuit’s tree of connections and resources in order, from point to point. This can be visualised in the sense of a network of resources. Although the resource network must be maintained in terms of timing, the actual physical spatial replication is flexible. In other words, the mapping can changed as long as the gate and timing criteria are met.

There is therefore a choice between direct and relative mapping. Direct mapping relates to a faithful copy of the two dimensional form the circuit takes, it actually corresponds visually to the original. This kind of mapping uses the original design as a template onto the new area, with the same resources instantiated in the same physical pattern. A relative copy takes into account distance, in terms of resource (such as gates) and time but may not look anything like the original if mapped onto a two-dimensional matrix. Resources in effect, are redistributed in a different pattern taking up an entirely new area, which may be more compact or have greater physical distance, depending on the needs of the situation.

It is also proposed here that it would, for example, be possible to map one circuit over the same area as another, if the resources were available, albeit dispersed over an area – possibly intermeshed with another circuit. As well as the required resources, there must be enough available interconnect, that is, wires and switching to reach between the points. Direct mapping would almost certainly be impossible to use in such circumstances – the wiring routes are known and set. Relative mapping relies on automatic routing functions rather than templates. A search for free routing resources takes place at the expense of time and a possibly rather convoluted path. However, if there is a route, it will usually be found and in a reasonable amount of time, even in a real-time application.

If each circuit to be converted into an entity is implemented in a bitstream alone, then it could be easily extracted singularly. Any I/O ports could be detected by tracing them to I/O pads around the periphery of the design matrix and linking them to designated names with the UCF file already mentioned. The design name labels can then be associated with I/O pads. Nets which reach toward such points, from the main body of the circuit, are largely superfluous, as they mainly indicate I/O points on the actual physical object itself. A problem here is the possibility that there may be many points on the circuit which reach to an I/O pad. The answer here is to create special I/O points, which collect such nets together. This reduces the amount of run-time wiring, resources used and overall time for implementation.

### 3.1.3 Run-time

The primary method of deployment for such entities here is through a run-time system. This is a typical client-server model with the run-time system acting as the server to device client nodes. These device nodes can in theory be local or distributed remotely over some distance. The system could use TCP/IP protocol for network communication, as the software infrastructure is already supplied via XHWIF in JBits. The server could therefore operate on one machine and could control several devices.

There is a division of intelligence or functionality between the entities and the run-time system. It is viewed here as important that the objects themselves are as self-contained as possible. Not only is this in keeping with current trends in software development but having self contained units allows them to be used or communicated with, by other systems in the future. They may also be utilised within user-written programs outside of the run-time system altogether. They should therefore be able to build themselves, know their own port details for communication, various circuit statistics, have means for communication and finally be able to destroy their instantiated circuits, releasing resources. The run-time system should therefore merely initiate requests to the entity.

The actual placement on the two dimensional matrix can entirely be under the control of simple algorithms which determine resource availability and other metrics. An area can be determined whether it is relatively empty, by maintaining a simple representation of the resource usage. This may be a boolean multi-dimensional array. Overlaps can be avoided and processed quickly at run-time. This form of placement intelligence, although simple, stops the necessity for the user to work out where objects should be placed. It is then possible for the user to just request entities, rather than having to consider where they are located, in a similar way to many high-level languages that manage memory resource usage at run-time, rather than explicit allocation by the programmer. The above scheme works on the idea that designs have defined areas rather than being intermeshed with other processes. This makes resource management easier and guarantees a degree of freedom and efficiency at run-time. It would be a little more complex but not impossible, to implement a system which utilises any available resources over the surface of the device matrix. It may be harder to guarantee routing to such dispersed resources and also there may be some degree of wastage.

This model of entities that are composed of both a software and hardware component are defined here as Mobile Hardware Processes (MHP).

Processes can be handled in the run-time script in a similar way to objects in object-oriented languages. An archived process forms the basis of a class or type.

Within the language, new processes could be brought into use through the connection of a variable name with a base process.

The language itself must have capabilities for interacting with the instantiated process. A process's registers form a good method of communication with the run-time system. Similar to object types mentioned above, such registers can also be connected with a label. If such locations are already defined as ports, then a user script can utilise the names used from design-time. These can then be accessed as variables in any other programming languages. The variables can be checked and initiate activity, such as the loading of new objects, or modification of existing circuits.

It is also possible that virtual properties could be formed between the circuit and its support software, for example, it may also be possible to develop channels or wires that do not exist in any physical sense but emulate, in some way, CSP channels. Taking this idea further, there is the possibility of creating virtual resources that the physical circuit can tap at a software level and vice versa. A circuit, for example, may require access to memory which is available through the host system. This memory could be arbitrated through functionality supplied by its support software, emulating, possibly, a virtual bus. It would be equally possible to turn this round and create an entity which is a memory block with appropriate interface circuitry and connect software to it. Some solutions will always be easier to implement in hardware and some in software. This partitioning of tasks can be suitably developed, as mentioned above.

As well as the use of scripts to control and manipulate designs, there is the possibility of utilising graphic front-ends. This would allow designs to be developed easily and quickly. Using a visualisation tool would allow entities to be added to specific locations easily, or in a more simple sense, just added to the design. All the standard methods of the graphic interface could be brought to bear, including the ability to drag and drop entities and connect them using point and click. A graphic system only sees a static image at any point in time, whereas a script implies a temporal element in itself, as each command or statement is executed. To not lose the flexibility of the run-time system this would in some way have to be captured. One possible way would be the introduction of something similar to an animation workshop, that is, a design is built up in frames, which are executed over time at a pre-determined rate. Initial frames may include the building up of the basic design and interconnection. Frames may also include elements for the input or interaction with entities, areas could also include visualisation windows which allow the contents of registers or variables to be spied upon. The disconnection and destruction of entities could easily be visualised in this way.

### 3.1.4 Extrinsic Mobility

Extrinsic mobility is defined here as the ability of designs to be mobile, not just in the sense of positioning within an FPGA device but over networks too.

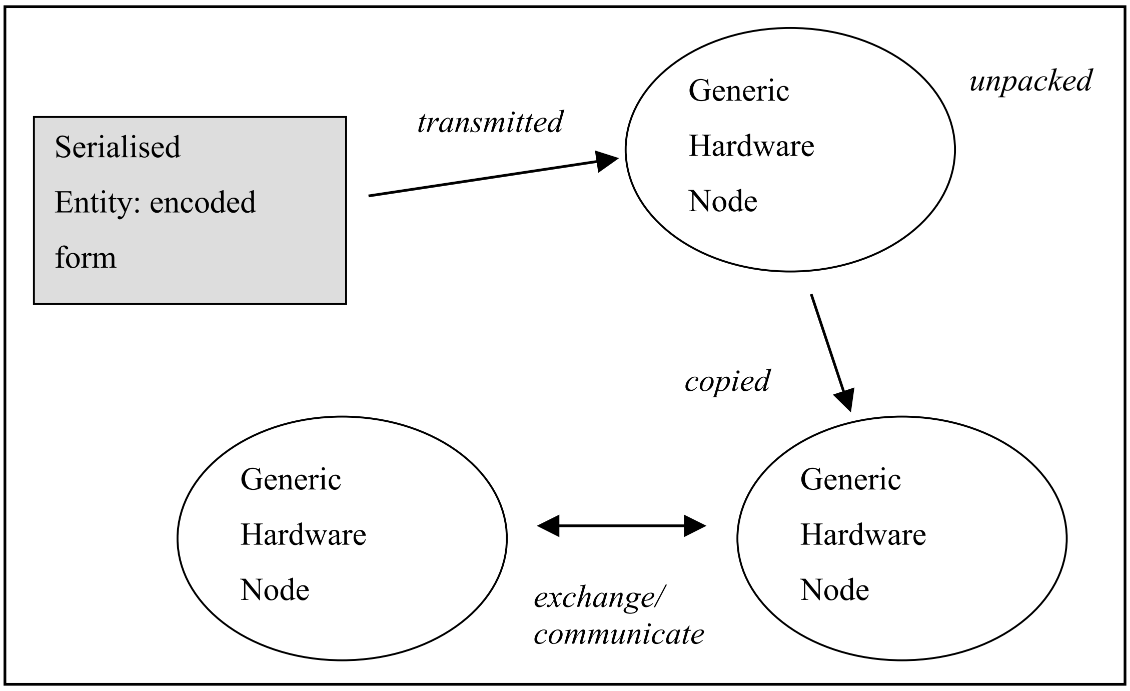


**Figure 10. Typical life-cycle of an entity**

The envisaged approach to distributing entities is via a client–server model. This focuses on a central node with the capability to fully implement a server, which is part of a run-time system for the FPGAs, which are distributed locally or remotely. This allows such nodes to be essentially dumb units only requiring enough hardware or software support for TCP/IP connection. Integrated circuits already exist for this purpose alone and the actual software support for such FPGA reception of configuration bitstreams is available from Xilinx. It would be the job of the central controller/server to run user scripts which act on outer nodes.

It is also possible to have a network which contains several such servers and many more dumb FPGA nodes. A server unit basically holds a model of the connected node in its memory, which is interacted with firstly, before being transmitted to an FPGA device. The software component of the entity exists in the host server of the run-time system, or possibly memory attached to the remote node. These software components would act as normal software objects, with the added functionality of being able to communicate with its instantiated circuit. Communication is mediated through the memory model, which is periodically updated on specific events taking place or by request. It is possible that the user written script could initiate connection and disconnection at will to such online devices, rather than only having one script per node. The server would have to release run-time models as new nodes are connected and build new ones on demand.

A server node contains the design pool, or collection of processes which the script can access.

Other methods could be devised than the client–server model. Such entities, for example, could become autonomous by increasing their complexity. Another alternative would be self-configuration of devices where the run-time system itself is on board the device. This bears close relation to the idea of the mobile agent [11, 34]. A mobile agent is a program which basically directs its own movement over a network, eventually operating in an environment where it is supported by existing processes. Usually such agents have been written in scripted languages that have been sent from place to place on a network. They exist within a context or framework on the target node (the machine at that point) on the network. On reaching the destination, the node must invoke a particular method on the agent which may, for example, be called start. This invocation gives the agent a thread of control. The thread then interprets the code in the script, in order to perform the various commands and actions described there, which may be file processing or document searching. The agent is forced to support a certain API, which includes tasks such as starting, stopping and moving.

The operation of moving means that the agent must be able to package itself up and cause transmission to a new node, whereupon reaching it starts all over again. It may be that the basic set of tasks supported in the agent's API far exceeds those mentioned here to coordinate with processes already active at the destination point. For example, in Java these may be the normal operations found in a thread such as suspend, resume and sleep. Other methods may also be provided to clone and destroy agents too.

An example task for such mobile agents may be searching for information on the web. The information may be, for the sake of example, on one of 1000 web sites. This would ordinarily be a massive search, however, it is a suitable problem for mobile agents. Normally a large portion of the documents from each of the websites would need to be downloaded and searched. Instead, a large number of agents can be sent out – one to each of the 1000 web sites with the functionality to search each site. It is easier to collect the information from each of the agents than the alternative. Yet another simple example may be the prospect of moving a large database over a network for processing, or the alternative of moving a mobile agent to the location to do the task.

The mobile agent model can be difficult to implement to solve some problems where the situation is not as clear cut and there are many complex events happening.

Yet another model is the mobile objects perspective. This relies within a Java framework on the ability of individual objects to be serialisable and transmittable. Mobile objects are part of a larger application and are moved as part of a normal remote invocation-style communication. Very little change is required to a Java object to make it behave as a mobile object and unlike the mobile agent, no particular API is essential to its functioning in another environment. Any serialisable Java object can therefore be a mobile object, as long as it implements the java.io.Serializable interface.

Mobile objects can be seen as components which shift between location on demand from an application, where a mobile agent is a complete solution to a problem. The agent therefore tends to be heavyweight, with a large amount of coding.

A mobile object in Java could be as small as a couple of member variables, up to a complete application.

These ideas equally apply here. The entity which carries the physical circuit's design, if implemented in Java, could also be made serialisable and therefore fits the criteria of a mobile object. Given the above scenario of applications being able to remotely swap components, this could be applied here where the run-time system acts as the host to such mobile elements. This, however, would mean increasing the complexity of the hardware host, which supports the device at a node, as it no longer acts purely as a client system but would require a full implementation of the run-time system.

It may be possible to produce a form of the mobile process which is self sufficient, to some extent, requiring only a degree of support, although some minimal run-time system would still have to be present at the FPGA node. A process transferred to such a "blank" node has to install itself and would therefore require access to a JVM to interpret the byte code, which points to a much more complex node.

It is feasible that some pure hardware design could be developed, which can access and manipulate the configuration of such a device, not requiring the software transport mechanism. This would be made more feasible if a microcontroller is instantiated, or at least some kind of FSM. A feedback mechanism would have to exist which allows readbacks to be processed by its own self-configuration capabilities. It is possible, for example, that a cache of design objects could be available locally to the device which the on-board processor would then be able to access.

A much simpler scenario is that designs are simply distributed from the central server, having no software component whatsoever but this lacks much of the power and flexibility of the system described here.

The three basic scenarios above detail the main network configurations. Which one to use depends obviously on the actual purpose of the system. Only a relatively simple system needs to be in place for the update of remote equipment, say cell phone transmitter stations, the advantage being that no engineer need be present. Yet another application may be the update of consumer goods. Both of these require little more than the download of a configuration bitstream. More sophisticated arrangements, involving the application of hybrid software and hardware processes, could be in use where there is a more complex behaviour needed. This may be to exhibit fault tolerance for example, or configuration on specific events taking place. It may be that a degree of interaction is needed with the local environment, before a configuration is decided upon. All these issues point to the requirement of a run-time system.

This chapter has examined the main aspects highlighted by previous work in this field, that is: basic control and architecture; run-time sequencing; mobility; packaging and partitioning.

We have seen that it is possible to build or purchase boards which have a reasonably transparent infrastructure, to allow generic communications with the FPGA components, utilising proprietary drivers by the manufacturers. These drivers have been extended to allow a basic communications system, along with more complex internet-based reconfiguration.

The idea was proposed of highly mobile, compositional, interactive entities known as mobile hardware processes. These components are part hardware, part software, in form; the software effectively encapsulating the hardware circuit, providing communication and interaction with its circuit, as well as extending its functionality into the software domain. The program effectively acts as both a builder and transport mechanism for the hardware part. It was shown that the entity should be mobile within devices for placement and in the sense of movement over networks and systems. If correctly done, such entities could be used with any software, rather than any run-time system developed here.

The level of complexity of such entities was explored. Processes, it was stated, could be provided with either a basic level of intelligence, or be relatively dumb. A basic level of functionality would provide such capabilities as building, communicating and transporting its hardware circuit, whereas there is also the possibility of extending this further. It could provide, for example, functionality, which is difficult or inefficient on hardware.

A means of developing such entities was proposed, which entails capturing a hardware circuit from a bitstream, which was developed from any standard electronic design tools. This allows legacy designs to be easily utilised, allowing access to many libraries of components and sub-systems. This also allows designs to develop using previously known methods, with no new techniques or tools to be learnt.

A run-time system was proposed that can utilise the above mobile hardware processes, within the context of a client – server network architecture. This would allow internet linked, or local devices to be updated whenever required. Processes operating on such devices could be communicated with, altered and transferred, as needed.

The main method of control over the design engine was suggested as a user-written program script. The idea was developed that such a language should be at a high-level and could be made in such way as to hide much of the hardware detail. This would include wiring, low-level resources and possibly, placement. Detail regarding actual positioning and orientation of processes would be placed under the control of the run-time system. Another proposed idea was that of making design information persist from the hardware development stage, through to the manipulation with any high-level language. An example of this is the ability of named wires and ports to be referred to by the same identifying label within the user written control program at some later stage.

This section has extended past research into several new areas, the most prominent being the introduction of greater mobility and the ideas behind producing a hybrid software/hardware entity. These concepts are the basis of the dynamic run-time system proposed here. In the next section, the language requirements of such a system are investigated.

## 3.2 Language

Here language features are explored that are required for the run-time system. Probing issues from a user or language point of view helps create an understanding of what is required in systems, such as the design engine and refines the development of ideas from previous chapters.

The language explored here is the main method of control within the proposed dynamic run-time system for FPGAs. It has been seen from the discussion so far that the requirements for such a language necessarily exhibit certain characteristics which bear relation to object-oriented programming, as well as aspects more closely resembling CSP/occam [15, 16] constructions, in particular, channel style communications.

It has been reasoned that a high-level approach should be taken, that is, where possible, low-level hardware detail should be avoided and unnecessary complexities for the user, such as placement and routing, to be dealt with by the system.

A user written script should enable such entities to be controlled, interacted with and deployed. This script could be parsed and statements executed in the run-time system contained on the host.

The script need not include hardware detail at all, as the level of abstraction can be kept high, rather than dealing with routing and resource details. This further brings the use of such technology into the realm of the software engineer, or to a hardware engineer with little training. A level of abstraction can be reached which visualises an entity that can be seen as a “black box”, with defined I/O points. This simple definition is similar to visual tools used in both hardware and software engineering. Once the design is built and supplied, little attention needs to be given to clock cycles, delicate temporal issues or gate propagation, which can be dealt with by the run-time system.

At a basic level, language would require the following capabilities:

- Load process.
- Place process.
- Destroy process.
- Connect processes.

These basic functions allow manipulation of the process at a local device level. Load, for example, is seen here as a way of loading the basic class into memory for replication as various named entities, which become instantiations of the class type. Place is seen as more specific, as a way of physically locating a circuit on the device matrix. Destroy would release resources taken by placement on the device, so that other circuits can be instantiated. Connection commands are seen as ones which allow the independent physical circuits to be routed for communication purposes. This may involve multiple or single wiring at a low-level, as well as interconnect through multiplexors. At a high-level this would not involve the naming of hardware resources but rather the linking together of various size ports, with no mention of the resources at a physical level which lie between. Preferably there would be no difference in syntax format to the user between connecting hardware, containing process objects and purely software objects.

To enable a process to be moved to another system, or simply reprocessed once instantiated, we also need features to include:

- Re-capture process.

The idea of being able to recapture a design after placement is an interesting one. It must mimic the capture process which occurs prior to run-time and yet be able to isolate one entitie's circuit from another. Using methods that merely trace wiring would lead to any interconnected processes being gathered into the process and captured too. There is therefore the need for an idea of identification of resources and routing between processes. This need not be complex but could be done by the run-time system when the circuit is placed by keeping track of used areas. An instantiated entity with a placed circuit should no doubt keep its origin of placement and size, from which can be worked out its basic CLB layout. If capturing follows and analyses routing, then it would be known when a resource is encountered that does not belong to that particular process.

A recapture may not just build a program in a similar way to the initial capture stage but could also re-compile the generated program and add it to the pool for immediate re-use. Such use of dynamic re-compilation would allow entities to be transported from location to location, as well as frozen for later use. It would also allow for replication or cloning of processes for use in the same device. A cloned entity would be externally mobile to the device too, incorporating any new data it has gained since its initial instantiation. A replication of a process need not imply that the original is destroyed, in fact, it could be left functioning during such a re-capture. The newly formed entity must have a new name assigned to it for the design pool. It would be possible to shut down the original, if necessary.

These various capabilities can only take place if there is significant control over the software environment in which the run-time system is executing. After the re-capture process itself and the generation of the program encapsulating the circuit together with software functionality that it inherits, a Java system would have to be able to access the compiler dynamically. It would then have to start the compiler as a separate process and hopefully manipulate the files that are output into the various directories, such as the pool.

There also needs to be a way of interacting with the active process. There are only two kinds of information that could be extracted from a physical circuit, that is, structural and data. Structural includes connection and routing, while data refers to the state of registers or RAM based resources. A means of viewing routing data would be useful for the more knowledgeable user in diagnostic work, for example, to trace connections. This is seen as beyond the scope of the normal user who need not be concerned with such a low-level of abstraction. We need therefore to add to our list:

- Diagnostic.

It would be useful to access data in the script as if variables in a normal program. This gives the user the ability to write programs which act on given states arising on the board. To do this, a location on the physical circuit could be mapped to a declared name, therefore the language should be extended to include:

- Variable declaration.
- Variables and Labels.

Labels are required to allow for particular instantiations of a process type to take place, as opposed to the class name itself.

To check variables and react on their contents, a range of conditional statements are also required:

- Conditional statements, sequencing.

This would include basic manipulation operators such as “==”, “<”, “>” etc

Repetition will also be required:

- Repetition.

It would be useful to repeat blocks of code for checking states of registers. This could be for a set number of times, or to exit on a particular condition arising.

Basic control over FPGA functionality is also required; such as turning the on-board clocks on and off and their frequency. Other functions may include; the reset of the board and set and reset of BRAMs:

- Basic device functions.

Other useful language features would include the ability to save a configuration to file, console messaging and variable output:

- User I/O.

Finally, a means of initialising a session is also required. This would include device node location, type and initial configuration state, so we need to include:

- Initialisation.

The syntactic style of the language is important; popular programming language features would enable the user to be instantly at home, at least with basic structures. Hardware objects can be defined in a similar way to software objects, because of the way they can be encapsulated within a software class. The software acting as transport and mediating interface also provides the connecting class for its definition for usage in a user written program. This lends itself to defining classes in a similar way to Java and JavaScript, that is:

```
TypeClassName definedLabel = new TypeClassName();
```

or similar.

The software interface enables hardware objects to be treated just as an extension of their software self – it is the methods in that part that control and communicate with it after all. This mimics quite well what is actually occurring underneath at the run-time system level; the mobile hardware process will be loaded in dynamically as a class, so there is a direct correspondence between the user program and the actual underlying functionality.

The language would benefit from maintaining a high-level approach and being as simplistic as possible. The language's main aims, at a basic level, are to allow the loading, placement and interconnection of processes. This kind of functionality can be very simply represented in a language sense, as definitions of objects/ports and placement and interconnection commands. At this level it need not contain any control or repetition mechanisms, as it is basically a place and route instruction set. The amount of detail required for such a simple run-time program varies with the complexity of the design engine itself. For example, a top level approach would allow the user to state that certain objects are required and what the connections are. The system itself takes care of the management of resource allocation. A slightly lower approach would allow some control of placement, for example, where objects are actually positioned. It is far better to allow the system to take care of low-level activity, such as routing and resources. This could be compared to conventional computing in the way memory resources are taken care of with higher-level languages, such as Java.

Control structures can be kept the same as most popular languages, such as C++, Java etc. and it is worth keeping them syntactically the same, for the very reason of familiarity. This enables a quick learning curve for the new user.

For multiple placement there is a necessity to bring in control structures that allow repetition of blocks of code. Further control than this is required in the form of conditional statements for load-on-event.

It may not be necessary to include variables within the actual program; the definition of points on the device itself, such as registers, would allow storage of values. Even structures such as the repetition, mentioned above, can be dealt with using constants or defined locations.

Many interactive features can be added to the language. The simple monitoring of defined locations, without the need for hardware probes, is itself useful for testing and control. Such values can also determine the necessity of a load (or reload) of certain objects and connections could also be made or broken.

During run-time there may be some features which are useful that manipulate the actual design itself. An example of this may be the ability to isolate a process's inputs or shut down outputs. To be able to decouple a process would be useful.

There are some features which would be useful for such activities as testing, for example, to be able to supply values at inputs without recourse to physical intervention. By being able to set states at such points allows processes to be tested while in circuit.

Diagnosis tools could be supplied which would necessarily be low-level, although existing alongside high-level language components available for the more informed user. Routing connections, for example, could be displayed to check resource usage, or simply that a correct target connection was made.

It would be very useful to able to access data about the original design, such as the resources which it was set up with at the time of its design. It may be that a design, for instance, was set up with very specific resources in mind; such as a given set of I/O pads, this data is likely to be contained in the entity but overridden at instantiation. Instead of writing out default information again, if it is required, a language feature could enable the instantiation of a hardware object using default data. The mediating software structure which builds the object could simply have the functionality for such an event to take place. It may be that the designer who built the hardware object had a specific set up in mind and already had the details for the end destination and device. Using this ability, all he need do is instantiate the design without the need to write out such data again, which can be complex and lengthy. Another use may be for the user who knows nothing about the device that he is connected to, or is a new user to the system. In such cases the level of detail required to program the device with the process decreases. Pad information, that is, connections to the outside world, are therefore completely contained within the process itself.

There needs to be some level of detail captured and accessible to the user/programmer about the device he is dealing with. This device-specific data needs to be available within the language itself. An example of this is the pad input and outputs around the periphery of the chip. They have names which are used for various purposes; these names vary from device to device. A typical name may be AW34, which is a unique name for identifying a specific pad. It would be useful to be able to use such detail, as well as a more generic method (such as simple position data), if such detail was unavailable or not necessary.

In this section the key features of the language were assessed and defined. This exploration, from a user point of view, has helped specify further what features are required within the design engine and other sub-systems.

It was proposed that the language must contain features to manipulate the processes, as well as control and initialisation for FPGAs, over networks. The language must have ways of expressing the normal control, sequencing and repetition that is currently available in most languages. More specifically, there should also be the option of some diagnostic functions available for the more advanced user. There should be methods of describing interconnections between processes at a high-level, as well as the ability to utilise information, such as port labels, which were defined during the hardware development stage. Inclusion of the ability to have specific information for a particular type of device would allow pad names and I/O to be used in programs. This would be in addition to the ability for a generic positioning system to be in place, which could identify such points in terms of coordinates, for example.

# 4. Implementation

## 4.1 Implementation Preface

### 4.1.1 Scope

This section details the development and implementation of Reconnetics, a run-time system for manipulating MHPs. The main objective behind this implementation was to provide proof of concept toward the central ideas contained in this work. The main areas of development were the Reconnetics run-time system and the MHPs. This includes several other key aspects, which were integral to these main units, such as the run-time language and necessary software sub-systems, including the server communications system.

Firstly, it is necessary to look at the environment in which the system was developed, which includes hardware and software tools used.

### 4.1.1 Hardware

The main hardware used was an RC1000-PP board. This included a Xilinx XCV1000 FPGA which has 1,124,022 system gates with a 64 × 96 CLB array. The maximum available I/O to the FPGA was 512 pins and a block RAM which totals 131,072 bits. There are also 393,212 bits of RAM available in resources around the chip. The board is connected to the PCI port of a 1Ghz PC, running MS Windows 2000 Professional™. Four banks of fast SRAM are available on the board, organised as 512k × 32 bits, which can be accessed from the FPGA or host. The board has two clock pins, which are connected through GCLK2 and GCLK3 clock buffers for external input and two on board programmable clocks are also available. A clock signal input, for single step purposes, is also available for the host.

I/O is available in the form of headers on the board, which provide access to FPGA pins. A bank of eight LEDs are available, which allow immediate visual confirmation for debugging of circuits. These are connected to various I/O lines on the FPGA.

Generally, access to the hardware was provided by Xilinx software; for normal Reconnetics use this was via the stand-alone Xilinx server software, XHWIF, which allows the board to be accessed through TCP/IP, either locally, or over networks. Hardware was also simulated on several PCs, of various speeds and OS types, to allow a group of nodes to be formed for the purpose of testing a distributed FPGA system, controlled from Reconnetics. Each simulated device, which could be of various types, could be connected to and responds in the same way as a normal FPGA device, plus the server software.

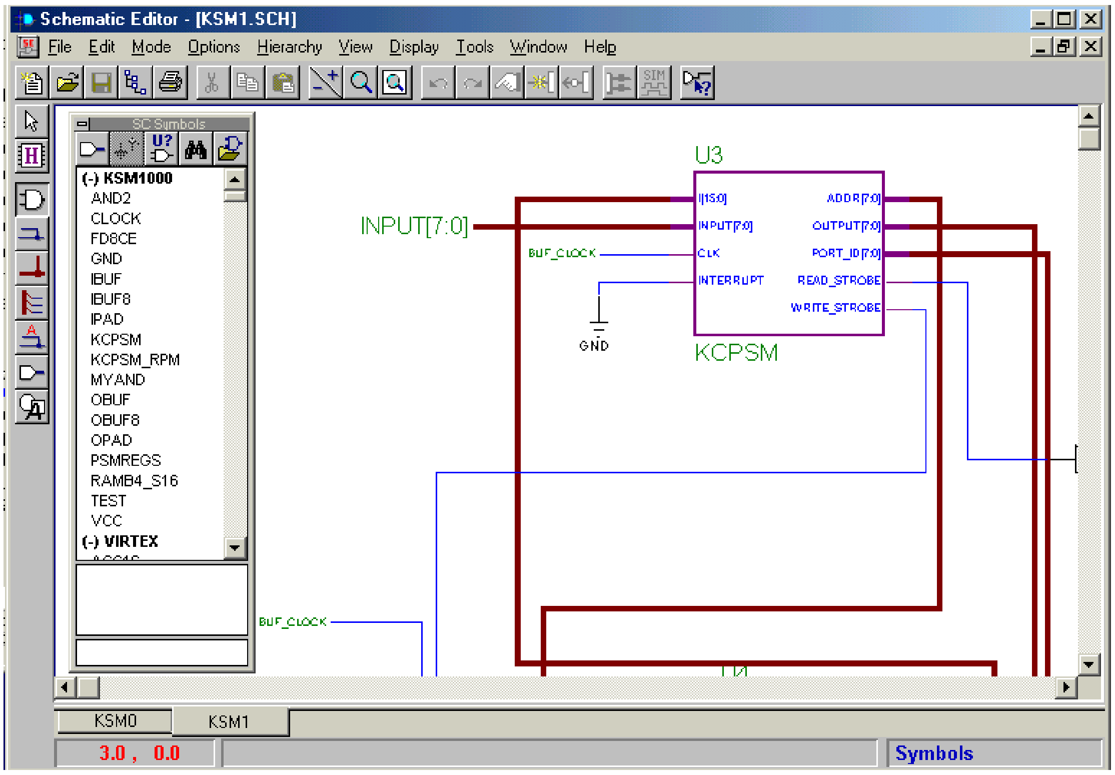
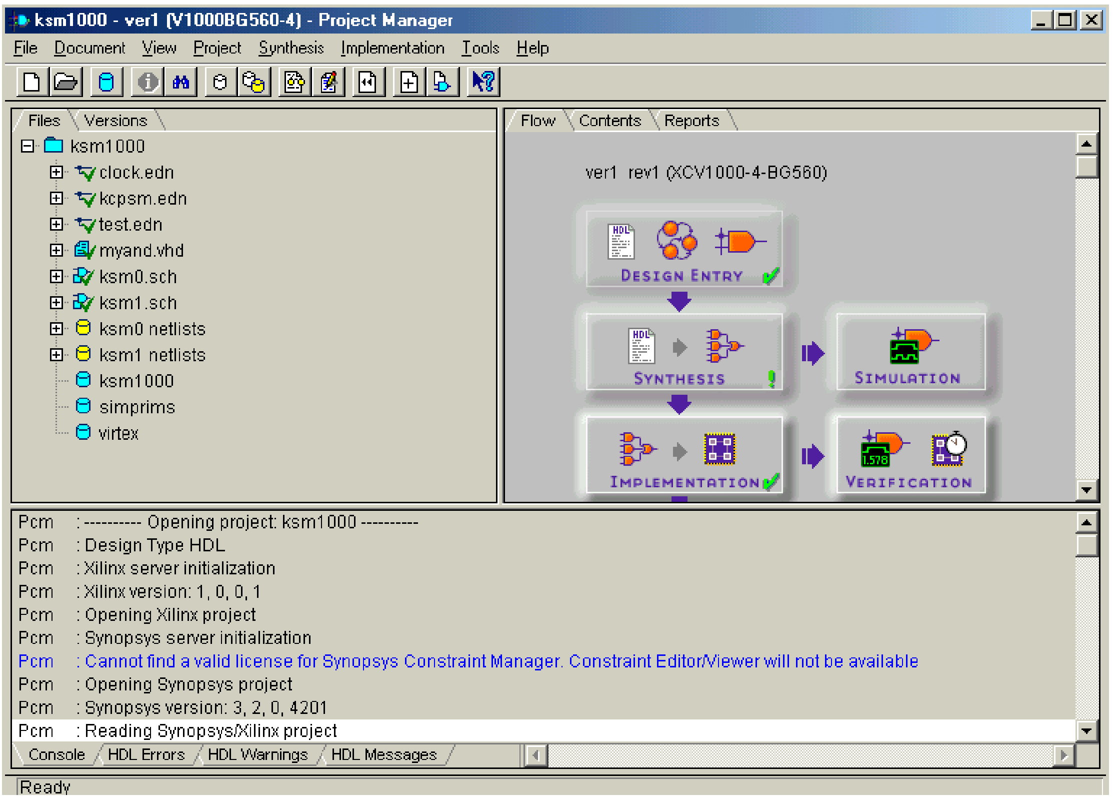


Figure 11. The Schematic editor

## 4.1.2 Design Tools

The Xilinx Foundation 3.1i tools were the main synthesis and implementation engines used for the generation of bitstreams in this project. Designs were given over to the synthesis process in many different languages. These included VHDL, Handel-C, EDIF and ABEL. Other methods were also used for designs, such as graphical CAD and FSM, which were part of the actual integrated design environment, known as the project manager.



```
Pcm   : ---------- Opening project: ksm1000 ----------
Pcm   : Design Type HDL
Pcm   : Xilinx server initialization
Pcm   : Xilinx version: 1, 0, 0, 1
Pcm   : Opening Xilinx project
Pcm   : Synopsys server initialization
Pcm   : Cannot find a valid license for Synopsys Constraint Manager. Constraint Editor/Viewer will not be available
Pcm   : Opening Synopsys project
Pcm   : Synopsys version: 3, 2, 0, 4201
Pcm   : Reading Synopsys/Xilinx project
```

**Figure 12. Foundation project manager**

The next stage after design entry and manipulation is logic synthesis, that is, the process of compiling the HDL code into an XNF or EDIF netlist of gates. The netlist can then be put through implementation stage. Initial details are normally required at this stage, which encompass the synthesis and implementation processes. These details select the hierarchy of the design, a top level file being highlighted. They also give the target device details, such as; family, device and speed. Details also can be given for applying constraints.

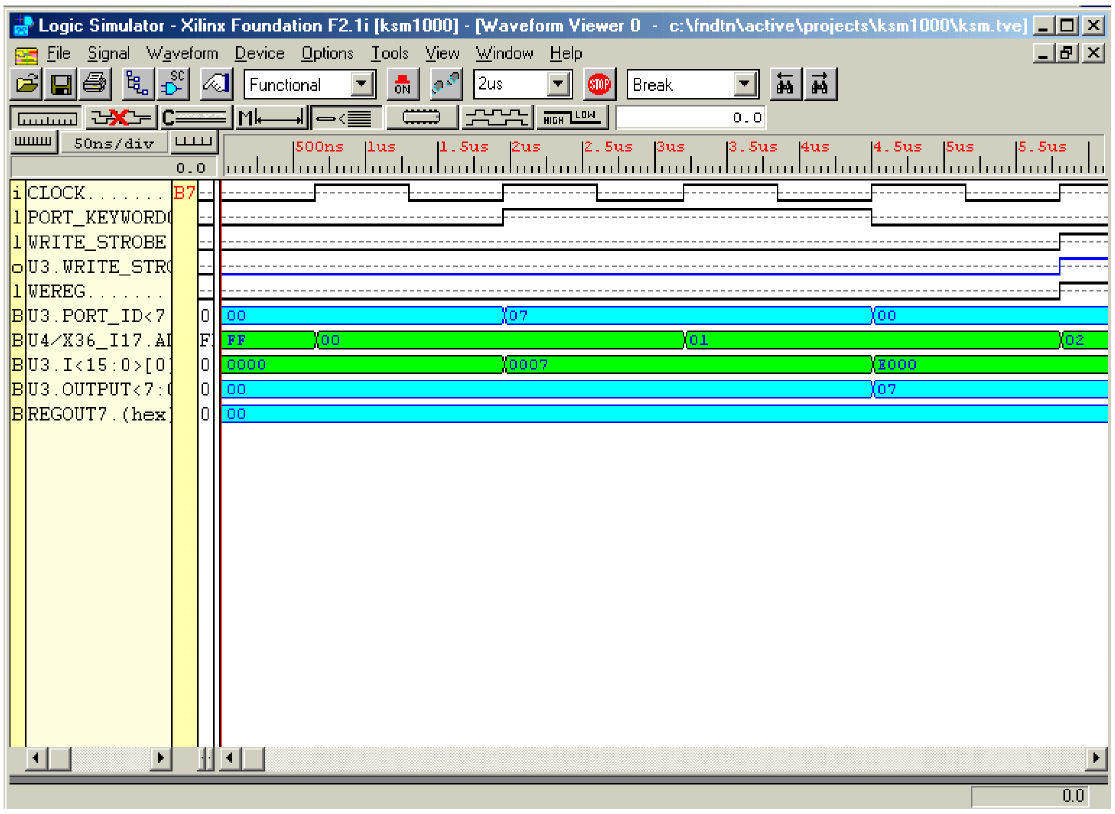

Logic Simulator - Xilinx Foundation F2.1i [ksm1000] - [Waveform Viewer 0 - c:\fndtn\active\projects\ksm1000\ksm.tve]
File Signal Waveform Device Options Tools View Window Help
Functional
2us
Break
50ns/div
CLOCK
PORT_KEYWORD
WRITE_STROBE
U3.WRITE_STR
WEREG
U3.PORT_ID<7
U4/X36_I17.AI
U3.I<15:0>[0]
U3.OUTPUT<7:
REGOUT7.(hex)

Figure 13. The logic simulator

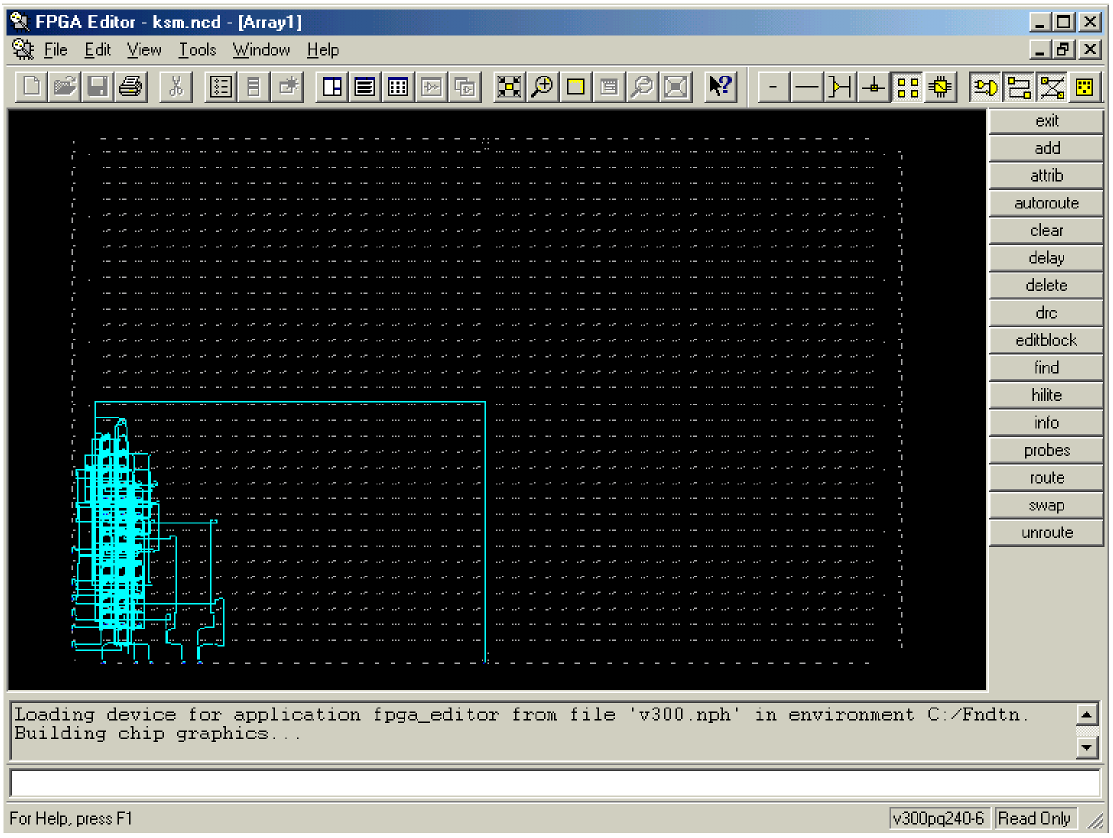
After the synthesis stage there is the possibility of functional verification, allowing basic timing to be tested and viewed using waveform visualisation. Firstly, signals such as inputs and outputs are selected. Inputs are then selected for stimulus, such as being attached to clocks of various cycles, or manual input of some basic signal varying or constant. Outputs can be selected for visualisation over time, allowing gate transition timing to be checked. Access to BRAMs allows data files from disk to be input. They may, for example, hold simple programs for a design, which includes a processor. A simulation test run produces various outputs, which can be checked for correct functionality. Expansion and contraction of the selected viewpoint in time allows the correct level of detail to be gained, producing accurate models of a specified time frame. Glitches can be easily detected in the state transitions, viewed as waveforms. Both the test set and results can then be saved for reuse on later versions. Hardcopies can also be taken of the results, again viewed over various levels of detail.

The synthesis and implementation stages of development can have constraints applied, which specify timing necessities, gate propagation, area and size limitations.

Input and output can be locked to specific pins or pins named, which must not be used in the final design for some reason.

The implementation phase utilises earlier placed options, prior to synthesis. Other options are available, affecting place and routing, such as the degree of effort applied to the process. This is the amount of time to spend working out the routing, obviously the longer time spent the better and closer the end result will be to the constraints given. A shorter period of place and routing gives a result that may physically work but not as efficient, as the placement may be dispersed over a larger area, utilising a greater amount of resources. The whole implementation stage is five fold, involving translation of logic, mapping of logic to resources, placing and routing, timing simulation and finally configuration. Final timing simulation uses the block and routing information from the routed design to give a closer and more accurate assessment of the behaviour of the circuit under worse case scenarios. This is the reason why the final timing simulation occurs after place and routing. This timing simulation uses the same tools as earlier functional simulation but this time the design loaded into the tool contains worse-case routing delays, based on the actual place and routed design. A script file is written to test the design, which uses the input and output points as before. The waveform viewer again allows timing information to be checked and log files are produced of the commands executed and the output produced.

Through all the above phases, report logs are generated which can be viewed to check specific details, such as logic to gate translation mapping, use of constraints, pins used or locked and timing details. Each of the separate stages, for example, produces some kind of timing detail; synthesis produces timing reports to allow evaluation of the effects of logic delays on timing constraints. Post place and routing timing reports incorporate both block and routing delays as a final analysis for post-map or post-place and route implementations. The various tools available and these various data outputs, allow static timing analysis to be very accurate.



```
Loading device for application fpga_editor from file 'v300.nph' in environment C:/Fndtn.
Building chip graphics...
```

**Figure 14. FPGA Editor**

Using the FPGA editor tool from Xilinx, it is possible to view a mapped design. This tool does not work with the configuration bitstream itself but with a file output at an intermediate point and contains detail which can only be derived from having access to the original design plus the possible implementation data. This map file can be edited and re-entered into the development cycle, finally output as a bitstream configuration file. During some points in the development of this project, it was used, initially, to get around the problem of certain components being used during normal bitstream generation. These unidentifiable components (to JBits) were manually routed around until a solution could be found.

The main display of the FPGA Editor consists of a mapping of the device's matrix to the screen, which can be viewed, edited and zoomed in upon. Any existing wiring can be re-routed manually from point to point, or a start and end location can be given, with automatic routing taking place. As there are so many resources on the surface of an FPGA and so much possible wiring, various views and levels of detail can be seen. For example, a view can be seen which eliminates specific wiring; multiplexors; CLBs or BRAMs, allowing particular information to be highlighted.

The generation of the actual placement is largely automated by machine, so being able to manipulate at this stage gives the engineer some degree of control, other than the constraints available at the start of synthesis. The use of resources can also be viewed, which can be enlightening as to how the design has been physically instantiated. Individual CLBs can be internally inspected for logic use and routing, as can IOBs and BRAMs.

After alteration of a design, it can be saved back to its original file location. When the implementation is started again on the design manager, it will detect the change in the file and start implementation from that point, developing the final output bitstream with the relevant changes taken into account.

### 4.1.3 JBits

The language choice for the proof of concept implementation was Java due to the availability of the JBits API which allows access to FPGA resources. Other ways of manipulating the bitstream may have been found but not with the degree of flexibility.

Java is also a good choice from the point of view of platform independence – many different operating systems are in use over networks. Java also contains many facets which make it ideal for network and distributed programming.

The JBits system provides a means for accessing the low-level resources of the FPGA. It also includes many useful methods for manipulating resources. The system provides ways of communicating the bitstream to a FPGA device and also tools which allow designs to be visualised and manipulated, to some extent. This section will provide an overview of its use here, in this project.

Resources available to JBits are basically most of the physical assets of the device, in the form of CLBs, multiplexors, IOB blocks, BRAMs and routing. There are a few less numerous resources, such as clocks and capture blocks[18].

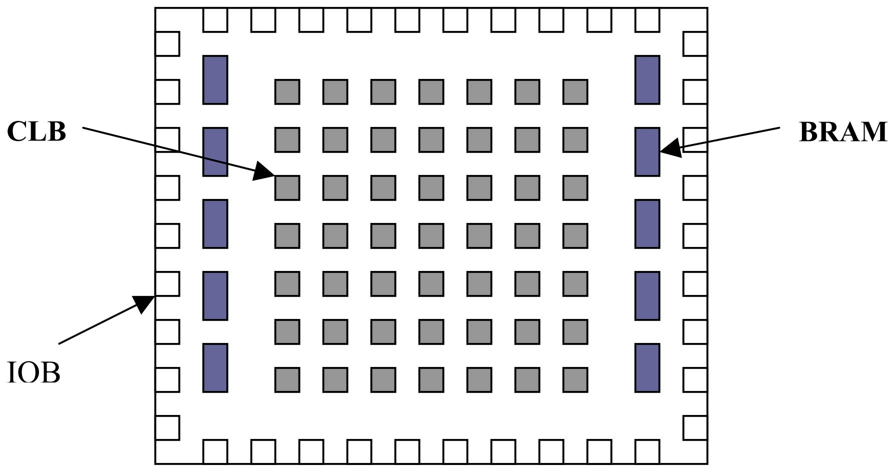


**Figure 15. Virtex device resource overview**

The basic model presented by JBits is fairly true to what is actually physically present in the device. To quickly overview resource availability: a matrix is first visualised, which is sectioned by CLB positions. At each CLB position there is the CLB block itself plus two multiplexors. One multiplexor is for input connections towards the CLB, the other is for connection to the routing, which forms the basis of interconnection between the CLBs and other resources. More multiplexors on the interconnect allow for various routing alternatives. Other resources are also accessible too, for example there are IOB positions around the periphery, as well as BRAMs.

JBits sees such resources at various levels of granularity; for example, there is the level abstraction seen above (the level of CLB blocks) and also that of slice level, where it is possible to actually manipulate resources in terms of smaller units. The CLB can be divided into two slices.

[18] Technical details of the Virtex series are available from Xilinx, Technical Overview [54] and Virtex 2.5 FPGA Data Sheet [57].

Clock pathways also criss-cross this network and are controllable through JBits. It is possible, for example, to route CLBs to any one of numerous (depending on the device in question) inbuilt clock units. Each clock can be made to run at variable time cycles, the upper limit is currently around 500MHz. This parameter is controllable within JBits. Each CLB has two clock inputs, one for each slice, which can be inhibited or switched off. A design which has already been developed and implemented using one inbuilt clock source can be re-routed using an available tool, RouteClock. It is relatively easy, in most instances, to develop programs which will do this task, although the task can become complex and time consuming to do in real time. The basic layout of clock routing is that a chosen global clock gets distributed throughout the chip and has gated buffers at each row of a few select columns. So, if you want to route to a CLB clock, it is necessary to turn on the row/column gated buffer and set the CLB clock multiplexor to use the appropriate GCLK as an input. This is equally true for BRAMs, where the gated clock buffers are located on the edge of the chip.

Resources can be set and unset. For example, multiplexors, which switch routing between areas, can be set to specific configurations, routing an input to several possible outputs. CLB resources can also be set and configured; they can be set to various logic functions and routed to clocked and unclocked outputs. The two slices which make up one CLB can be programmed independently.

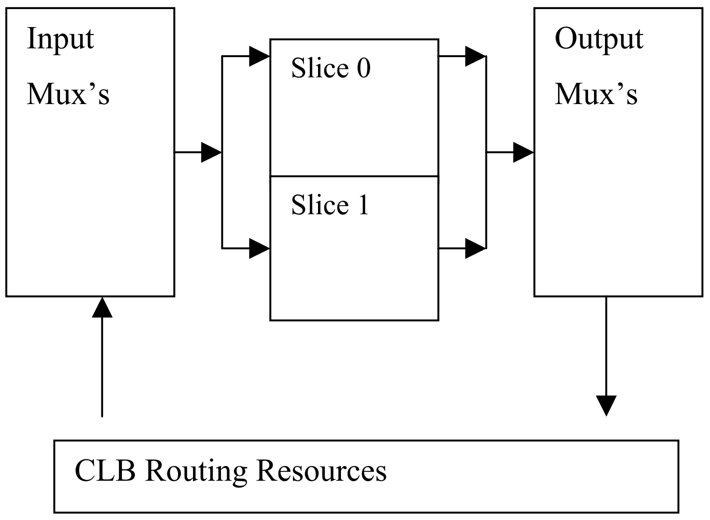


**Figure 16. Slice overview**

To access CLB resources there is a similar format for all types. For example, this code shows how the CLB multiplexors at the inputs can be accessed:

```
// get CLK
int s0clk[] = Jbits_in.get(row, col, S0Clk.S0Clk);
int s1clk[] = Jbits_in.get(row, col, S1Clk.S1Clk);

//get CE
int s0ce[] = Jbits_in.get(row, col, S0CE.S0CE);
int s1ce[] = Jbits_in.get(row, col, S1CE.S1CE);

//get SR
int s0sr[] = Jbits_in.get(row, col, S0SR.S0SR);
int s1sr[] = Jbits_in.get(row, col, S1SR.S1SR);
```

An array is returned which represents the states at the location (row, col). The third parameter is the actual name of the resource. The name is location-specific, being a concatenation of the slice and resource location, followed by the specific resource tag.

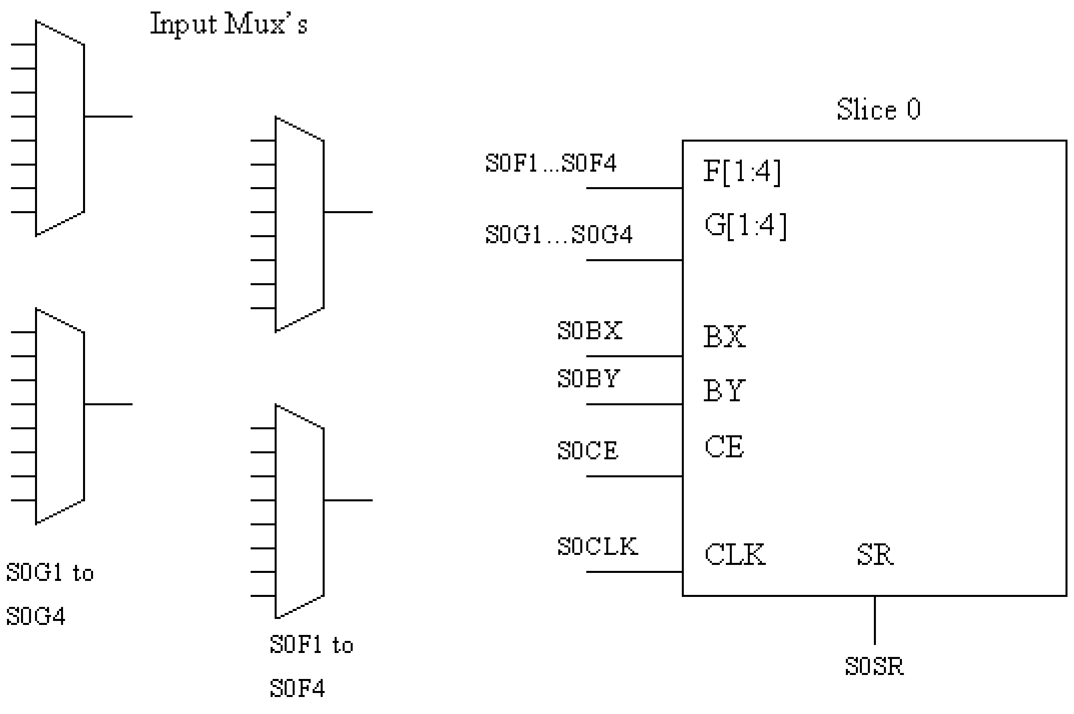


**Figure 17. Input multiplexors on slice 0**

These three code fragments get the clock, chip enable and set-reset resources. To set such values:

```
Jbits.set(row,col,S0Clk.S0Clk,S0Clk.GCLK1);
Jbits.set(row,col,S1Clk.S1Clk,S1Clk.OFF);
Jbits.set(row,col,S0CE.S0CE,S0CE.OFF);
Jbits.set(row,col,S1CE.S1CE,S1CE.OFF);
Jbits.set(row,col,S0SR.S0SR,S0SR.OFF);
Jbits.set(row,col,S1SR.S1SR,S1SR.OFF);
```

LUT values need also to be manipulated and viewed at times:

```
int lut0F[] = Jbits_in.get(row, col, LUT.SLICE0_F);
int lut1F[] = Jbits_in.get(row, col, LUT.SLICE1_F);
int lut0G[] = Jbits_in.get(row, col, LUT.SLICE0_G);
int lut1G[] = Jbits_in.get(row, col, LUT.SLICE1_G);
```

This allows the LUT values to be loaded into integer arrays, while:

```
int lut0F [] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
Jbits.set(row,col,LUT.SLICE0_F, lut0F);
int lut1F [] = {1,1,0,0,1,0,1,0,1,1,0,0,1,0,1,0};
Jbits.set(row,col,LUT.SLICE1_F, lut1F);
int lut0G [] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
Jbits.set(row,col,LUT.SLICE0_G, lut0G);
int lut1G [] = {1,1,1,1,1,1,0,0,0,0,1,1,0,0,0,0};
Jbits.set(row,col,LUT.SLICE1_G, lut1G);
```

gives an example of how they are set up.

The various internal logic of the CLB for its configuration can be set up in a similar way to that mentioned above, as can BRAM and IOBs.

Access is available to clocked outputs of the CLB flip-flops by utilising the readback configuration bitstream, together with the appropriate code, for example:

```
int slice_0_yq_value = Jbits.get(row, col, CLB_SLICE0_YQ)[0];
```

This will allow the slice 0 YQ register at (row, col) to be accessed for its content, it being the first element in the resource array.

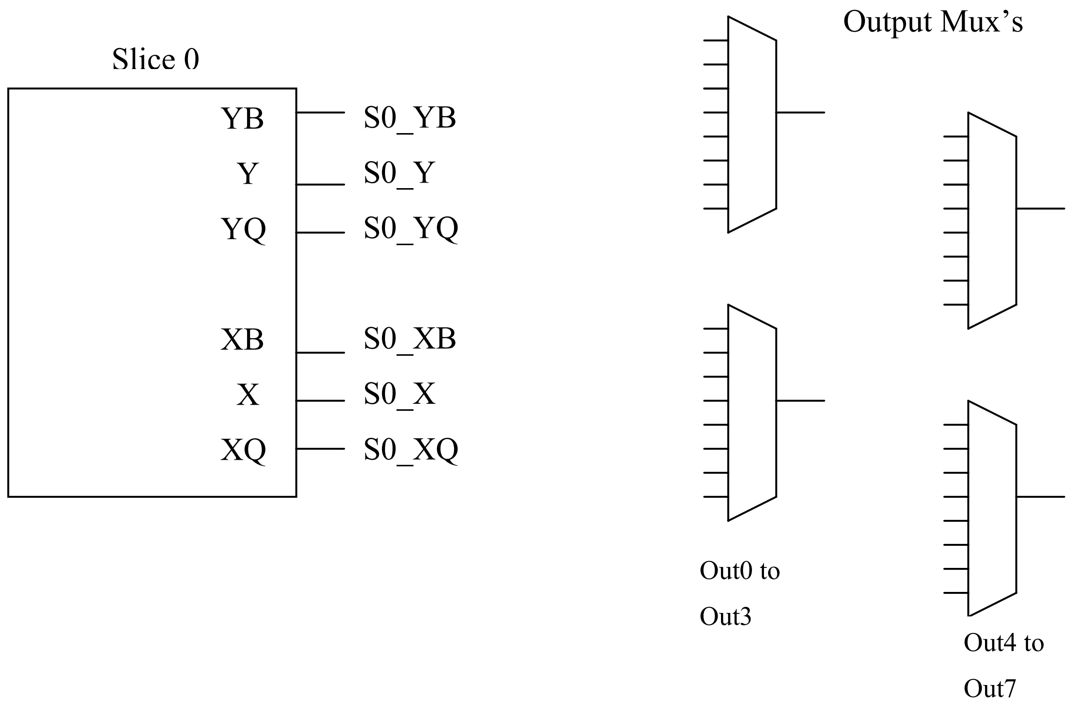


**Figure 18. Outputs and multiplexors on slice 0**

BRAMs are accessible and can be programmed with kilobits of data[19]; although small amounts of data can also be stored in CLBs, if they are configured for that purpose.

IOBs allow for connection to the outside world and are programmed with various options for input and output. It is possible to configure various buffers at such positions, to allow tri-stating of lines, or simple input/output wiring.

As well as naming resources and allowing them to be set and unset, JBits also allows control and manipulation of the process of routing. Wires can be routed from location to location, usually source to sink(s). This can be done manually, by explicit setting of routing resources, or automatically, supplying a source and sink(s) to routing methods. It is also possible to use a combination of the two. At a higher level of abstraction, routing templates can also be used to form pre-designed routes over areas.

[19] A technical description of BRAMs and their use is available from Xilinx, Using the Virtex Block SelectRAM+ Features XAPP097 [55].

Routing manually can be difficult, as there are specific resources, such as multiplexors, which conform to certain physical restrictions. There are three types of routing resources; hex lines, single lines and long lines. Hex lines are wires that run six CLBs. There are twelve of them in each North, South, East and West. So, for example, `HexNorth[0]`, starting in (row, col), becomes `HexSouth[0]` in (row+6, col), which means that you connect `HexSouth[0]` in (row+6, col) to something else. They can also be tapped in the middle, which from the same example is referred to as `HexVertM[0]` in (row+3, col). Single lines are wires that run one CLB. There are 24 of them in each N, S, E, W. Long lines run the length of the chip. There are access to two vertical and two horizontal in each CLB. They connect to other CLBs every six CLBs away. There is a twist in them, which changes their name (a little confusing). So, for example, if you connect to `LongVert[0]` in (row+12, col), becomes `LongVert[6]` in (row+18, col).

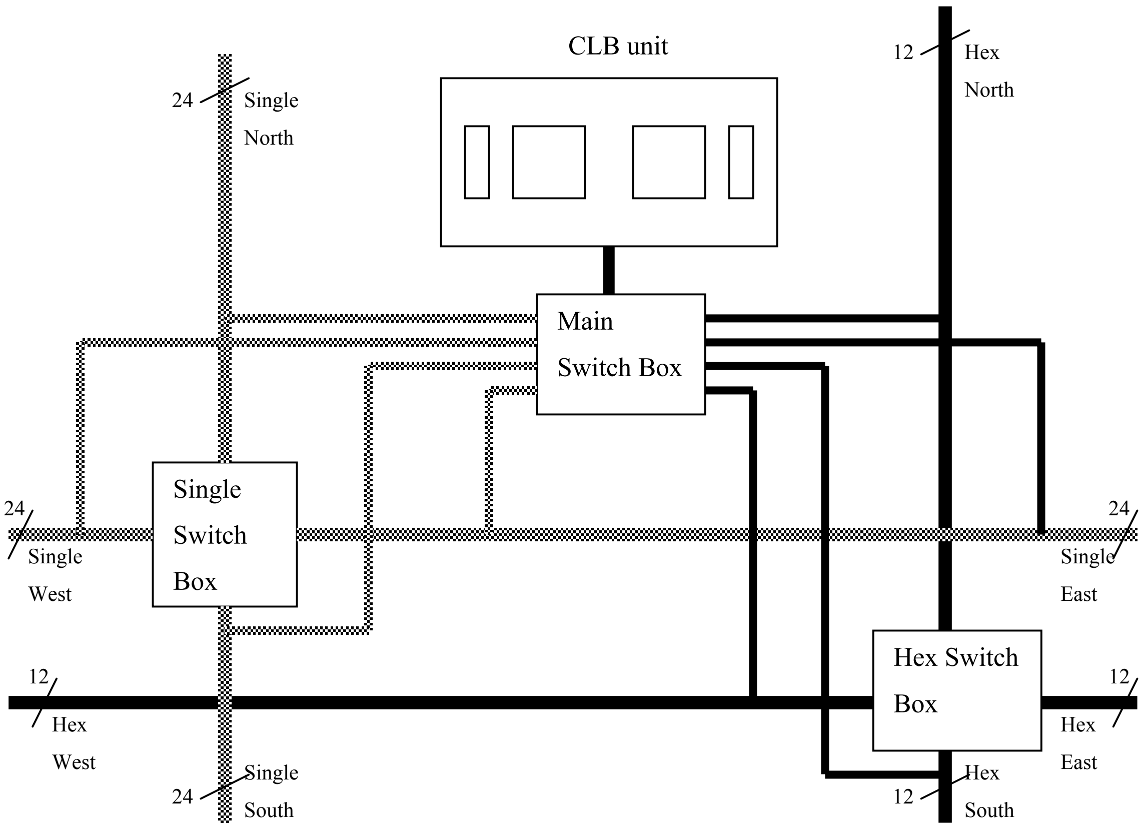


**Figure 19. Virtex routing overview**

Routing works in a hierarchical manner. Long lines drive hex lines only, hex lines can drive hex lines and single lines. Single lines can drive single lines and vertical long lines. There are eight multiplexors at each CLB which select from the CLB outputs. The output multiplexors can then connect to each of the routing resources or direct connect. The direct connect links an output multiplexor directly to an input either, one CLB east, or one CLB west. Within JBits each resource is available and wires are enumerated within classes. For example, the "wires" class enumerates each resource that you can use (i.e. `CenterWires.Single_North[0]`). There are two arrays, one called `drives` and one called `drivenBy` (for BRAM these are in `BramWiresDrives.java`). For each resource defined, i.e. `SingleNorth`, there is a corresponding entry in those arrays (using the resource as the index). The result is a list of resources that the resource either drives, or is driven by. Wire details can be also written to the console screen, using `getWireName(int wire)` to print them out.

A simple example would be useful here; to connect Hex East 1 to Hex South 2, manually:

```
Jbits.set(row, col, HexSouth2.HexSouth, HexSouth2.HEX_EAST1);

Jbits.set(row, col, HexSouth2.OUTPUT, HexSouth2.ON);
```

The second line turns on the multiplexor at that location. Using another class (Jroute2) it can done with:

```
Pin Src = new Pin(Pin.CLB, row, col,
CenterWires.Hex_Horiz_East[1]);
Pin Sink = new Pin(Pin.CLB, row, col,
CenterWires.Hex_Vert_South[2]);
JbitsConnector(jbits, src, sink);
```

To print out the JBits calls, as well as make the connections, change the last line to:

```
JbitsConnector(jbits, src, sink, System.out);
```

The format for automated routing takes the structure of the following code:

```
sinkPins = new Pin[8];

srcPin=
   new Pin(Pin.CLB,BaseRow+0,BaseCol+0,CenterWires.S0_YQ);
sinkPins[0]=
   new Pin(Pin.CLB,BaseRow+3,BaseCol+0,CenterWires.S0_G4);
sinkPins[1]=
   new Pin(Pin.CLB,BaseRow+2,BaseCol+0,CenterWires.S0_G1);
sinkPins[2]=
  new Pin(Pin.CLB,BaseRow+2,BaseCol+0,CenterWires.S1_G4);
sinkPins[3]=
  new Pin(Pin.CLB,BaseRow+3,BaseCol+2,CenterWires.S0_G1);
sinkPins[4]=
  new Pin(Pin.CLB,BaseRow+2,BaseCol+2,CenterWires.S1_G4);
sinkPins[5]=
  new Pin(Pin.CLB,BaseRow+0,BaseCol+0,CenterWires.S1_G2);
sinkPins[6]=
  new Pin(Pin.CLB,BaseRow+0,BaseCol+2,CenterWires.S0_G1);
sinkPins[7]=
  new Pin(Pin.CLB,BaseRow+0,BaseCol+1,CenterWires.S0_G1);
jroute.route(srcPin,sinkPins);
```

This supplies eight pins in an array, plus a source pin and routes the appropriate network. The route is searched for, starting at the source point and spreads out to the sink(s). It is possible for auto-routing to fail, as enough resources may not be available, or the source may be in some way enclosed. Another kind of failure is one in which the automatic routing forms a loop, though this rarely occurs and can be checked for.

Routing and the resources it connects can be investigated using the RouteTracer class. A RouteTracer object is initialised with a configuration bitstream and a RouteTree object created with a specified pin. The RouteTracer.trace method then takes the object from the RouteTree and forms a connection tree, which can be accessed and analysed. The tracer returns a structure like:

```
CLB(6, 40, S1_YQ)
.     CLB(6, 40, OUT[3])
.     .     CLB(6, 40, Hex_Vert_North[9])
.     .     CLB(6, 40, Hex_Vert_South[8])
.     .     .     [CLB(3, 40, Hex_Vert_M[8])]
.     .     .     .     CLB[3, 40, Hex_Horiz_East[9])
```
and so on.

In this printout each dot represents a level (i.e. OUT[3] is a child of S1_YQ). A square bracketed entry, such as [CLB(3, 40, Hex_Vert_M[8])], means that it is at the same resource as its parent (in this case Hex_Vert_South[8]) but at a different point on it. During routing a connection can be checked for unique sinks, using

RouteTree.unique, which are the primary connection points at a resource. Unique simply means the first time that a resource is on the tree. Bracketed points are not unique. Connection networks can be investigated both ways, that is, from source to sink(s) and from a specific sink to source. For example, the code for doing a reverse trace would be:

```
RouteTracer tracer = new RouteTracer(jbitsconfig);
RouteTree rtree =new RouteTree(sink);

Tracer.reverseTrace(rtree);
Pin src = rtree.getTop().getPin();
```

The tree can then be accessed through rtree.toString() or walked with various available methods.

Automated routing, which takes a start point and sinks, exist in the Jroute class. This uses a fanout routing method, employing a A* search technique. It uses a cost function to work its way towards the sink. A degree of flexibility is available, allowing for the amount of effort the routine will use before it gives up looking for a suitable route. For example:

```
Jroute.getFanOutRouter().setMaxLoopCount(n)
```

Allows this value to be set, the default being 1000. The loop count is how many segments it will look at before it gives up. The default value is often not enough and can be safely raised.

As well as the setting up of resources, the ability exists to reset them to a blank state. This is done by nulling them, using similar coding to this:

```
NullConfig NullCLB = new NullConfig();

NullBramConfig NullBRAM = new NullBramConfig();

NullCLB.nullClbLogic(Jbits,BaseRow+0,BaseCol+0);
NullCLB.nullClbRoutes(Jbits,BaseRow+0,BaseCol+0);
NullCLB.nullClbLogic(Jbits,BaseRow+0,BaseCol+1);
NullCLB.nullClbRoutes(Jbits,BaseRow+0,BaseCol+1);
```

It can be quite hard to determine whether an area is actually in use or not on a device's matrix; a problem which is quite crucial to this project, as it must be known whether an area is in use (for copying), or free (for placing). LUTs with no input wires and an INIT of zero can be in use as zero generators, as well as being the default configuration. If no inputs are in use and no outputs in use, the CLB can still be in use as a pass through for the carry chain. A recursive algorithm, which checks wires and internal configurations, marking components as in use or not, is the way this was solved and is discussed later.

There is the ability to create small macro objects, known as Cores, which work at a higher level of abstraction than has been mentioned so far. The Core Template package provides support for these objects to be built up to form a design hierarchically, using nets and buses. This level abstraction is useful for the average designer who does not wish to be concerned with low-level wiring and such like. It is still necessary, however, to deal with CLB logic, routing (but at a slightly higher level) and placing management. Typical objects are shift-registers, registers and adders.

It is possible to extract timing from placed or simulated circuits using JBits. Timing can be calculated using internal tools, which know various metrics for the placed components.

The actual bitstream can be investigated. The bitstream itself consists of a program which acts on a state machine, on the device for loading configurations. After an initialisation string the device data is broken into frames, which contain the settings for the various resources.

Partial reconfiguration is available to Virtex devices, using JBits. The idea here is to only make the changes necessary to a device which will bring it into a desired configuration. The partial reconfiguration model used in JBits performs this function by determining changes made between the last configuration, sent to the device and the present configuration in memory. Then it must create a sequence of packets that will partially reconfigure the device. The model will then mark the device and memory as synchronised and the process will start all over again.

The partial reconfiguration API performs these synchronisation functions fully automatically, with no intervention needed by the user. Most of this is therefore dealt with implicitly but the need for more flexibility is catered for by using some explicit calls that exist. The concept is used of marking frames as "dirty", to determine which frames need to be included in the partial reconfiguration packet sequence. A packet is marked as dirty by any use of the `JBits.set()` call, or by multiple calls of the `readPartial()` command. Flags exist that determine when a partial or full configuration exists. A partial reconfiguration can only exist if a full configuration exists and the device has been synchronised with the configuration memory at some point in time, with one of the API calls. The full configuration will be set after any full configuration has been parsed, at some point in time.

Here is a simple example which writes a partial configuration to the board:

```
/* Initialise the JBits object */
JBits jbits = new JBits(deviceType);

/* Read the bitstream. */
jbits.readPartial(inputfile);

/* Send the present configuration to the board
(XHWIF network connection to a device)
*/
board.setConfiguration(device, jbits.getPartial());

/* Make some changes to the configuration memory. */
jbits.set(...);
jbits.set(...);

/* Write the new configuration out to the board.
 * This will consist of ONLY changes made
 * between the last getPartial() and now. */
board.setConfiguration(device, jbits.getPartial());
```

The following example reads a partial configuration from the board:

```
/* Initialise the JBits object */
JBits jbits = new JBits(deviceType);



/* Read the bitstream. */
jbits.readPartial(inputfile);
```

```
/* Send the present configuration to the board. */
board.setConfiguration(device, jbits.getPartial());

/* Step the board 10 times. */
board.clockStep(10);

/* Send a readback command for the entire CLB
config. (using ReadbackCommand utility). */
readbackCommand =
    readbackCommand.getClbConfig(deviceType);
board.setConfiguration(device, readbackCommand);

/* Get the returned readback data from the board. */
readbackData =  board.getConfiguration(device,
    ReadbackCommand.getReadLength()*4);

/* Parse the readback data into the jbits object. */
jbits.parsePartial(readbackCommand, readbackData);
```

The important thing to remember about this model is the way in which the representation is held in memory, altered by your commands and finally written to the device, as above. The memory representation on a `readback()` can be updated with changes that have occurred on the board while running.

Implicit calls include; `readPartial()`, `writePartial()`, `parsePartial()` and `getPartial()`. After a `readPartial` command the configuration memory will match that of the bitstream file, after the completion of this read. If no configuration exits in memory, a subsequent `getPartial()` call will return a full configuration to use to program the device. If a configuration already exists in memory, the differences between the file read and the configuration in memory will be marked as dirty. However, if these frames are already marked dirty from a previous read, they will remain dirty. Another `getPartial` will return a partial reconfiguration sequence consisting of the dirty frames. The `writePartial` method writes out a partial configuration consisting of dirty frames. A call is also made to the `getPartial` method. This marks the configuration memory and the device as synchronised, by clearing out all the dirty frames. Full configurations can therefore be obtained by using the `clearPartial()` explicit call, as in the case of `getConfiguration()`. The `parsePartial()` method is used for parsing a readback bitstream returned from the device. Commands which obtain the readback stream are also required to parse the data appropriately for both partial and full readbacks. The `getPartial` method will return a sequence of packets that can be used to configure a device. The returned packets consist of a partial configuration of the marked dirty frames, or a full configuration, if the partial reconfiguration flag has not been set. A NULL is returned if no frames have been marked as dirty.

Explicit calls exist which allow more flexibility but must be used with care. The method `clearDirtyFrames()` clears all frames marked as dirty, effectively synchronising the device with the configuration. To clear the partial reconfiguration flag there is the `clearPartial()` call, which will result in a forced full configuration on the next `getPartial()` or `writePartial()`. The `clearFull()` method clears both full and partial reconfiguration flags, erasing the configuration memory and allowing the object to start from scratch. Given a packet configuration, the call `compareClbConfiguration()` will compare the input packet with the existing CLB configuration memory and mark the differences as dirty frames. The configuration memory does NOT change, this is useful for comparing the two and then downloading with the `getPartial()` method to synchronise the device. A CRC check can be enabled or disabled with the `enableCrc()` call, for partial and full reconfigurations, returned by `getPartial()` or `writePartial()`. This has the side effect of slowing down operations. The `enableSoftReset()` method enables soft resets after each partial reconfiguration. The reset forces all registers to their initial states, after a partial reconfiguration, where they would otherwise remain in their present state.

JBits also provides the means for transmitting and receiving bitstreams, known as XHWIF API, over networks utilising TCP/IP. Basically this involves the setting up of a connection of the device with the server software provided, after which it becomes possible to communicate with the device, using several built in (to JBits) functions. It is possible to send a configuration bitstream and activate it; set the clocks to frequencies and turn on or off; and to cause a readback, if the capture blocks are set in the design or reset the device entirely. Any configuration can be done dynamically while the device is active and running, without disturbing any occupied circuitry (unless, of course, this is what is required). The smallest unit of exchange is one frame, the size of which varies between devices.

XHWIF API provides a generic set of methods to interface with FPGA hardware. These method calls can be used to communicate with a board, once the board interface is obtained. The `XHWIF.get()` is used for this purpose. XHWIF can be used to connect to local and networked hardware, as well as boards which contain more than one FPGA device. As an example, this code describes how a board can be connected to:

```
String boardName ="rc1000pp";
XHWIF board = XHWIF.Get(boardName);

/* Get the remote host name (if we have one) */
string remoteHostName =
   XHWIF.GetRemoteHostName(boardName);

/* Get the remote port number (if we have one) */
int port = XHWIF.GetPort(boardName);

/* Connect to the hardware */
result = board.connect(remoteHostName, port);
```

Different devices can be supported and accessed on the same board:

```
/* Get the Device and Package Type on the Board */
int[] deviceType = board.getDeviceType()
int[] packageType = board.getPackageType()

/* Get the Number of Devices on the Board */
int deviceCount = board.getDeviceCount();

/* Create a JBits Object for the Device 0 on the board */
JBits jbits = new JBits(deviceType[0]);

/* Read Bitstream */
jbits.readPartial(inFileName);
```

This code sets the program up to communicate with device 0 on the board.

Reading back the configuration from a device is relatively simple:

```
/* Get the Readback Command to read the CLB Data */
byte[] readCommand =
    readbackCommand.getClbConfig(deviceType[0]);

/* Get the readback data's length in Bytes */
int readbackBytes = (ReadbackCommand.getReadLength() *4);

/* Send the readback command */
board.setConfiguration(0, readCommand);

/* Readback the Data */
data = board.getConfiguration(0, readbackBytes);

/* Parse the Readback bitstream into the JBits */
jbits.parsePartial(readCommand, data);

/* write the readback data to a bitstream */
jbits.writePartial(outFileName);
```

Readbacks and configurations can be applied in such a way as to only affect changed resources. This limits the size of the bitstream, which is transferred in both cases. A model held in a computer of the actual device is updated with any changes. Likewise, any changes which are sent, only occur to the appropriate locations and resources. A resource which has changed while running can be made to trigger a readback and then the alteration viewed on the host computer.

JBits has various tools built in to allow observation of activity on the device itself. The Boardscope program, for example, gives access to various views, such as state view (which allows the contents of flip flops to be seen), routing density view, (showing routing density in the form of coloured mapping), or power view.

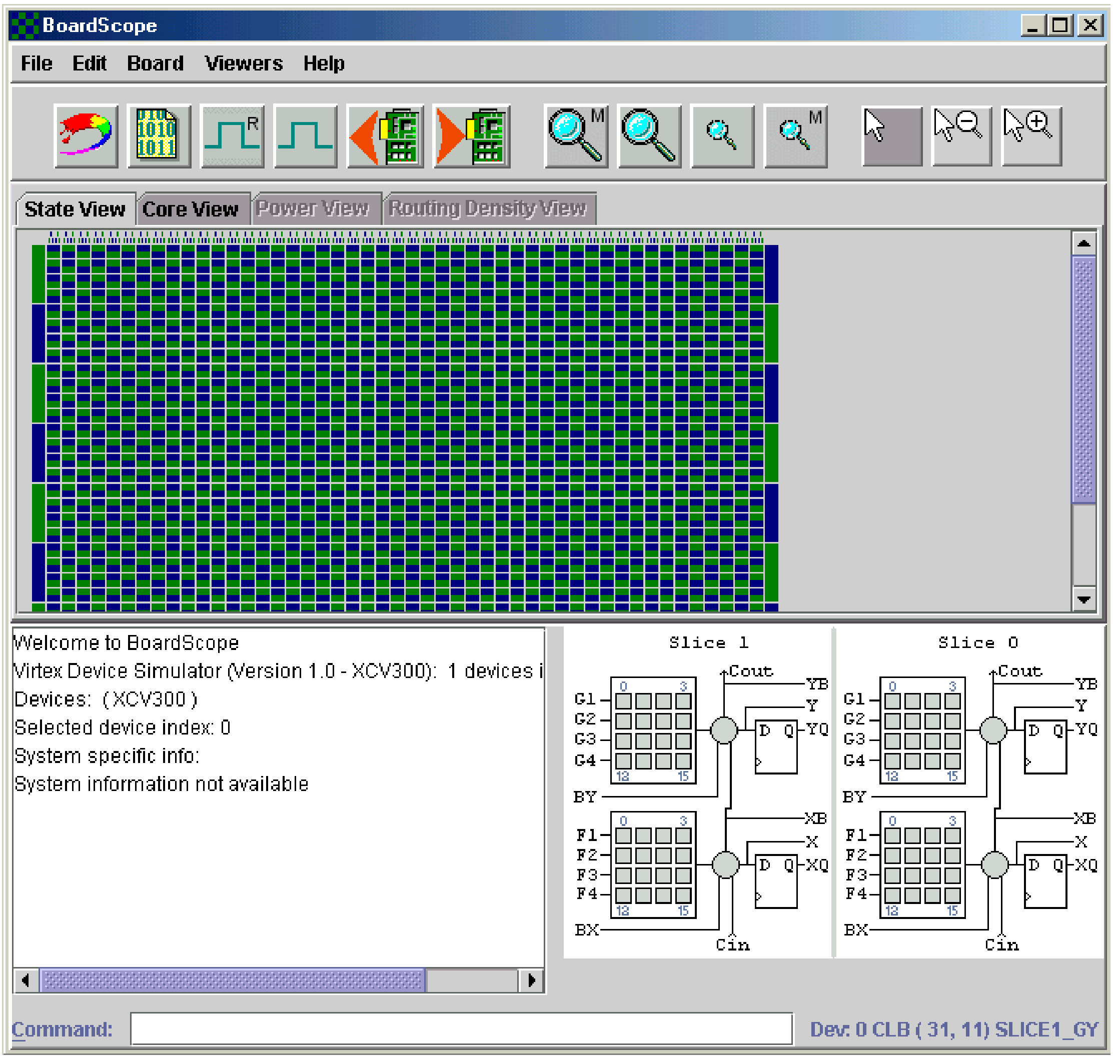
In the state view, CLB locations can be viewed and investigated. Routing can be traced to some degree. BRAMs contents can be probed by clicking on their graphic representation, the output is displayed in a text area in a hex format. IOBs, which are present round the outside of the device, indicate their state by colour. Each IOB has three flip-flops: IN, OUT and TRI. The four flip flops available at each CLB point are colour indicated, depending on their state. Clicking on a CLB in any view allows the detailed CLB view to be available at the bottom of the screen. This display shows the contents of the LUTs and clicking on an input or output reveals the source or sink connection in a text box. The display is updated whenever the device is reset, a new bitstream is configured and the device configuration is read back.

Using the density view it was easy to be able to check the positioning of processes and by utilising the routing tracing from the selected CLB point connections could be easily verified. The actual routing density of a design can be verified and is colour coded. White indicates no wires used, white to green indicates 0 to 50% and green to red indicates 50% to 100%.

The power view enables the user to visualise the power consumed in the CLB flip-flops. Power is measured for each flip-flop by calculating the total number of flip-flop toggles and dividing it by the number of clock cycles. Power is measured exclusively by the toggle rates. Flip flops that toggle every clock cycle are shown having high power, while those toggling less frequently, with respect to the clock, are shown having lower power. The various toggle rates are indicated by having a colour scale from white to red (high).

JBits includes a device simulator known as the VirtexDS. This simulates at the device/bitstream level, providing an interface which operates like the actual target hardware. This approach supports not only simulation for run-time reconfiguration but also interfaces to JBits applications, such as BoardScope and the XHWIF server. The simulator allows the user to develop and test designs, without the need for actual hardware. By utilising and implementing the XHWIF portable interface, many different components of the JBits system can be “plugged in” to the simulator, as well as user written programs. A variety of devices are supported for simulation from the relatively small XCV50, up to the XCV1000. For test purposes there is the Stimulus/Probe interface, which allows direct access to simulator resources, rather than using readback. This is useful for testing arrangements or “test benches”, prior to using on real boards.

A basic server application is available, which once set up on the host machine containing the device, allows connection through TCP/IP.



Welcome to BoardScope
Virtex Device Simulator (Version 1.0 - XCV300): 1 devices i
Devices: ( XCV300 )
Selected device index: 0
System specific info:
System information not available

**Figure 20. Xilinx BoardScope application**

BoardScope is a graphical, interactive tool for debugging designs on Virtex FPGAs or simulators. It allows the subject of investigation to be a simulation of a board, or an actual FPGA device situated locally, or at some remote location. Communication is performed through TCP/IP. Simulation capabilities are actually built in to the BoardScope and server applications, allowing a user access to a virtual XCV300 device. This allows a user with no hardware available to develop designs and act on the simulation to check behaviour. The simulation is fairly close in most respects to the real board; certainly to date, everything developed for this project has worked on the simulation and hardware in a similar way.

At the time of developing this project, much of JBits was still evolving, so a lot of functionality had to be written to make up for areas which were lacking. An example of this was not all resources on the surface of a device being identifiable and therefore out of use by the JBits classes. Resources such as “pips” and various small connectors, which may have been used in a standard design too during the implementation cycle, were not able to be identified by JBits, leaving routing network connections “hanging” with a missing source or sink. This whole problem and its solution, will be looked at in more detail later.

While JBits provided a means of low-level access to resources and to dynamic reconfiguration options, many methods had to be developed for manipulating processes as a whole and the individual components contained within it. With the JBits classes providing access to low-level resources and routing, there was still a question of handling various collections of resources, which form a process, a potentially very complex, physical object. Methods were required for copying CLBs and associated wiring within and outside of their original bitstreams. Mechanisms were also added for manipulating connection trees, BRAMs, physical pads and the capability to read in hex data files.

Many high-level features of JBits were not required but can be integrated if so desired, for example, “cores” can be turned into MHPs for manipulation in real-time at a high-level.

The purpose of this section was to give a comprehensive background to the implementation of the Reconnetics system, which detailed the tools used and the way in which they were applied. It also served as a brief introduction to the usage of the main programming extension added to the project, JBits. This provided low-level access to FPGA resources, as well as many useful utilities, for example BoardScope and the Server software.

At the time of development, some aspects of the JBits system were still being resolved. This required that patches were built into the system, to allow for unidentifiable resources to be dealt with.

## 4.2 System Implementation

The following section presents an implementation of a run-time system for the manipulation of design elements, known here as Mobile Hardware Processes. This system is named Reconnetics (from Reconfigurable, network, interconnect). The following section considers how the system is constructed. Further details of all classes used are given in Appendix III.

### 4.2.1 System Overview

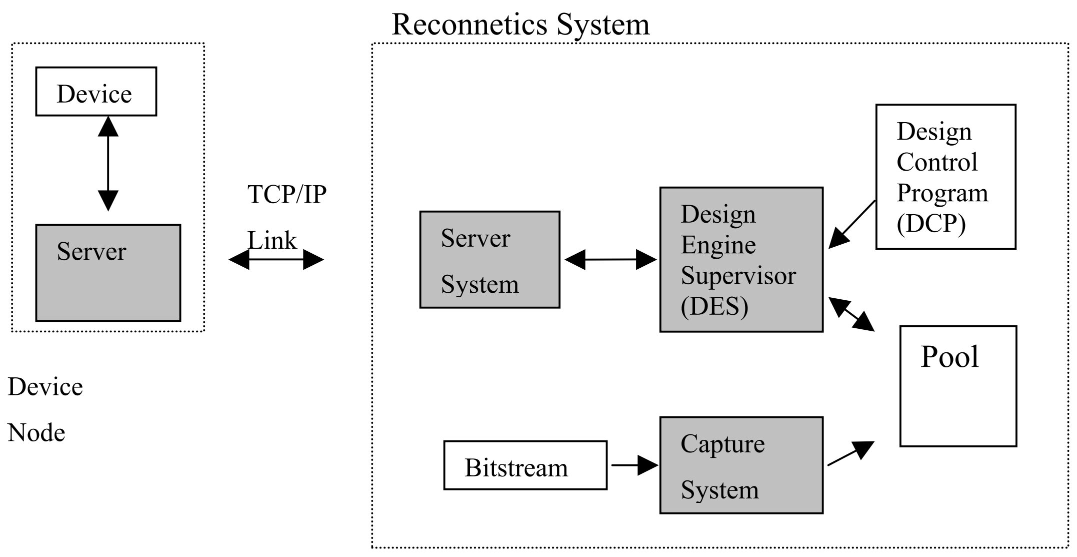


**Figure 21. Reconnetics overview**

The Reconnetics system allows dynamic interaction with a reconfigurable device. The user supplies designs, as a synthesised "bitstream", written in any given HDL or graphic tool. The design itself can be as large and as complex as required, although, it would make more sense to build from smaller blocks, compositionally. These are captured and placed in an archive, called the "Pool", for later utilisation by the design engine.

The engine is directed by a high-level user program, to load, place and interact with these processes, as well as manage on-board resources.

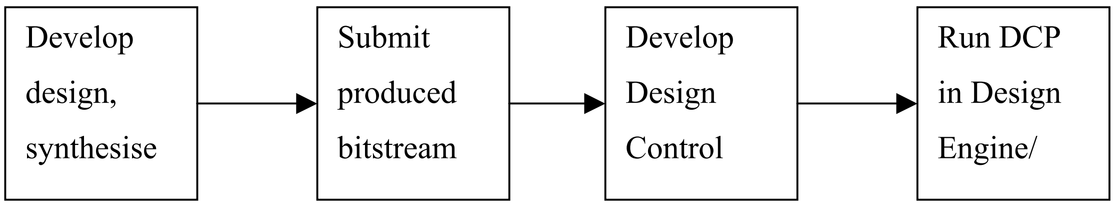

Develop design, synthesise
Submit produced bitstream
Develop Design Control
Run DCP in Design Engine/

**Figure 22. Development cycle**

The language used to control the engine is deliberately simple and focused at a high-level; all low-level wiring and resource handling is taken care of by the system itself. A user needs only to decide which processes need to be connected together and what kind of interaction is required. It may be that only placement is desired, or diagnostic work. This is possible but a wider scope of control and communication is also offered. For example, processes can be interacted with by Reconnetics in such a way that a user can supply input directly to the hardware, without external I/O connections, or other methods. This is true also for collecting output, making the system particularly suitable for development work.

The capture system analyses a bitstream, which can be the product of any design method or language (HDL, schematic etc). Its task is to produce a Java class which can rebuild the circuit at any given point on the surface of an FPGA. This includes status of clocked logic, routing and any other necessary resource usage. The class is primarily produced for utilisation by the run-time system but is sufficiently self-contained to be incorporated in any user written Java program too. After capture, the class is placed in an archive known as the “Pool”, which is simply a directory.

The Design Engine/Supervisor (DES) manages the device itself, along with organisation of the processes. It can run user-written scripts which direct activities, these are known as design control programs (DCP). All low-level resource management is left to DES, making the actual command language very simple, an example is given later. The language also includes various diagnostics, if required.

MHPs are created by the capture system. They are composed of two main structures; the Java program and the physical device circuit. The Java program's principal function is to act as a vehicle for the circuit and to reproduce the circuit at any given point on an FPGA matrix. Another function of the Java program is to act as a mediator between DES and the physical circuit. A user can communicate with the process in this way, collecting information, or directly manipulating the circuit's form or connections.

The components of the system will now be detailed more fully.

### 4.2.2 Capture

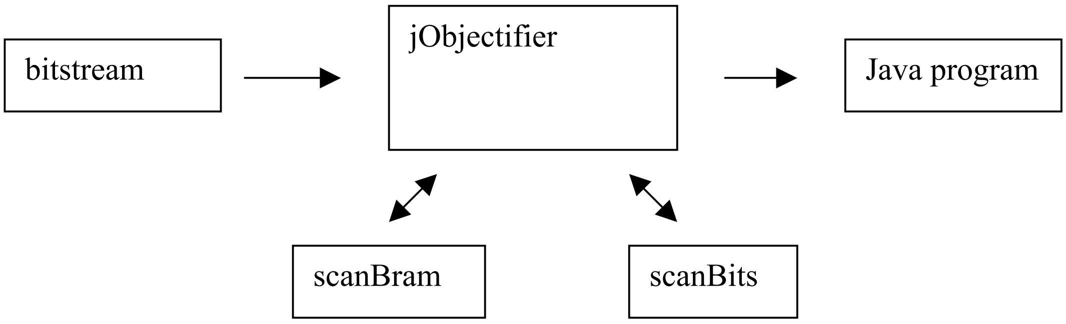


**Figure 23. Capture system showing main classes**

As stated above, the main task of this sub-system is to produce Java classes which can reproduce a circuit at some other location. It must capture all essential design data, such as the status of multiplexors, look-up tables (LUTs) and routing. A large number of resource types are available in devices – some of which are not mapped into JBits at this time.

During analysis of a circuit, this led to incomplete route tracings. This was resolved by building in a certain amount of design knowledge. A first scan by the system makes a list of incomplete nets. Solutions to an incomplete net can be found by recognising specific attributes of the end resource and by being aware (through previous scans) of disrupted routes elsewhere, which conform to certain criteria. In this way the system can apply a patch to fix an incomplete tracing.

A net tracing may run, for example, from west to east, stopping at a multiplexor. Another incomplete tracing may be found at a position east of the multiplexor, originating at a valid source point, such as an I/O pad and buffer. By considering each incomplete net in turn, together with its attributes, it is possible to rebuild connections. The two incomplete nets in this case would have their end point resources analysed. The multiplexor has certain constraints as to what directions it can connect to and how far away the connection is made to. This at least gives a possible direction and general idea of connections available. A search through other dangling connections in that area will present possibilities. A candidate for connection can then be likewise analysed in the same way, observing the dangling end point's constraints and directions or linkage connections. A degree of probability can then be derived and the candidate connection with the greatest certainty can be routed.

This method has not yet been known to fail in its attempt to re-establish a problem design. It is possible that this technique could be applied to correct corrupted bitstreams, or instantiated designs, up to a certain point.

A further optional stage is available, called "pre-capture", which adds connection points to a process, allowing several modes of communication with other processes, or external I/O. Further detail is provided below.

### 4.2.3 Scanning

Here resource usage is determined and mapped. Three different areas are checked; the CLBs, BRAMs and external I/O. The supplied bitstream is analysed; firstly, each CLB is checked for used resources and, if found, the matrix position is marked for further examination. This stage is potentially very difficult. Problems arise because it is possible that a CLB may be in use, although not obviously. This may take the form of a CLB resource being in "passive" use. An example of this is that the CLB could be used as part of a carry chain, even if no inputs or outputs are in use. The LUT content may be deliberately zero for some purpose, such as a zero generator, or be part of a default configuration. In such cases, further levels of analysis are required, which look at the surrounding circuitry.

CLBs are subjected to a specific order of scrutiny. An increasing level of sophistication is applied in tests, quick tests are applied at the beginning to speed up detection. All that is needed at this stage is to find out whether a CLB is to be included in a design. To detect whether a CLB is at all in use, firstly, the LUTs are checked for values. If no LUTs are found that indicate possible use, the next stage, checking wires, is begun. This checks for connections to the CLB unit from other resources. It also tests the integrity of such routes, checking that they have a valid sink or source and are not left "dangling". If this stage does not indicate any use, then the carry chain is finally checked for connection to other units.

All CLB units detected as in use are noted in terms of their CLB row and column positions. When the entire matrix bitstream has been scanned, an object is built which contains the details of the CLB resources and their relationships to each other, such as positioning. This object is a sub-matrix of the larger containing matrix within the bitstream. The sub-matrix contains smaller units representing the individual CLBs. The sub-matrix can be manipulated in terms of its geometry, it can use its default shape, or have new positioning imposed on it, while still maintaining correct connections.

The next stage looks at which BRAMs are in use, simply taking a note of their position for later processing.

The sub-matrix object now contains positional and resource information, covering used CLBs, BRAM and I/O.

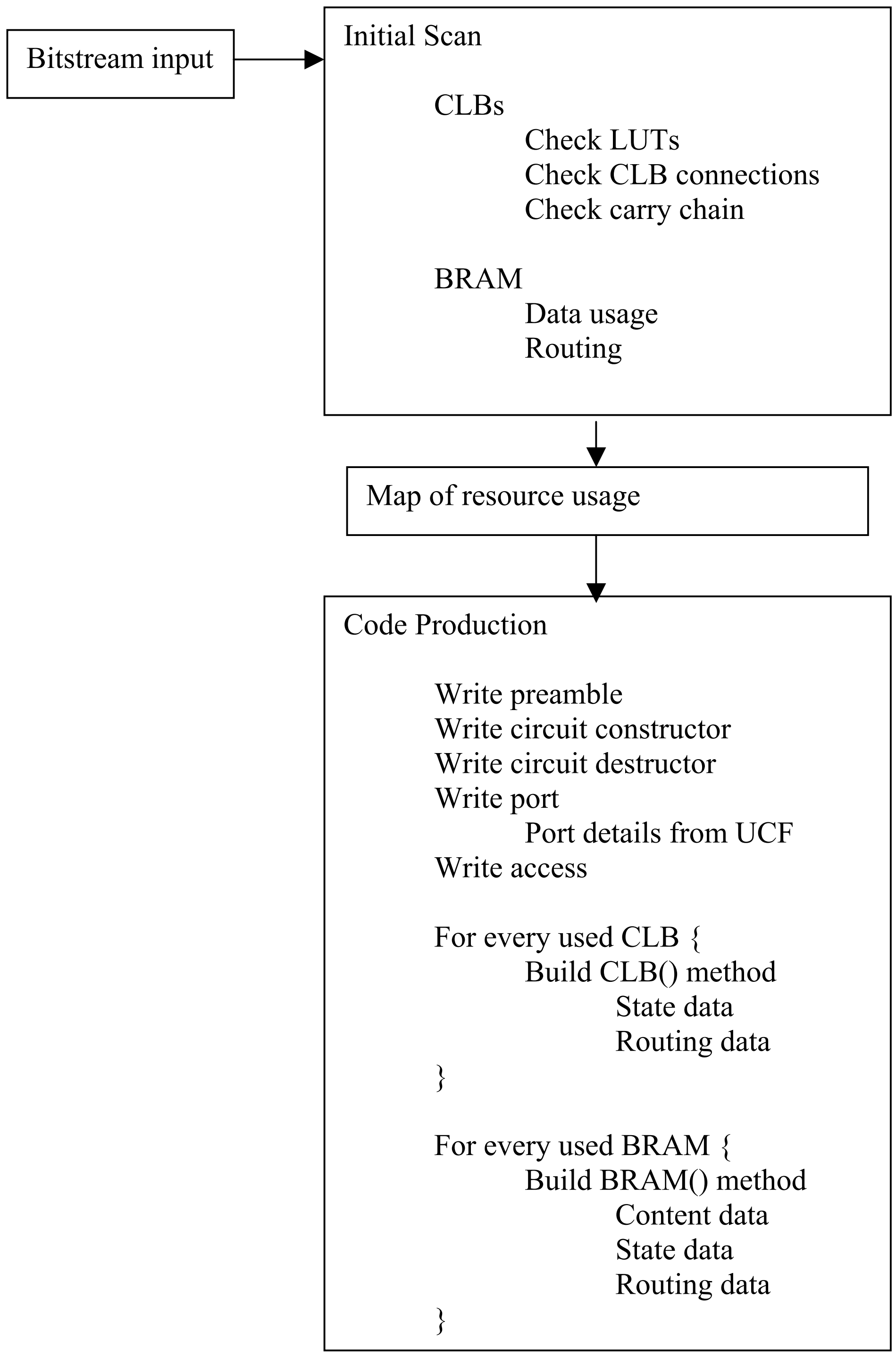


**Figure 24. Capture system**

### 4.2.4 Code Production

The scanning procedure gives a basic map of the CLB and BRAM usage. This map is then used to analyse, in greater depth, specific resources and their settings. Finally, a program is generated which describes the implementation. There are many types of resource contained within each of the basic areas, whose state has to be captured and translated into program statements, such as multiplexors, switches, LUT arrays and routing, some of which may be to external I/O points.

The format of the Java program produced is broken into several key methods. These allow a control program to build, destroy, communicate and route a process's physical circuit.

The main methods in the Java program are as follows:

*Build()*. This builds the process at a given position in a device's matrix. An example of this from the `pulsecounter` process is:

```
public boolean build (JBits Jbits_in, int Row_in, int Col_in)
{
      boolean place_ok = true;
      int max_row_idx = Jbits_in.getClbRows()-1;
      int max_col_idx = Jbits_in.getClbColumns()-1;
      int max_bram_idx = (Jbits_in.getClbRows()/4)-1;
      if ((Row_in+row_Size)>max_row_idx) place_ok = false;
      if ((Col_in+col_Size)>max_col_idx) place_ok = false;
      if (((Row_in/4)+num_BRAM)>max_bram_idx) place_ok = false;

      if (place_ok) {

            Jbits = Jbits_in;
            jroute = new JRoute(Jbits);
            jroute.getFanoutRouter().setMaxLoopCount(20000);

                  BaseRow = Row_in;
                  BaseCol = Col_in;
                  pulsecntCLB041();
            }
            return place_ok;

      } // end build circuit
```

A bitstream is passed, together with the build row and column origin points. Various checks are implemented, such as free BRAMs (if required) and the fit of the process. If this is all right then a placement is attempted. In this example the actual building is done by the `pulsecntCLB041()` call, see `CLB()` below for details.

`Destroy()`. This releases all resources – multiplexors, routing, LUTs etc. for that process. For example:

```
public void destroy() {

  NullConfig NullCLB = new NullConfig();
  try {

     NullCLB.nullClbLogic(Jbits,BaseRow+0,BaseCol+0);
     NullCLB.nullClbRoutes(Jbits,BaseRow+0,BaseCol+0);

  } catch (Exception ce) { System.out.println(ce); };

}
```

`Port()`. By using the port method it is possible to direct a process to connect, via physical routing, to external pads and other processes internal to the device. These can be default locations, or specified by a user. It also allows information about a port to flow back to the supervising program.

`CLB()` methods are called by the above mentioned `build()` during construction. They contain the physical implementation details for each used CLB.

`BRAM()` methods, when called upon, implement the described BRAM into a specified, or automatically generated, location.

Other minor methods are available for information purposes, such as to return the row and column size of the process, together with how many BRAMs are used.

From the point of view of the average user of the system, the above method details are not required. DES and the command language take care of such detail. This information is useful if the process is to be used by a Java programmer, outside of Reconnetics.

In the case of CLB areas, each resource within a CLB location is looked at and the appropriate statement is written to the file. The particular resources included in such a location are:

- Clock, chip enable, set/reset selects.
- LUTs (four in total, two in each slice).
- Internal switches/muxes, such as carry select, control inputs, clock invert, chip enable invert, data input select, LUT mode selection, write enable etc.

CLB areas also contain routing. Beginning and endpoints are noted and auto-route statements generated. It is also possible to produce code for routing "templates", rather than start/end points. This template can then be moved and placed anywhere across the device, using relative coordinates. The benefit being that a precise routing is generated, although at the expense of a larger program file. External port points are not linked but left open at this stage.

A `Port()` method is written in such a way that external I/O points can be connected to dynamically by the user. The signature of the `port()` method reveals some of the detail:

```
public conPort port(String portName, Pin cxPin, int mode)
```

By using this method, it is possible to instruct the circuit's I/O to be connected, dynamically, to any given point on the FPGA. The different mechanisms available are accessed through instructing the `Port()` method with different values in mode:

DELETE (mode=0) – deletes a named route starting at portName.
ROUTE (mode=1) – routes a port to a destination, stored in cxPin.
DEFAULT (mode=2) – routes a port to a default connection point.
INFO (mode=3) – retrieve port information.
CLOCKED (mode=4) – retrieve the clocked version of a port.
NAMES (mode=5) – retrieve names of ports on this circuit.

Using these various modes it is possible to get information about any particular named port and route or unroute it. The DEFAULT mode allows a named port to be linked to its design-time point, which is useful for having a specific I/O set up installed every time for interface to external circuits, for example. INFO mode allows data about a named port to be returned as a `conPort` object. This contains the port's details, such as its width and default connections. CLOCKED mode allows a version of a port to be returned, which is always linked through a clocked register, overriding its default status, which may be a simple, straight-through connection. Finally, NAMES mode allows access to the circuit's port names as strings. This is used for retaining labels if the circuit is recaptured, so they persist through to the new design. More details on this follow.

It is only necessary to trace output routing from a CLB since the only other place that incoming wires may be from is IOBs or BRAM, which are dealt with separately, being less numerous. Any used inputs are therefore found as a result of being traced from another CLB.

The basic "shape" of the design is kept intact, that is, the map which shows resource usage also holds the basic relational details. At the time of code production such detail is captured by producing offset row/column data between each CLB and a point of origin. The entire design can then be relocated anywhere on the surface of an FPGA, while maintaining its relational and connection constraints. This ensures that issues such as timing and gate propagation, are met, as they were when initially compiled. It is possible to override positioning data for CLBs, if necessary.

Used BRAMs are analysed and separate methods built for their construction. There are three main resource areas; settings for logic (switches, mux), data content and routing. As before, in the case of the CLBs, appropriate statements are generated to implement correct settings. Data content in the BRAM is translated into an array format, held in the method, which is downloaded at build time. Finally, statements are written for output wiring. This connects the BRAM to its owning process. It is important to note here that it is possible to position the BRAM at any BRAM location and it will still be wired to correctly. This makes it easy for a run-time system to place and route such resources in whatever way is best suited at the time.

In all the above generated code, no routing is generated that leads to external IOB positions. External I/O is scanned though, for connection points to the internal circuit. Each IOB is looked at in turn. Inputs and Outputs are checked for valid routes to previously captured CLB positions. Tracing routes (using JBits) is a question of working through a tree of connections from source to sink(s). It is also possible to trace in reverse, which is necessary when looking at the output connections. Each net is checked for its uniqueness of connection to an I/O point and that such a link is valid (that it is a true source, for example). Each active I/O point is eventually given a port object containing its relevant details; such as start point, direction and end point(s). This data binds together the IOB, its net and the CLBs it is linked to. Obviously, the duplication of data is possible with a high number of net wires and physical routing problems; such as recursive loops. Details are checked, as noted above, for uniqueness and integrity. The information gathered here is used as default connection data, stored in the generated program.

### 4.2.5 Input/Output Connection Points

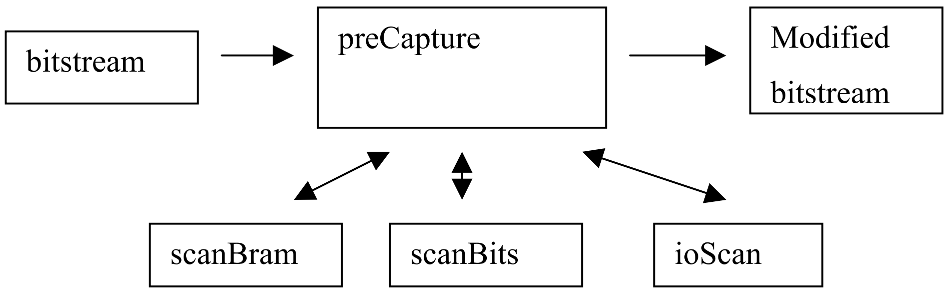


**Figure 25. Pre-capture system showing main classes used**

It is possible to pre-process a bitstream, so that any I/O is passed through a special connection point. This pre-processing stage is particularly useful for designs which already exist (for example, as a legacy design) and you wish to turn into a fully self-contained process. It is also possible (though not as simple), to manually add a connection point, using the system itself at run-time. In both cases a special system object known as `procio` is used. This unit added in is actually very small, being only a quarter of a CLB for each wire.

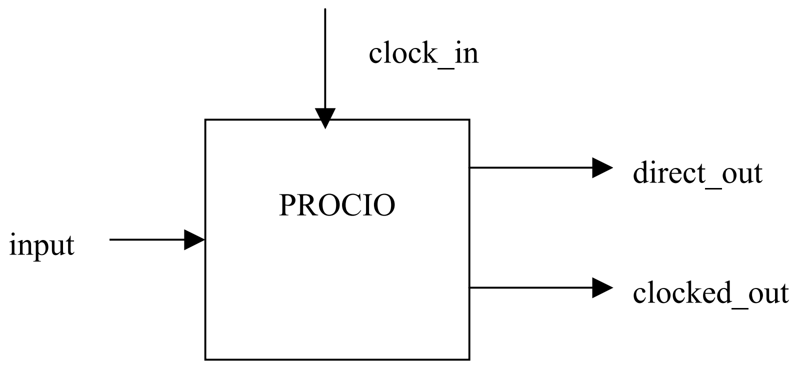


**Figure 26. PROCIO unit showing main connections**

There are three functions for the I/O points; to decouple, to simplify routing to other processes and finally, to act as a controllable gateway to the world. The idea here is to allow a degree of control and buffering at I/O points. When this is done it is possible to control the connection point with commands, such as `inputON` and `inputOFF`, described in the next chapter. The functionality of the `PROCIO` object allows I/O signals to be disconnected, clocked, or fed straight through.

The connect point allows internal (process) wires, which act as I/O, to be gathered in one location. Even one wire from a logical/user perspective can equal many physical wires, which would have to be re-routed individually every time a new connection was requested, using both time and extra resources. The connect point groups these wires and allows a single wire or reduced bus to take its place. A process, in this way, rarely needs internally reconfiguring or re-routing.

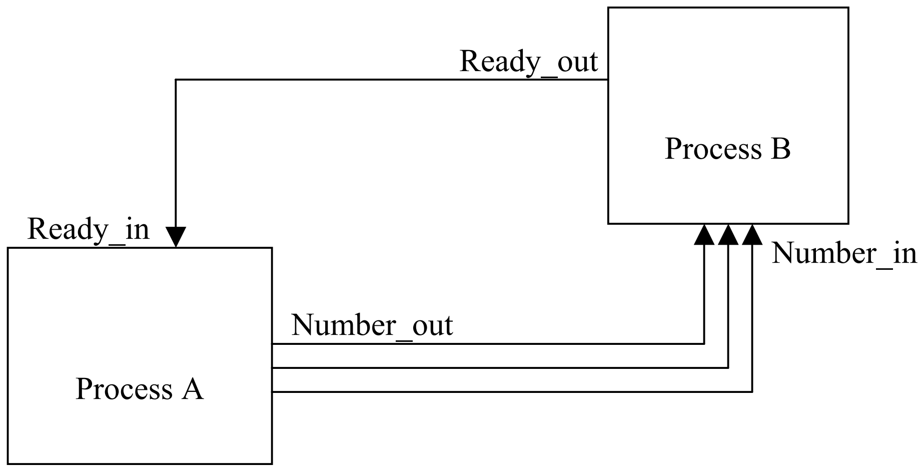


**Figure 27. Process buffering (before) showing logical user view**

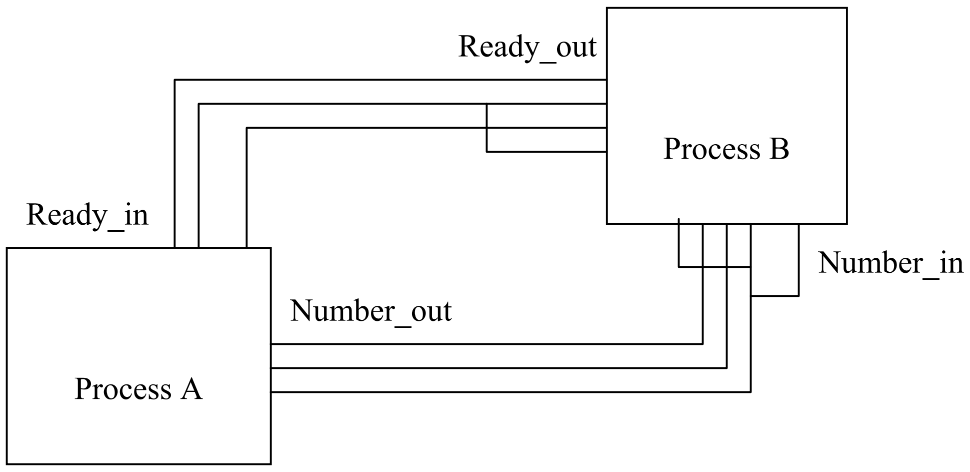


**Figure 28. Process buffering (before) showing actual physical view**

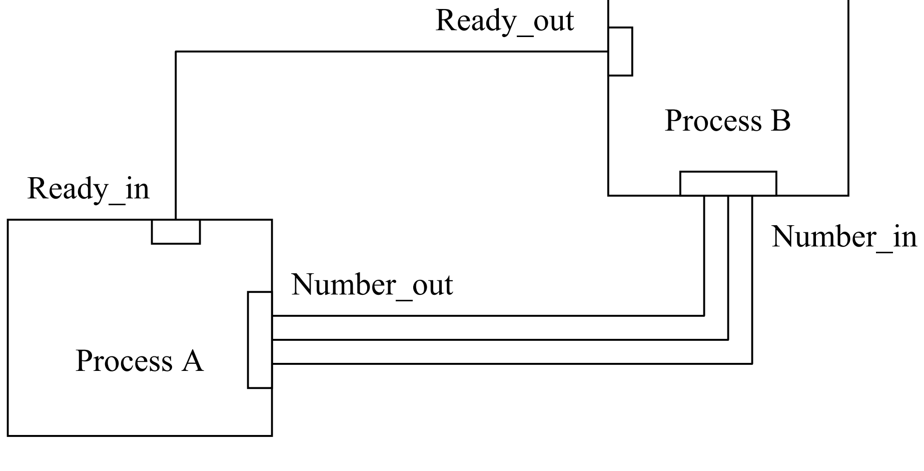


**Figure 29. Process buffering (after) showing I/O buffers in physical view**

The previous three figures illustrate how the I/O connection points simplify routing. Figure 27 shows how the user perceives the channel-based wiring in a logical sense, at a high-level of abstraction. If precapture was not used then a large number of wires would run from process A to process B, making the design utilise a greater number of resources over a larger area and take longer to place. This can be seen in Figure 28. When I/O connection points are added within the process, internal I/O routing is passed through them, reducing the wire count, as seen in Figure 29.

There are two ways to link to a connect point; as a direct feed through, or as a clocked register. A simple method of connection choice is available, by using the name of the location, followed by `_clk` for a clocked connection.

A connect point can also be switched on and off to allow isolation, particularly during process configuration and interconnection, while the device is active. Yet another possibility is the ability to select values at the end of such connections, which is useful, particularly in the case of run-time testing.

This stage of adding I/O units was left as optional, to allow maximum flexibility for the user, who may be considering utilising the system in a way that does not require the control features available at run-time. This may be for a simple place and route, for example.

### 4.2.6 Recapture

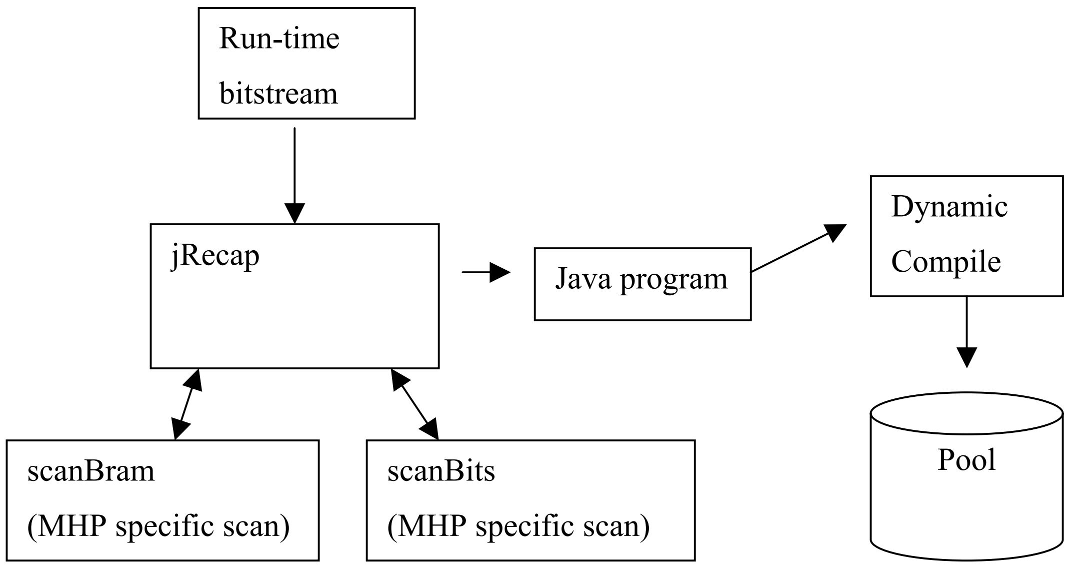


**Figure 30. Recapture main classes**

Recapture is the name given to the ability to capture a process at run-time. Many of the techniques used in the pre-run-time capture class, `jObjectifier`, are reused within recapture, although obviously initial design details are not available (such as the data contained in the UCF file, giving a port description). Information for capture at run-time is gathered from two sources: the instantiated circuit and the encapsulating Java program via communicating with its `port()` method. These two units that make up an MHP, provide current resource details, structure and port I/O information. Together, the information gathered allows a new entity to be produced, which contains any changes made to it since its initial instantiation to be captured in real-time, while maintaining persistence of initial design semantics.

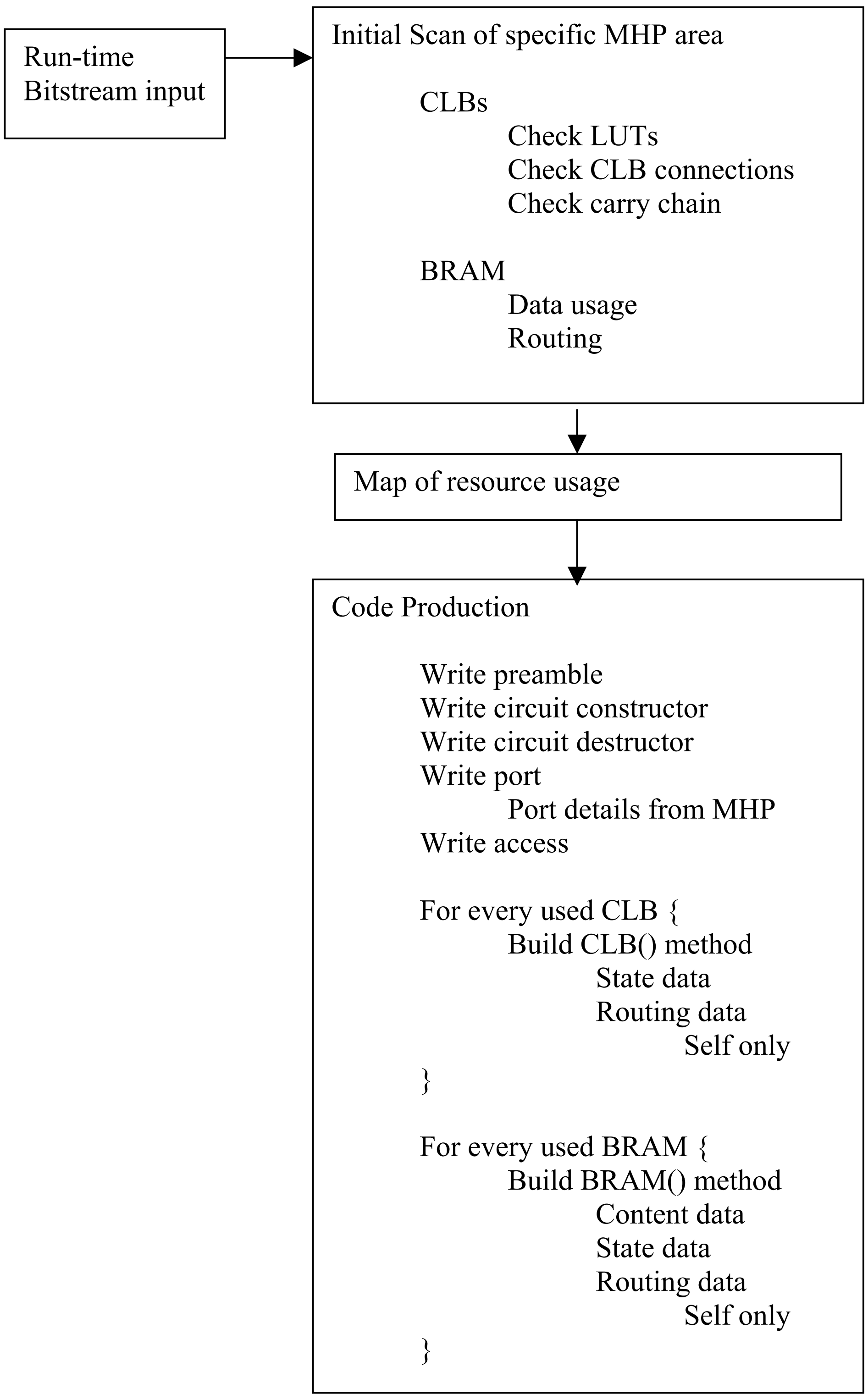


**Figure 31. Recapture system**

The recapture program scans the circuit, capturing resources and tracing routes as before, building up a new Java program with a name given by the user. An entity's circuit, when instantiated in the FPGA, could obviously be linked into other circuitry, such as other MHPs. Blindly tracing routing would lead to all connected circuitry, possibly through its I/O ports, to be gathered into the process. The mechanism used here was to give the recapture circuitry the ability to distinguish between the selected process's resources (self) and another's (non-self). The routing scanner, for example, determines the extent of the process's wiring, by checking whether encountered sinks are possessed by the process itself. In this way, even a fully connected and active process can be captured. A process can be made to enter a frozen state, to allow non-ambiguous capture of the state of its resources (which may be changing while the capture is underway). Such a state can easily be communicated to the run-time system.

This generated program is then dynamically compiled and its separate files organised into the appropriate directories: `.java` output to the `design` directory and .class output to the `pool` directory. This all occurs at run-time, automatically, without the user intervening in any way. The new MHP can then be used in the same user program, if so desired.

As well as the structural description generated as described above, a state map is produced which is a separate program for reproducing the flip-flop states. This program can then be used, via the appropriate commands, to enter a specific state at some future point.

## 4.2.7 Design Engine/Supervisor

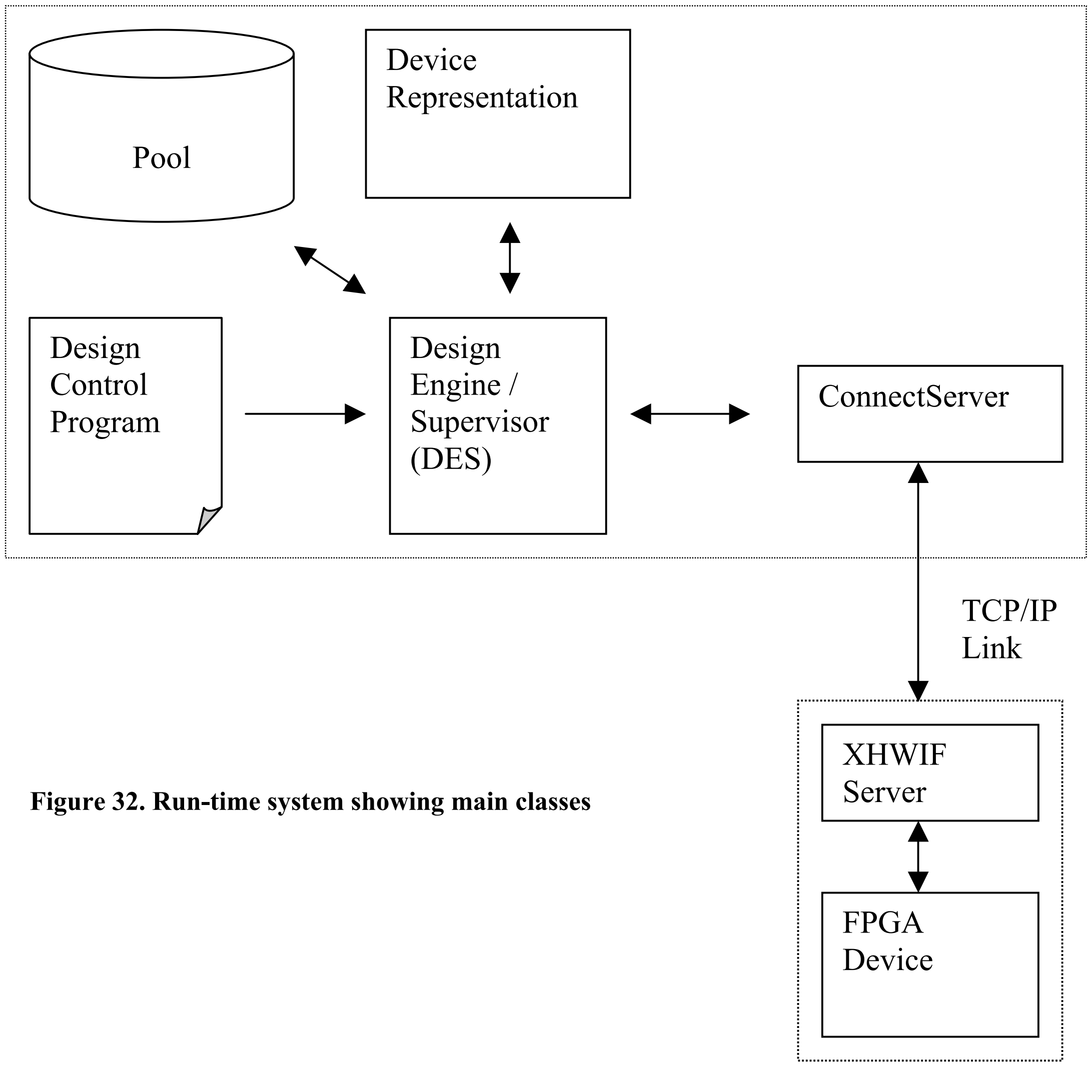


**Figure 32. Run-time system showing main classes**

DES is the main class operating at run-time. It takes a supplied DCP, parses it, interpreting and executing the various commands in the script. The parser is a major part of DES using traditional interpretive methods [1, 2, 31]. The commands are described in the next chapter, as well as in the reference guide in the appendix.

There are several areas that the control language deals with:

*Initialising device and internal representation model.* This includes commands which configure the system to a connected device (which may be local or remote), so the model, for example, conforms to the correct type.

*Location or port definitions.* Locations can be assigned a name, which can be used for referencing within the program. Some locations, such as a process's ports, have already defined names associated with them.

*Device control.* This includes such commands as; resetting the device, clock stepping and forcing a "readback".

*Process control.* A number of commands are available for the placing and routing of processes. They may be automatically placed, or organised within a user specified pattern. Various ports (or locations) can be defined on a process, then observed or linked to I/O of other processes, or external device pads. Processes can be unloaded and the resources associated with them can be freed for further use. A process's I/O can be controlled and switched on and off for isolation. A process may also be "re-captured", for storage and transport to another location.

*Interconnect.* This set of commands allows defined locations to be routed or unrouted. Locations can be device pads, IOBs or process I/O.

*Interaction.* Using defined locations, it is possible to collect data from active processes, which can then be acted on.

*Program sequence and control.* A range of repetition and control structures exist.

*Diagnostics.* Although most low-level activity is shielded from the user, it is still possible to use diagnostic functions, if required, to check circuits. A good example of this is the "tree" command, which enables a user to view a hierarchical tracing of connections from a defined point.

When DES runs a user supplied program, it initialises a device model, which is a representation of the external physical device. The processes, initially in the form of the Java classes, are loaded into memory (via a Java class loader) from the archive `pool`. Processes can then be directed to instantiate onto the FPGA, route to I/O and interact.

This is visualised in Figure 33. The readback and implementation loop, which in reality takes place over a local or network TCP/ IP connection, occurs only when necessary, such as an event instigated on either side. Any exchange of data is limited to packets, rather than whole bitstreams, to enable fast update or information retrieval.

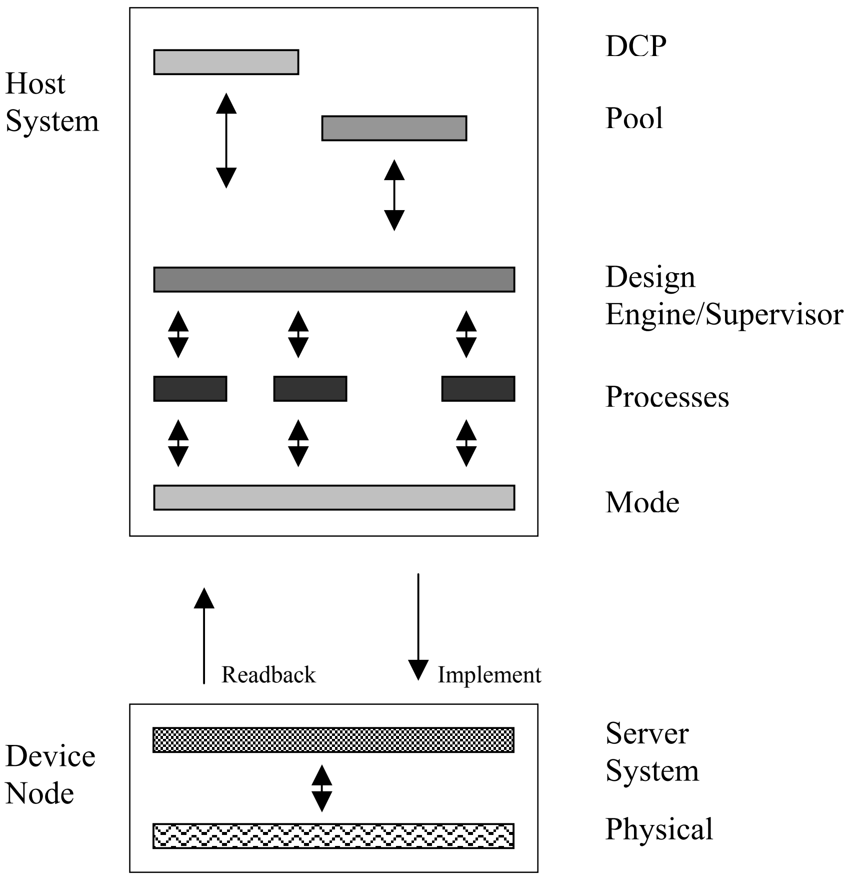


**Figure 33. Run-time visualisation**

When a process is first initialised in a DCP, a `process` object is created that contains the specific name of an MHP and its class type. The user may then request an instantiation on the FPGA of the MHP and DES will then create an `fpgaObject`, which is the generated program (class) from the capture system. A `loadObject` is also created; this contains a reference to the `fpgaObject` but also can contain more detail which is specific to that run-time, such as where its circuit's origin is on the device and port details. A reference also exists to the `loadObject` from the `process`.

A `var` is another object which is essential to DES. This contains details of a defined register, or set of registers on the device, which can act as ports. The object contains its user-defined name, the process's name it is part of, the bus details, which it connects to and other such details. It also contains methods which allow its value to be read from the FPGA, or set to the FPGA. Another method also converts any returned values to its appropriate type and form, usually integer.

All this stored information and the links which bind the objects together, allow DES to access information cleanly and efficiently at varying levels of abstraction, depending on what is being asked of it by the user program script.

The code for recapture, within the DES class, is a good example of this:

```
command = parseTool.getNextToken();  // get the old name
process proc=findProcess(command);
loadObject lObject = proc.lObject;
fpgaObject fpgaObject = lObject.fObject;
command = parseTool.getNextToken();  // get the new name

jRecap capt = new jRecap(JbitsCanvas, fpgaObject, command);

capt.jCapture(lObject.row, lObject.col,
fpgaObject.getRowSize(),      fpgaObject.getColSize());
runDosProcess dproc = new runDosProcess();
dproc.exec(new String[] {"javac", command+".java"});
dproc.exec(new String[] {"move",command+".class","pool"});
dproc.exec(new String[]{"move",command+".java","design"});
```

Once the command recapture has been extracted from the script, the above code is executed. The syntax for recapture in DCP script is fairly simple:

```
recapture(oldprocessName, newprocessName);
```

The first task is to locate which process requires recapturing. Once this has been ascertained, it is an easy matter to locate the relevant details, recapture the process, compile the Java dynamically and add the class to the pool for immediate re-use.

One of the main tasks of DES is the loading and placing of processes. Within the DCP language there are two ways of doing this which relate to modern programming techniques, utilised in programming frameworks such as .NET by Microsoft.

DES should be preferably used with DCP as a managed programming language, that is, resource allocation is looked after by the run-time system itself, rather than the user. How this translates to FPGA programming is similar in that resources at low-levels are looked after, in the sense of their allocation and release. This also relates to where such resources are allocated on a devices matrix. DCP allows a user to simply say:

```
load(pulsecounter);
```

rather than having to state explicitly where the process is to be located and what resources it should use. The explicit version, `loadAt`, is available but generally it is much better practice to use managed capabilities inherent within the system. To allow fast, automatic placement, a boolean matrix representing CLB usage is present in memory to quickly check area usage. Once an area of the correct size is determined as free, then placement can be initiated by calling the `build()` method on the process's Java class.

### 4.2.8 Communications

The update of the internal representation of an external device and all other communications for control, are dealt with by the connectServer class. This class uses the Xilinx software known as XHWIF (Xilinx Hardware Interface), built into JBits, to provide TCP/IP communication, with external and local classes. At the device to be controlled end, a XHWIF server is activated, which passes data and control packets to the connected FPGA hardware. The connectServer at the Reconnetics side can then link to this software, with normal TCP/IP, again developed in Java.

The main methods within connectServer show the functionality that it allows DES.

- `getConnection()` – connects to a hardware device.
- `reset()` – sends a reset command to the connected device.
- `update()` – sends any changes from the internal representation to the hardware.

- `step()` – when the device is in step mode, will allow a specific number of steps to be executed.
- `clock()` – switches between clock mode and step mode.
- `frequency()` – sets the clock frequency on the hardware.
- `disconnect()` – disconnects from the hardware.
- `readback()` – forces a readback from the hardware, returning an up to date bitstream from the device.

After parsing the DCP user script and executing commands, such as `link()`, an update command is issued to bring the FPGA configuration in line with the internal model. Viewing a defined variable, which is in effect a register on the hardware, forces a readback, so the value held on this variable is updated. Updates can also be forced in the DCP, using the `readback()` command, which also brings the internal representation up-to-date with any changes that have occurred. Transfer of such data is in the form of packets, which only contain changed information, reducing the size of any transmission. These packets are targeted at specific areas within the bitstream, which relate to areas that have changed since the last update. A process on the hardware device can also force an update by initiating an interrupt. It is therefore possible to initiate updates with the usual methods, such as polling, or interrupt driven.

## 4.3 Mobile Hardware Processes and Design Considerations

A process is extracted from a configuration bitstream by analysing resource usage and I/O requirements. Nets which lead to I/O points are viewed as the process-to-be's main interface points to the outside world. If such points are named in an accompanying user constraints file (UCF), in the usual format, then the name persists through from the design/programming stage to run-time.

### 4.3.1 Resource Mappings and layout

The system works best with designs that have a roughly rectangular layout. This is because when used CLB resources are found, a 2D sub-matrix is built from the layout shape of the design. Rectangles allow space to be used efficiently and quickly, rather than complex transformations having to occur at run-time. If the design is not particularly rectangular in shape, or is sparse in its layout (few CLBs laid out erratically over a large area), then there would be wastage in terms of resources.

It is possible to assert constraints to synthesis tools to make layouts compact and rectangular in shape, although tools have been developed in this system to make sure designs can be utilised properly. Precapture, for example, allows designs to be compacted and transformed prior to run-time use, with a rectangular shape.

A new technique is also being developed that allows such 2D objects to overlap and resources to be shared within their respective areas. This inter-mesh works at a finer level of granulation, to allow resources to be individually possessed.

### 4.3.2 Block RAM (BRAM)

On device block RAMs are supported and captured by Reconnetics. When a process is instantiated that uses BRAMs, certain strategies are used for their handling and placement. The location to place the BRAM is taken primarily from the desired loading point of the process. Other BRAMs are located sequentially on the side nearest the process. This resource allocation can fail if not enough BRAMs are available.

Another way of using BRAMs is through the special `bramobj` object. This is used in a similar way to processes, the main difference being that the coordinates you give for the location in a `loadAt()` statement refer to the BRAM row and column. Automatic loading and placing is also available.

There are a few ways of changing BRAM contents. To initialise BRAM contents to zero, the `resetBRAM()` command can be used. To set specific values, two methods can be used. The first is `setBRAM()`; mainly for inserting a small number of values. The second is to use `readHex()`, which will allow an Intel Hex format file to be read into a BRAM. Most assemblers can output this particular format.

### 4.3.3 Bi-Directional Wires

Bi-directional lines are nearly always (dependent on the device in question) split into two uni-directional lines, avoiding contention. The Reconnetics system analyses networks and decides how such lines should be handled. Usually, such nets are handled by adding `_IN` and `_OUT` to the wire name to identify the separate nets.

### 4.3.4 Tri-State Wires

At the present time nets which lead to tri-state input controls are locked to the destination point by default. To utilise, simply keep the I/O lines at their default location as well. Of course, some HDLs do not use tri-state wires at all.

### 4.3.5 Mobility

In the case of MHPs, mobility can be thought of in at least two ways; internal to the device and external, over networks. The idea was to create a vehicle in which the process's physical design can be transported and re-created wherever. The method that was chosen, a Java class, allows the design to be held in a host computer, transmitted and then instantiated in an FPGA, which may actually be different than the original compilation intended.

### 4.3.6 Re-Use

While processes can easily be used on devices they were originally compiled for, there is also a degree of device independence. Re-use capability is determined by three main parameters – the actual device type, clock speed and clock routing. Using this system the device type (currently within the Xilinx Virtex series) can be changed quite easily. A design originally intended for a small device (such as XCV50) can easily be transferred to a much larger device and vice versa, resources allowing. The clock speed can be maintained at its old frequency. The clock routing for a specific canvas can be changed, there are tools which do this; particular care needs to be taken when adjusting clock nets attached to BRAMs.

### 4.3.7 Interaction

Interaction is possible between the processes themselves and between the processes and the system. Inter-process communication is, of course, wire based. Such wire connections can be controlled within the user written DCP.

The system itself can observe and interact with the instantiated design by the user defining locations, usually registers, in the circuit, with a given name. These usually are clocked registers, which may be, for example, buffered I/O points. As well as collecting data from such points, it is also possible to set values at inputs.

### 4.3.8 Usage outside of Reconnetics

MHPs are capable of being used outside of Reconnetics, as long as the JBits and the Reconnetics classes are in the class path of the Java compiler and the interface for the MHP is followed and understood. The methods for a standard MHP (an `fpgaObject`) have been elaborated as; `build()`, `destroy()`, `port()`, with numerous `CLB()` and `BRAM()` methods. A call to the `build()` method with the appropriate parameters will result in the circuit being instantiated, for example:

```
fpgaObject pulsecounter = new fpgaObject();
boolean placed_ok = pulsecounter.build(JbitsCanvas, row, col);
```

This will deposit the circuit in a bitstream (`JbitsCanvas`) at row, col for download to a device. Using MHPs in this way does have some advantages for the more knowledgeable user – it is certainly more high-level than trying to do the same thing in pure JBits code, although lacks many of the managed and secure features of the actual run-time system. Reconnetics allows placement to be very simple and safe, with many supporting features that would probably take a long time and therefore be inefficient to develop for a "one-off" system.

Further details of class signatures, methods and usage of Reconnetics classes are available in Appendix III.

This section has presented the Reconnetics dynamic run-time system and its mobile components, from the system's programmer perspective. A system overview was presented and the main aspects of the implementation explored. Detail was given of how MHP programs are generated and their constituent methods.

The next chapter will examine how the system is utilised from a user point of view, giving examples and exploring possible scenarios for deployment.

# 5. User-Orientated Perspective

## 5.1 Introduction

The following section takes a user perspective and shows how the managed run-time system, Reconnetics, can be utilised to manipulate the mobile hardware processes with the DCP language. It also offers scenarios in which the system can be deployed. Several examples depicting usage are discussed.

### 5.1.1 Scenarios for using Reconnetics

There are several ways that the system can be used:-

Case 1: Static Place and Route

At the most simplistic level it could be used to build designs controlling place and routing to a specific pattern. This may include no dynamic interaction. The bitstream produced could then be used, as usual, to configure devices, not necessarily using any advanced features of the system itself.

Case 2: Distributed Design

Designs within this scenario are manipulated either locally and, if required, at some distance. Updates to hardware can then be distributed over networks.

Case 3: Dynamic Design

This envisages a design cycle extending into run-time. Instantiated processes can be interacted with, verified and data downloaded from. The device can be amended as required, or on specific events taking place. A device may also *request* certain processes to be instantiated, for example, on a fault condition arising.

### 5.1.2 Mobile Hardware Processes

The Reconnetics system uses a model of design incorporating objects defined as Mobile Hardware Processes. Processes are designs which are instantiated as self-contained hardware units, with their own well defined I/O to the rest of the world.

Although a process could be a whole complex design, it is better to break down a larger entity into component parts. Each part can then be manipulated as a separate component, interacted with and altered as necessary. Using this system, processes become mobile and easily transferred over networks.

## 5.2 Engineering the Processes

### 5.2.1 Designing Processes

There is little difference between normal engineering practice and the designing of components for this system. Each unit undergoes the usual cycle of development, testing and implementation. This may be using VHDL, Verilog, schematic tools, or behavioural languages, such as C. Existing designs can usually be instantiated as processes with minimal change.

A design to be captured by Reconnetics is developed so I/O points to other processes or devices can be easily recognised. This is done by directing synthesis to such points, as if connecting to external pad positions. In appendix II there are designs which are developed in such a way. Note how the input and output lines within the entity port definition are directed to a pad position; it doesn't matter so much *which* pad – it is merely saying that this is a unique I/O connection to the outside world. Any names defined for these input and output lines persist for use within any control programs in later use. The UCF file can be usually written manually or generated automatically.

During synthesis and implementation a selected chip is targeted, for example, XCV300 or XCV1000. If it is only a small design, then choose the XCV300, as this marginally speeds up capture time. Choosing a specific chip at this stage has no effect later when placing, as the generated process has device independence.

A bitstream is then generated with this design. By feeding the bitstream itself (and a UCF file if you required automatic named points) to the capture program, a process is generated for placement in the design pool of the Reconnetics system. The capture application also analyses and checks the design for problems and corrects them where necessary. It can, for example, detect disconnected wires or corrupted circuitry and using design rules "rebuild" the design. If you require I/O buffering (described later), use the preCapture. If not, use the grabObject program.

Other constraints can also be added, such as area and timing, using the UCF file as you would normally. It is advantageous to constrain the area so a process occupies less space.

A process is captured and occupies a rectangular matrix which is overlaid (placed) on a device's gate space. Coordinate positions for placing that process are taken from the bottom left corner, although placement positioning can, in most cases, be left to the system itself.

### 5.2.2 PROCIO connect points

Connect points can be added to designs which simplify and improve I/O handling. This is done by utilising the preCapture program which scans the design file and places special CLB logic at the I/O lines. This extra logic acts as a buffer or connection point between placed processes. The idea here is to:

- Decouple processes
- Act as a collection point for internal nets (helps with keeping compact)
- Provide a controllable gateway to the outside world

The actual unit which is used to do this is a “system object” within the design pool called `procio`, with the same signature and methods as any other MHP. Only a quarter of a CLB is used for each I/O wire (or net) required.

When `procio` is used, the I/O points can be connected to either clocked or unclocked outputs. For connection to the clocked output point of a buffer, add `_clk` to the designated name.

## 5.3 System Usage

### 5.3.1 Description of the Development Cycle

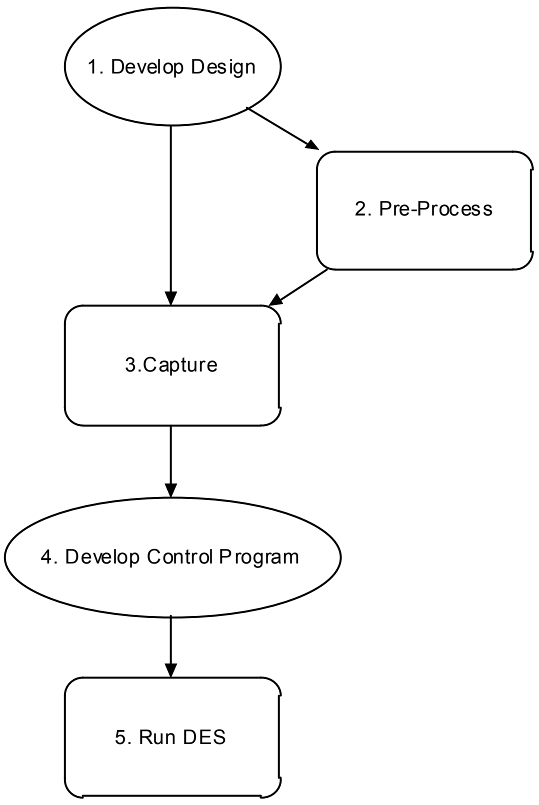


**Figure 34. Diagram showing design cycle**

**Develop Design**

The design is first developed using a HDL or schematic tool. This is then synthesised and implemented to a bitstream.

**Pre-Processing**

This optional stage allows the submitted design's bitstream to be processed so buffers can be inserted at I/O points, as well as some design compacting. This is recommended to allow various features of the system to be fully utilised in control programs. However, in some designs output points are already clocked and the design itself compact, so such processing becomes unnecessary.

To pre-process:-

```
java reconnetics.preCapture designName
```

The product of this is a file named `temp.bit` which may then be input to the next stage.

**Capture**

The bitstream is then used to generate a Java file. A UCF file can also be submitted to allow naming of I/O ports. If this is not found then the naming can be done manually.

To start the capture:

```
java reconnetics.grabObject designName
```

If the previous pre-process stage was used, then the actual name would be `temp.bit`. A batch file, as described below, can be used to simplify matters.

The `designName.bit` file is then scanned for resource usage. This includes CLB, IOB, BRAM and routing information. A program is produced which can replicate this circuit at any given position in an FPGA matrix. As well as CLB and routing methods, a `port()` method is included which allows control and information exchange about the I/O on that process.

After scanning the bitstream and reviewing resource allocation, the next procedure for the capture system is the naming of I/O ports. If a UCF file is present, then information for ports is extracted from that, alternatively the labels can be added manually.

The output Java file is then compiled as usual and placed in the design pool for use. More designs can be added as needed.

**Develop Design Control Program (DCP)**

A program is written to control loading and linking of processes, I/O etc.

**Run Design Engine/Supervisor**

The program developed at the previous stage is run by the supervisor –

```
java runReconnetics programName
```

## 5.4 Design Control Programming Tutorial

### 5.4.1 Developing a Control Program

**Initialising**

A typical start-up sequence can be:

```
message("Program starts");
device(xcv300, null300GCLK1, localhost);
reset();
```

In this example a message is first sent to the screen to alert the user that the program is about to begin. A device is then chosen, followed by the canvas that is to be used. In this case the device is going to be a XCV300, so a null XCV300 configuration bitstream is going to be used which has a clock set up of GCLK1. The canvas can also be used to begin with an already configured design, which is then added to, manipulated, or interacted with in some way. If the device is local, as it is here, `localhost` is used as the connection name. The board may also require resetting to a default state. It is also possible to continue a previous design already set up on the board. To do this, no reset would be used and a readback would be performed first to bring DES into line with the configuration.

**Definitions**

Definitions can occur anywhere in the program. Two types of definitions are allowed. These are for processes and locations. To use a process a name has to be given and a class type. The class type is the name of an archived process within the `pool`. An example could be:

```
process(pulse_counter, pulsecnt);
process(pulse_generator, count4);
```

The first parameter is the name of the process and the second is the class type. Here two processes are being defined; `pulse_counter` and `pulse_generator`. It is important to note here that they have not been placed on the board at this point and no resources are being used; merely that there is a definition of a name with a type.

Locations (or ports) are also defined. These take the form:

```
var(pulse_counter_out, pulse_counter, count, 4);
```

The first parameter is the name associated with that defined location. The second is the process which owns that location. Finally, there is the name of the port and the number of wires. It is possible to make multiple definitions of the same port and use different data widths for each. The only restriction being the actual size of the port itself. In this case the location defined is an output port belonging to `pulse_counter` by the name of `count` with a data width of 4. Any future reference to this location can now be made through the defined name `pulse_counter_out`. It may also be useful to define several other locations:

```
var(count0out, pulse_counter, count[0], 1);
var(bit0out, pulse_generator, count[0], 1);
var(pulsecounter_in,pulse_counter, data_in,1);
```

The starting point of the definition can be any bit in the port using an index notation. The above definition, for example, allows the first bit of the output port to be accessed individually by a separate definition. Two other locations are defined; `bit0out` is an output from the pulse generator, while input is a data input point on the other process.

Note here how wires are indexed via a subscript, with square brackets.

**Loading and Unloading**

There are two ways to load defined processes onto a device matrix. The first command is `load()`, which allows the system to organise the matrix.

```
load(pulse_counter);
```

The second is the manual command `loadAt()`, which as well as the name of the process, takes an additional two parameters for the row and column. These place the lower left bottom corner of the process at that position on the devices gate space.

```
loadAt(pulse_counter, 10, 20);
```

It is possible to build up a design with `loadAt()`, then switch to automatic placement with load. It is not advisable to switch between the two at various points in the program as the automatic placement may have instantiated a design where you believe an area is free.

The unload command simply requires the name of the process to extract, the resources are then free and released for re-use. Unloading deletes all link output wires which belong to that process. Any wires that act as input from external processes are the responsibility of the owning process.

```
unload(pulse_counter);
```

**Linking**

There are quite a few linking commands, for different purposes, although the main one is `link()`.

```
link(bit0out, pulsecounter_in);
```

This command links two defined locations, from output to input. In this case it links a single output wire to an input port, again of one wire data width. The two locations must be the same data width.

Other link commands allow default locations, pads or IOBs to be linked.

The opposite of linking is unlinking, using the `unlink()` command. This takes as its only parameter the name of an output point which is part of a connection. The connection could be to a process, an IOB or a BRAM.

**Executing**

Several commands influence the actual execution of processes. Prior to loading processes it may be advantageous to turn the device's clock off, using the `clock(off)` command. When loading is finished the entire board can then be activated so all processes are, in theory, in sync. Other considerations include the setting up of the clock frequency and deciding which clock to use. Some designs may already have a defined clock. This may run at odds with what is actually in use on the device matrix. Using the `gclk()` command, all loaded processes can be set to one clock.

By turning the clock off and using the `step()` command, it is possible to observe states on the board. This is useful both for educational, as well as diagnostic, purposes.

**Observing**

Defined output points can be watched, using the `print()` command:

```
print(pulse_counter);
```

Messages can also be output to the screen using `message()`.

**Diagnostics**

A location's wire link can be seen by using the `tree()` command. This prints out a low-level hierarchical list of connections.

Another useful command is `write()`. This allows the current configuration bitstream to be written to a file for further analysis by other tools, or for archiving.

**Conditions**

The main conditional control is provided by the `if...endif` statement. It takes the form of:-

```
if (pulse_counter < 4)  {

     message("fewer than 4 pulses...");

} endif;
```

The block being executed up to `endif`, if the condition is true.

**Repetition**

The `repeat` statement is a simple command for cycling a certain number of times:

```
repeat(4) {

     message("The pulse counter is:");
     print(pulse_counter);

} endrep;
```

Another loop mechanism is the `do ... while()` construct. This takes the form:

```
do {
     message(“checking...”);
     print(pulse_counter);

} while(pulse_count<=100);
```

**Interaction**

As well as observing output from processes, it also possible to feed input. This is done using the `inputVal()` command:

```
inputVal(pulsecounter_in, 1);
```

The value submitted must be appropriate for the data width of the input.

Processes may be effectively isolated from the world by turning off their connection points. This is done utilising the `inputON()` and `inputOFF()` commands:

```
inputOFF(pulsecounter_in);

// do something here

inputON(pulsecounter_in);
```

**Recapture**

A process can be recaptured after it has been placed on the device’s matrix. This can occur even after the process has been linked to other processes and pad locations. The process is scanned and information extracted to rebuild a new Java program, encapsulating its design. All named port details are replicated in this new process. Any structural and data differences which have occurred since placement are captured. The newly generated program is then dynamically compiled and placed in the pool, for immediate reuse.

```
recapture(pulsecounter, old_pulsecounter);


reload(old_pulsecounter);
```

When a process is recaptured it generates a structural map and a state map. A state map is a map of the values contained in flip-flops within the area possessed by the process. This map can also be used separately to overlay on an initial process class of the same type from the pool. In the above example this would be a `pulsecounter`. This could be used to initialise a process with specific values that you know are stored in flip-flops. The `stateMapAt()` command is used for this:

```
// set up an initial process
loadAt(pulsecounter, 10, 10);

// load in flip-flop states
stateMapAt(old_pulsecounter_state, 10, 10);
```

The suffix of `_state` is added to the name of the original recaptured process to access the state data alone.

### 5.4.2 Whole Process Manipulation

Processes can be placed (using `load()`), interacted with and destroyed (using `unload()`). If a design as a whole is to function properly during its manipulation (in particular, its destruction), then certain methods need to be used.

A problem could occur, for example, when some communication may be taking place between processes and the system wants to extract one of them. This situation necessitates the use of communication signals to determine the state of a process at a given time. A protocol in this way could be used to interact with the run-time system to make its safe extraction possible. This is also true during the re-capture of a process, when it must first enter a “safe” state that must be signalled via defined points.

Under normal conditions it may be possible to simply shut down the whole device while such an event occurs. One possible way to do this is by turning the clock (or clocks) off. Another way to do this is to shut the clocks down to only the affected processes. Specific commands exist in Reconnetics for this purpose [20].

The least interfering and non-invasive way to achieve safe extraction is to isolate the process. This involves switching off any inputs to the process and possibly generating a stand-by signal to the connected units. Input isolation is achieved through the `inputOFF()` command. It would be a simple matter to provide the correct hand-shaking protocol and routing for a stand-by signal in processes, if this is required.

A process owns its output routing but not any wires connected to external inputs. When the process is unloaded, its internal logic and routing are released, as well as any output links that it owns. Routing to a process's input port persists after its destruction and must be unrouted (if required) by the owning unit.

### 5.4.3 Using Extrinsic Mobility

It has been shown how MHPs can be easily mobile within a device using such statements as `load()` and `loadAt()`. It is also possible to consider manipulating a process extrinsically, that is, so it can be transferred between hardware nodes in a network. There are a few ways of achieving this, depending on what the aim of such a venture is and the device set-up at each node.

The minimum set-up using Reconnetics is a network of devices which are capable of running the XHWIF software at the client nodes and the run-time software at server nodes. Simple dedicated units are available which are internet linked and run this XHWIF software.

A network like this allows the processes to be mobile. For example, at its most simplistic, Reconnetics can connect to a node, deposit a set of processes and move to the next node. In such a way a group of distributed equipment can be updated, for example, with the latest release of hardware.

[20] All commands are discussed in the next chapter.

It is also possible for a process to request a transfer to another node by catching such a message while a DCP script is running. The message can be as simple as a change on one bit of a defined variable, data in BRAM, or value on a register, which could, for example, detail which node it wishes transfer to. Point to point transfer can be done using the system by disconnecting from one node and connecting to another, or by sending the process to another Reconnetics server. Recapture can be used to capture the process in its current state prior to transfer, then it may be unloaded and finally transferred, where it is rebuilt at the new node.

```
// simple move node program

message("Program move node…");

device(xcv1000, null1000GCLK3, 62.64.239.34);

reset();

clock(off);

process(smallcpu, cpu6502);
load(smallcpu);

// define location, only one bit for this signal
var(movereg, smallcpu, regOne[0], 1);

clock(on);

do {

      // just wait!

} while (movereg!=1);

// could be recaptured if required
recapture(smallcpu, nextstatecpu);

unload(smallcpu);
device(xcv1000, null1000GCK3, 62.64.232.12);


process(new_smallcpu, nextstatecpu);
load(new_smallcpu);
```

The large amount of flexibility built into the run-time system allows many methods for mobility of processes to be possible. The main method is mentioned above, of recapture, or simply sending an archived process to a new node. A whole bitstream could also be transferred, if required, or a bitstream written to file, modified and then transferred. A node can also be connected to with some unknown hardware loaded, which can then be augmented. In this case, free areas can be used for processes, or if interface points are known (possibly just I/O units or pads), then these can be connected to the new processes.

Useful features which are under development are the `saveMap()` and `loadMap()` statements, that will allow high-level details about nodes to be rebuilt if a connection has to be broken and picked up at a later time. The map will store information on where processes are placed and program specific details, such as variable definitions. This information could be stored at the client node, or more likely, the server.

### 5.4.4 Design Control Program Example

```
message("Program starts");
device(xcv300, null300GCLK1, localhost);
reset();
clock(off);
message("Loading");
process(pulse_counter, pulsecnt);
load(pulse_counter);
process(pulse_generator, count4);
load(pulse_generator);

var(counter_out, pulse_counter, count, 4);
var(count0out, pulse_counter, count[0],1);
var(count3out, pulse_counter, count[3],1);
var(bit0out, pulse_generator, count[0], 1);
var(input,pulse_counter,data_in,1);

message("Linking");
link(bit0out, input);
linkPad(count3out, P56);

message("entering loop");

do {


      step(1);
      print(counter_out);
} while (counter_out<=4);
```

```
unload(pulse_counter);
unload(pulse_generator);
```

Much of the code for this complete program was described in the preceding section. It demonstrates several important areas: initialising a device, defining and loading processes, defining locations, linking and interacting with processes within a loop. Finally, the processes are unloaded. The program will now be described in greater detail.

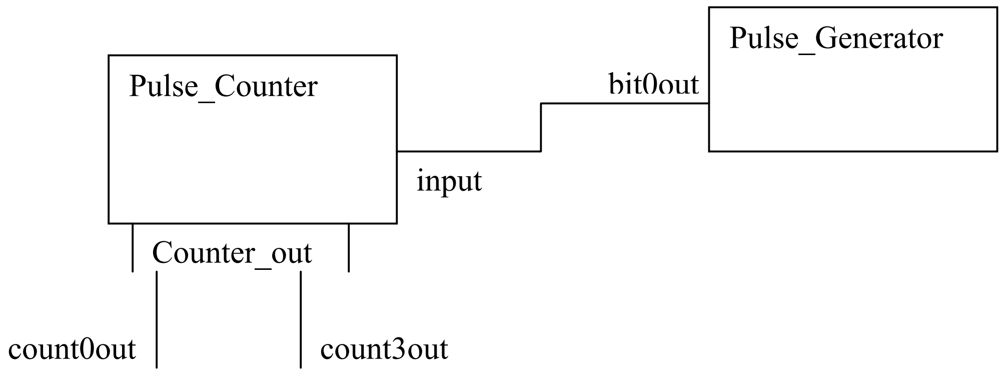


**Figure 35. Example visualisation**

This simple program uses two processes; a pulse generator and a pulse counter[21]. The initialising section sets up the appropriate connection to the device and selects the correct canvas to work upon. The device is also reset to its default state. Note here that the clock is set to `off` so it is possible to step the device and view the consequences at each pulse.

The processes are then defined and loaded using automatic placing and routing organisation.

The next section defines locations for both viewing and linking. As we can see from the visualisation, as well as defining an output port on the counter, two other bits have been defined on the same port, as an example.

[21] Detailed in Appendix II as VHDL and Handel-C code.

Following this are two links, one made between the two processes and one to a pad location on an IOB.

A loop is then entered which steps the device, prints the value at the counter output and checks to see if there is an exit condition.

The last lines unload the processes from the device, this may or may not be required at the end of a program’s run time. You could, for example, leave the configuration active and link to it at a later point, with either this system or another tool. It is also possible that you may just want to configure and disconnect rather than perform any kind of interaction. Yet another scenario would be the verifying of correct function and some kind of update, if a valid state is not found.

It is also possible to load several processes and then as they are finished, reclaim resources with `unload()`. The resources are then open for new assignment and organisation. Link resources are also reclaimed in this manner with the `unlink()` command.

If the device you wanted to connect to was not a XCV300, then the device command would need changing, as well as the canvas type. The canvas is merely a null bitstream with the appropriate global clock settings. If a design has inherited any clock settings already built in, these can be set to the appropriate clock, once loaded, using the `gclk()` command. No other alterations should need to be made, so designs have a certain degree of device independence, once captured. By way of example, to alter the above to run on a XCV1000 based RC1000P board, with a default set-up of global clock 3, the following changes could be made:

```
device(xcv1000, null1000GCLK3, localhost);
```

### 5.4.5 Other Features

As has already been mentioned in the previous section, there are several features within the system that can also be incorporated into designs. BRAMs can be used in a design and will be captured. Any data contained within the unit will, along with routing and resources, be caught. The BRAMs, because there are fewer of them than usual resources, are handled in a slightly different way to other resources. When the process is placed that uses a BRAM, the actual placement is located in the nearest space. If such an area is in use, then the next free BRAM is used. If all BRAMs are in use than the placement of the process will fail.

Users of the system can also utilise a process that is made of a singular BRAM, or a system object known as `bramobj`. A BRAM containing process is simply a captured design containing only a block ram which can be placed more flexibly. This utilises a special system object known as `bramobj`, which allows a position to be selected for placement in terms of its BRAM row and column. Once BRAMs are placed, they can be loaded with data, or if not already part of a process, they can be linked to using normal channel link commands.

```
// initialise device
message("Program starts");

device(xcv300, null300GCLK1, localhost);

reset();

// get some processes
process(bramproc, bramtest);
process(counter, count9):
process(abram, bramobj);

// load them up
message("loading...");

load(bramproc);
load(counter);
load(abram);

message("ok");

// define some locations and buses
message("defining....");

var(addr_out, counter, count, 9);
var(addr_in, abram, ADDRA, 9);
var(data_out, counter, count, 8);
var(data_in, abram, DIA, 8);

message("ok");

// link the BRAM up
message("linking:");
link(addr_out, addr_in);
link(data_out, data_in);
message("ok");

// step it 10x
```

```
step(10);

// unload all processes and bram config
unload(bramproc);
unload(counter);
unload(abram);
```

In this example, a process was written (`bramproc`) which writes the contents of a counter (`counter`) to a bramobj (`abram`). This example shows the linking and commands required and it is run in step mode for the purposes of experimentation.

Contents of such objects can be manipulated in three ways; by a simple reset (`resetBRAM`), by small amounts of data (`setBRAM`), or using a standard Intel Hex format file (`readHex`). The last option allows output from an assembler to be loaded in.

There are no changes required to designs that have been developed in standard ways, such as HDL and CAD. This allows legacy designs to be simply captured and used. However, designs which use bi-directional and tri-state wires at interface points must have special consideration.

A bi-directional wire (or channel) is split into two uni-directional lines by the capture system, with the addition of `_IN` and `_OUT` on the name label for access. This is actually closer to the physical implementation and can avoid confusion. Tri-state wires are not used by many HDLs, such as Handel-C but if encountered by the capture system, they are locked to the destination point by default. A design is therefore locked to its default position.

### 5.4.6 Mobile Processors: A More Complex Design

The examples so far presented were fairly simple. A more complex design is now examined incorporating a third party component. This is a microcontroller available from Xilinx known as KCPSM[22]. This processor has many useful features, such as extensive I/O capabilities, Harvard architecture and compact design. The design itself

[22] Appendix V: KCPSM Xilinx Processor Details

consists of two macros, which are supplied in EDIF format. The first macro is the processor itself, while the second is an instruction ROM. The ROM is instantiated as a BRAM, with a width of sixteen bits.

The main idea for using the processor was for the user to supply a program to the assembler which outputs an EDIF macro containing the program. This would then be attached to the design and synthesised. This method, however, is unnecessary with the Reconnetics system as programs can simply be downloaded into a device's BRAM whenever required. In this case, the Intel Hex format output file from the assembler is used. This stops lengthy re-synthesis and implementation. The EDIF design and its accompanying programs, such as the assembler, must be downloaded from Xilinx [52]. A simple design is then developed utilising the EDIF macro supplied and supporting circuitry:

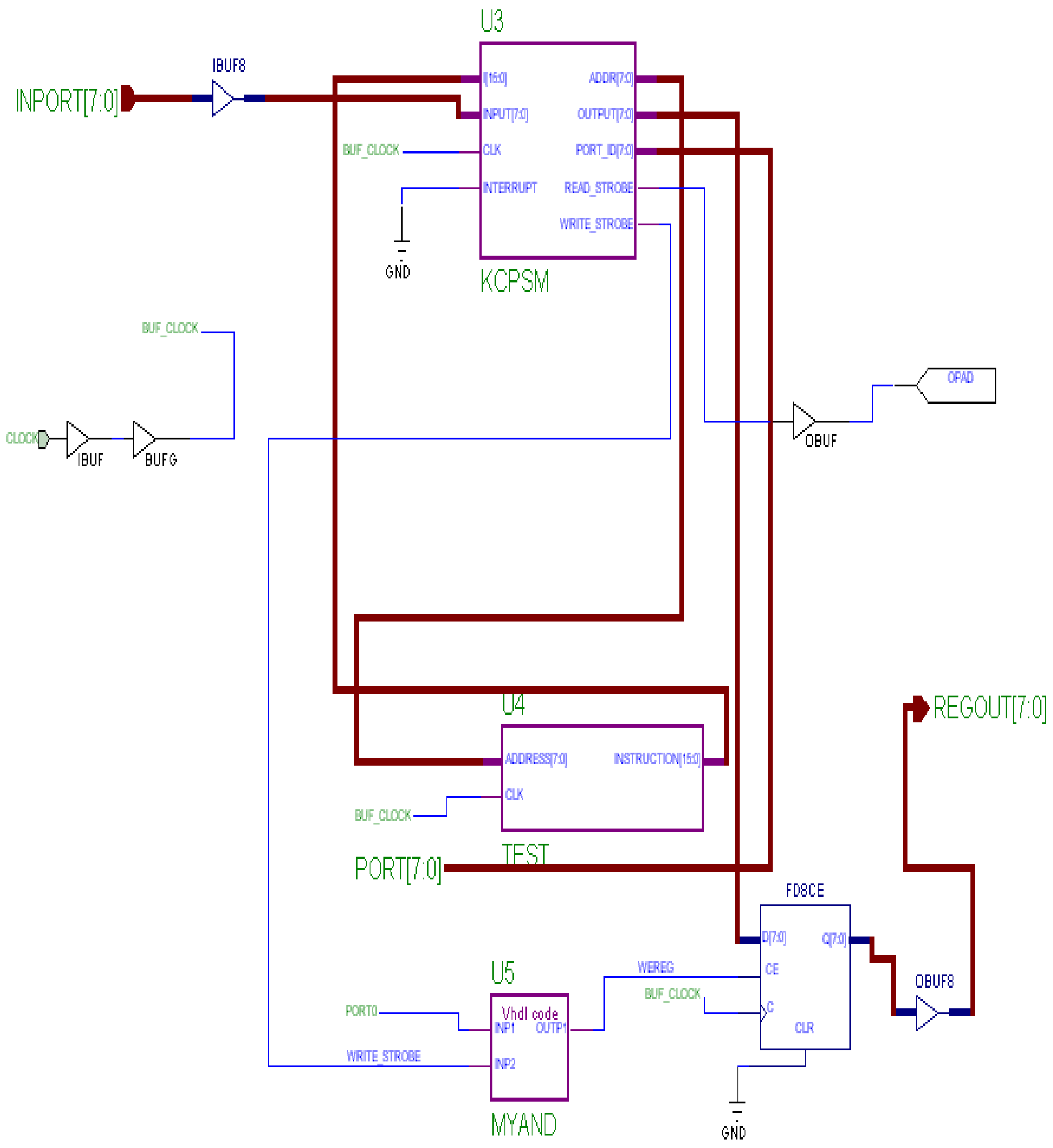


**Figure 36. Schematic for a simple microcontroller test**

This design represents a simple test example, where the output from the register (`FD8CE`) can be checked in real time. As the output from the register is the only thing that is going to be checked, the pins that need naming within the UCF file are the `REGOUT[7:0]` wires:

```
NET REGOUT<0> LOC=P63;
NET REGOUT<1> LOC=P64;
NET REGOUT<2> LOC=P65;
NET REGOUT<3> LOC=P66;
NET REGOUT<4> LOC=P67;
NET REGOUT<5> LOC=P68;
NET REGOUT<6> LOC=P70;
NET REGOUT<7> LOC=P71;
```

The design is then synthesised and implemented to the target device.

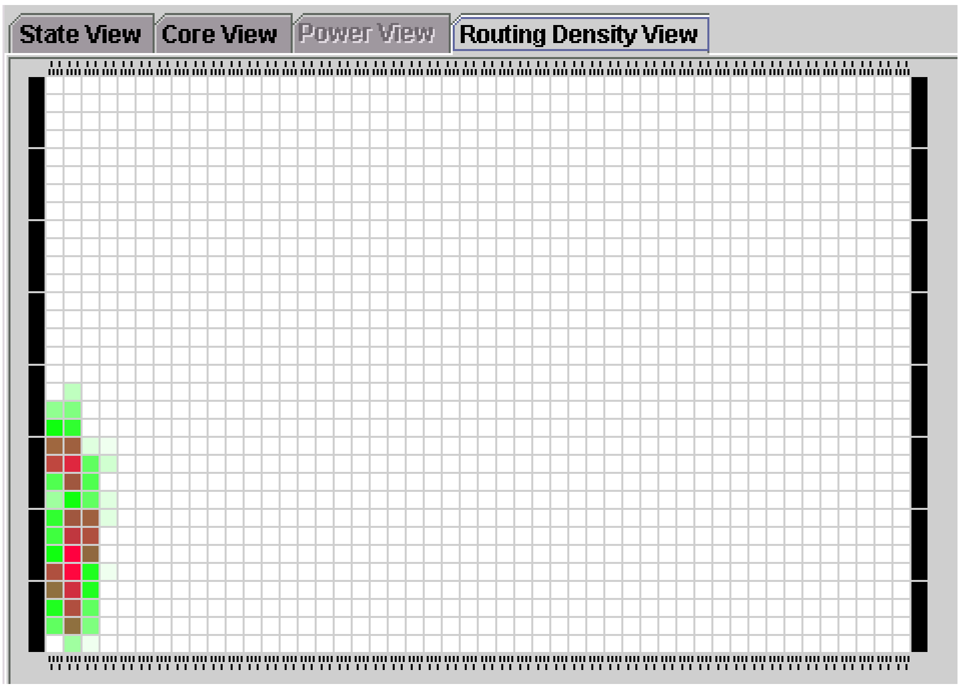


**Figure 37. BoardScope showing implement design**

This shows the implemented design on a XCV300, via the Xilinx BoardScope application. As you can see, it takes up very little space. The colours indicate the relative density of routing. White being little or no routing, through to darker colours, indicating greater use of resources. The black boxes around the left and right sides are the BRAMs, of which the processor uses one.

This output bit file is then captured and placed in the design `pool`. Notice here we do not require the design to be pre-processed. The reason being is that the design is already reasonably compact and the output is already buffered and clocked through a register. A design control program can then be written to control its instantiation and manipulation:

```
// KCPSM processor demo
// initialise device

message("Program starts");

device(xcv300, null300GCLK1, localhost);
reset();

// get the processor
process(processor0, ksm);

// load it
message("loading...");

loadAt(processor0, 1, 0);

message("ok");

// load a very small test program
message("loading program");
setBram(0,0,0,16,0,57344,16385,38145,33024);

// 00           ; test processor#0
// 00           ;
// 00           CONSTANT outp,00
// 00  0000     start:     LOAD s0,00
// 01      loop:
// 01  E000            OUTPUT s0,outp
// 02  4001            ADD s0,01
// 03  9501            JUMP NZ,loop
// 04  8100            JUMP start

var(reg0, processor0, REGOUT, 8);

do {
     step(1);
```

```
        message("reg0:");
        print(reg0);

} while (reg0<100);
```

This program loads a single processor and utilises the BRAM at 0, 0 as a program memory. Wiring stretches between the two but cannot always be seen clearly on the Boardscope application due to the low density of routing. The program to drive the processor in this case is placed via a `setBRAM()` command – it would be usual to use the `readHex()` command to read a longer program in Intel Hex format to the BRAM. The KCPSM processor uses instructions which are 16 bits wide, so any program must be read in:

```
readHex(0, 0, 16, "myprog.hex");
```

Obviously, many processors can be loaded in at a given time, the only real constraint being the resources the design uses. In the case of this processor the requirements are very small indeed; only around 35 CLBs and one BRAM are used! It is an easy matter to fit many processors in a single XCV300 using Reconnetics. Even more can be placed and routed dynamically in larger devices.

Here is the same design after being instantiated 5 times with a Reconnetics program:

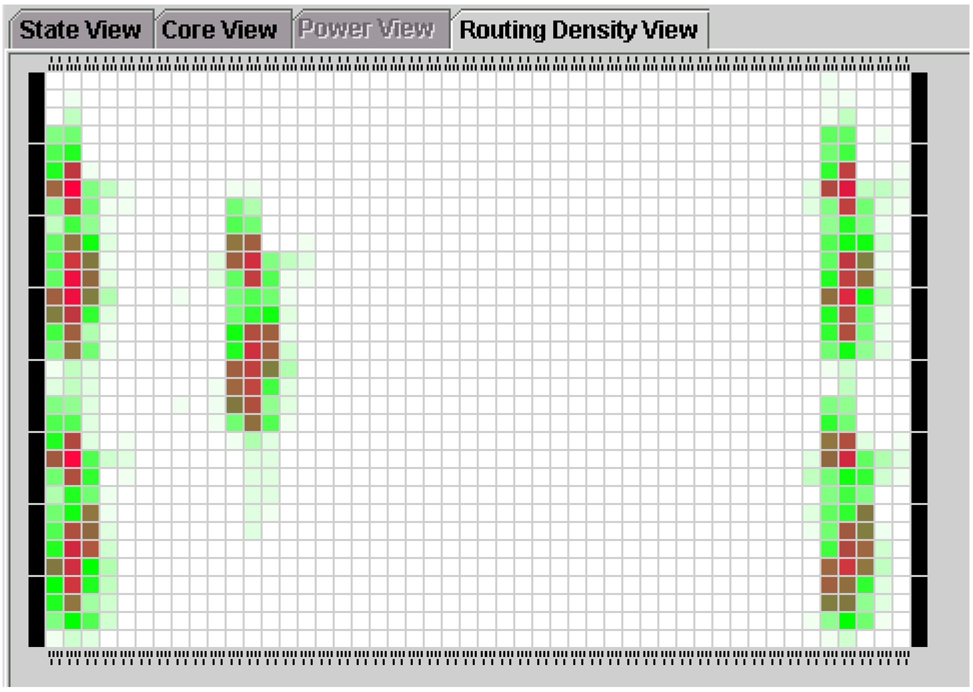


**Figure 38. BoardScope showing five processors**

Processors and their support components can be instantiated, moved and destroyed to allow other designs to be placed, in real-time. Using the `readHex()` command, programs can also be replaced. If more memory is required this can be supplied by wiring more BRAMs in with address decoding, or external memory added. The KCPSM processor is a useful mobile unit for use by this system, which can easily be added to, providing extensive I/O.

Very complex designs can be controlled and built up quickly and easily using the Reconnetics system.

### 5.4.7 Communicating Processes

This example shows the two different kinds of communication; between processes and between process and the run-time system.

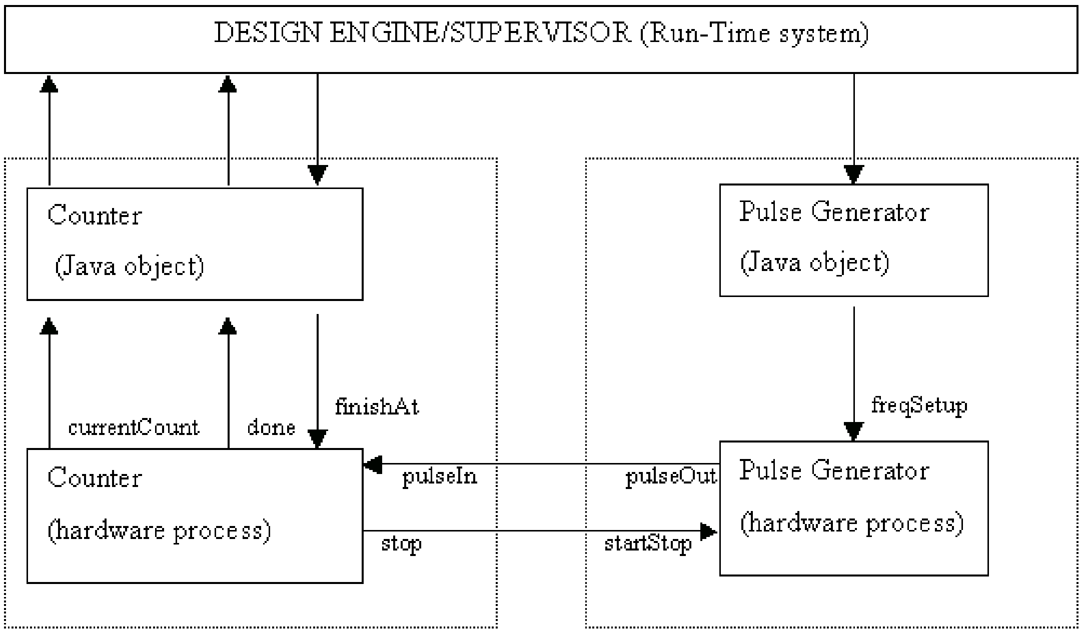


**Figure 39. Communicating processes**

Two processes are used; a 16-bit counter and a pulse generator. Figure 39 clearly shows how the Java program acts as a mediator between the run-time system and its hardware circuit, once instantiated. The `counter` counts pulses on its `pulseIn` port. The current value appears on its `currentCount` port. When the value (programmed via `finishAt`) has been reached, a stop signal is generated. The pulse generator process can be programmed with a frequency rate and stopped through its `startStop` port.

Using a DCP, the system is initialised and the processes loaded up, with manual place this time:

```
device(xcv1000, null1000GCLK3, localhost);

reset();
frequency(3.5); // set a clock speed of 3.5 MHz

process(counter, counter16);
process(generator, pulsegen);

loadAt(counter, 10,10);
loadAt(generator, 20,10);

//define test variables
var(currentCount, counter, COUNTOUT, 16);
var(finishAt, counter, COUNTIN, 16);
var(stop, counter, STOPOUT, 1);
var(pulseIn, counter, PULSEIN, 1);
var(done, counter, FINISHED, 1);

var(freqSetup, generator, FSETUP 8);
var(pulseOut, generator, PULSEOUT, 1);
var(startStop, generator, ONOFF, 1);
```

Links are established:

```
link(stop, startStop);
link(pulseOut, pulseIn);
```

The frequency and counter are programmed using the `inputVal()` command:

```
inputVal(freqSetup, 10);
inputVal(finishAt, 1000);
```

The process counts the pulses, taking `done` high when it has received the number of specified pulses. During the counting time, the run-time system simply waits:

```
wait(done, 1);
```

To show the transport of a process from one location to another we can capture it:

```
recapture(counter, new_counter16);
```

then release resources:

```
unlink(stop);
unlink(pulseOut);
unload(counter);
```

Finally, re-deploying somewhere else:

```
process(counter2, new_counter16);

load(counter2); // using auto-place
```

Variables can be then defined again and links re-established. The counter can be told the required amount of pulses and another `wait()` statement executed. When finished, the program can end with resources being released, through `unlink()` and `unload()`.

# 6. Results and Evaluation

## 6.1 Summary

The Reconnetics system, with its mobile design elements, represents an interesting and not insignificant development. It provides a high-level dynamic system for the control of FPGAs and associated design elements.

The current system has the following attributes:

- Flexibility and fast learning curve. By allowing any design to be used from any development tool, it maximises its flexibility for integration with other engineering environments. This being the case, engineers can re-use past designs and therefore are familiar with the components they are using.
- High-level. The language used is very high-level, hardware detail is avoided. Component designs can be seen as black boxes with connection ports, which can interconnect to other units. I/O can be accessed at various levels of abstraction and can be handled at any level of detail that the user is comfortable with. An I/O pad can be accessed, for example, through its particular name (such as AK13), its positioning value via row and column, or its JBits value.
- Mobility. MHPs are not only mobile within a device but over networks, allowing remote systems to be interacted with, updated and controlled.
- Re-use. MHPs can be re-used and distributed from the pool, as required. Utilising recapture, any changes at run-time can also be added to existing designs and placed in the pool as new components. MHPs are sufficiently self-

contained to be used outside of the run-time system and have a standard interface for communicating with a Java program.

- Generic. An aim of the system was to be able to use a wide range of devices. The entire Virtex range of devices can be used, which range from small devices, such as the XCV50, to the million gate devices such as XCV1000. Designs initially developed for one specific device can easily be transplanted to another, if there are sufficient resources available. This is done automatically.
- Dynamic. The system works in real time while devices are active. Not only can MHPs be instantiated while other activities take place on the FPGA but the there is a dynamic interplay between implemented hardware processes, their data state and the controlling system. Events can be reacted to and resources reallocated as required.
- Compositional and Scalable. Designs can be put together to build larger designs at any level of scale. An MHP may be an entire processor plus sub-system, for example, or may be as small as a few logic gates.
- Interactive. The system does not have to blindly place processes but can communicate with devices for exchange of data or to build designs on-demand, adding components as required. This may be on an event occurring which indicates some specific need. It could also be used in the case of some fault developing, such as a corrupted bitstream.
- Resource Management. The user can leave resource detail and management to the run-time system.

Some features and ideas were still being worked on and developed at the time of writing this thesis:

- Software functionality. MHPs can be made more “intelligent”, or have a greater presence on the software level by simply adding methods which can be called in run-time code. These have been labelled iMHPs to recognise their greater sophistication and have a structure which is in essence more hybrid in form between the software and hardware partitioning.

- Pure software processes and virtual channels. This is a natural extension of the current system. The idea here is to create virtual channels between instantiated hardware circuits and software processes.
- XML representations of FPGA node activity and mapping.

These will be explored in more detail in section 6.3.

## 6.2 Performance

| RC1000-PP Board (XCV1000 device) | | | | |
|---|---|---|---|---|
| | Local | | Over LAN 100Mbps | |
| | pulsecounter | KCPSM | pulsecounter | KCPSM |
| CLB size | 2 | 35 | 2 | 35 |
| Auto place (load) | 2000 | 8300 | 2600 | 8930 |
| User place (loadAt) | 1890 | 8000 | 2200 | 8812 |
| Destroy | 140 | 420 | 250 | 600 |
| Recapture | 4000 | 12000 | 5580 | 13200 |
| Recapture and deploy | 7500 | 15000 | 8570 | 16500 |
| Single wire link | 1000 | 1200 | 1580 | 1780 |
| Variable read | 800 | 1200 | 1500 | 1320 |

**Table 4. Performance of system in milliseconds**

There were three MHPs used for testing purposes. Two of them are listed as VHDL programs within appendix II, pulsecounter and pulsegenerator, which have 2 CLBs used apiece. The other is the freely available KCPSM processor detailed in appendix

V with 35 CLBs. Each of the seven tests (auto place, user place, destroy, recapture, recapture and deploy, single wire link and variable read) was tried on both the counter and processor MHPs.

Table 4 shows averaged performance figures in milliseconds of various tests with the system. They are intended to give a rough idea of system capabilities, rather than precise information. The system tested was pure Java with JBits API extensions. All data was gathered on 1GHz PCs and a Celoxica RC1000-PP board incorporating an XCV1000 Xilinx FPGA.

The 7 tests in table 4 consisted of:

- Auto place. This first test simply placed the MHP utilising the `load()` command which automatically finds the free resources for placement within the device matrix. This was repeated for both the pulsecounter and the KCPSM MHPs using local and remote (over network) machines.
- User place. The next test was similar but this time the user supplied the coordinates for placement. The timing reflects the fact that there was less processing to do, as an area does not have to be found with the required free resources.
- Destroy. The third test timed the opposite of loading and routing an MHP, that is, the releasing of used resources. The program loaded in the specified MHP, then timed the unloading. Again, this occurs, as in all the tests, in a local and remote context.
- Recapture. The forth test recaptured the loaded process after placement and recorded the result. The timing is taken from the start of the recapture itself to its completion. There were some varying factors here. As this recapture process relied on file operations it was dependant on the state of the filing system and whether the Java compiler was in memory already from previous use. The Java compiler in this instance was used dynamically to build the recaptured process.
- Recapture and deploy. The recapture and deploy test was similar to the last, except the recaptured process was deployed again and the two commands (`recapture()` and `load()`) timed.

- Single wire link. The single wire link test recorded how long it takes to link two processes with a single wire. In the first case it was between the pulsegenerator and pulsecounter; in the second between a KCPSM processor and pulsecounter. Obviously, in this case it was not the size of the processes involved but the width of the channel, which was one wire.
- Variable read. The final test involved reading a changing variable defined within the user program. Again, it was not the amount of CLBs used that was important but the actual width of the register. In the case of the pulsegenerator and pulsecounter the actual width was 4 and for the processor the width was 8. Results are again recorded for a local machine and over a network.

As mentioned, various factors must be taken into consideration when viewing the above timings. This includes language and memory considerations (such as whether a class has previously been loaded) as well in some cases the timing of access to the local filing system. Parallelism must also be considered for such devices, allowing that within such timings there could be much more being updated on the surface of the device at any given time rather than just one process, for example.

There are identifiable areas for optimisation and various techniques could be used to achieve speed-up. If speed is the critical factor, rather than adherence to the platform independence of Java, then certain functionality can be sped up using the Java Native Interface (JNI) to a pre-compiled language.

## 6.3 Comparison and Evaluation

Reconnetics improves on past attempts at run-time systems in many obvious ways and some more subtle ones. It combines the best features of systems mentioned in the background section and adds many aspects, such as interactivity and mobility. We have seen the high-level approach of certain languages to describe circuits in an object-oriented sense (JHDL) and a behavioural description (Handel-C). We have also looked at past approaches to universal dynamically configurable systems (Lysaght et al.) and systems which used C to control caching with mixed hardware/software functionality. Development environments that utilise dynamic sequencing for realisation of designs that change temporally (such as DYNASTY) were also described. A more sophisticated system, using an OS architecture approach, was reviewed, called RAGE, which uses geometrical transforming on hardware modules configured for the Xilinx XC6200.

Many of these systems had aspects that have been incorporated here. A basic outline of such features was outlined in the context section and developed towards the Reconnetics system. Many of them, while realising their own particular focus, were held back by reconfigurable technology, which was not capable of certain functions at that time. Virtex hardware has greatly improved, not only allowing reconfiguration while fully active but providing many dynamic communication features for manipulation and interaction with resources. The system presented here utilises these features fully, building on top of the accessibility provided by JBits, to give a high-level, dynamic, run-time system.

The RAGE system provides a reasonable comparison with Reconnetics. They share similar goals; for example, the approach is one of a design management system which manipulates the elements for eventual load to device. It also has a library which can be likened to the Reconnetics Pool. However, the system varies in the following:

- Control is device-specific. A driver needs to be written specifically for each device used, even within the same family.
- Hardware designs are device and system specific. Designs need to be built for, or adapted to, the system's design requirements.
- Mobility is considered only within a single linked device (i.e. not externally or networks).
- Combined software functionality is not considered, in either the designs themselves, or as separate processes, which could run concurrently.
- The symbolic language used for an intermediate state is not flexible. Reconnetics uses Java to provide an easily manipulated intermediate platform between the bitstream (and therefore the hardware development language) and

the physical circuit. Circuits described in this way can also be used outside of the main system.

- Lack of high-level language to control run-time system and placement.

The system therefore represents a substantial advance on previous systems, offering a way of controlling and interacting with designs in a flexible and efficient manner. It offers both a high-level language for control, or a way of incorporating design components into conventional Java programs.

The run-time system gives an advantage over conventional programming by allowing a “black-box” approach to handling the hardware objects, where very little knowledge is required from an engineering perspective, other than the details of the interface with an entity.

# 7. Future Work

A problem at the start of this project was the slowness associated with using Java. This was largely incurred by the overheads associated with the necessary use of JBits for accessing the low-level resources. The problem has largely been solved with much faster processors and buses available, which make any heavy duty analysis of routing and transfer of the resulting data very much quicker, with no noticeable delay.

While JBits generally proved to be excellent, future methods of access to reconfigurable resources could ideally offer more choice. JBits actually offers functionality on top of what is actually required for most intents and purposes. Some method would be useful which allows access in a symbolic format that is language independent, allowing the programmer to choose his software development path. It may be possible to provide some control language, similar to assembler, that allows access to and manipulation of, resources. This language could easily be incorporated into reconfigurable devices as a communicating "front-end", implemented as a state machine. This could simplify the approach to programming the device, in whatever language, with command strings being sent to directly control such detail as positioning and other resource information.

Another problem, that of slow network transfer of information, will become resolved as there is greater use of high speed broadband links, which are both physically connected and wireless, making communication with such devices very simple and efficient. Certainly, the current system proved very fast over 100Mbps LAN connections.

The various possible configurations of such run-time systems allow much room for experimentation. As we have seen here, probably the most useful two are the self-configuring stand-alone system and the distributed client-server architecture, which allows for remote reconfiguration.

It is possible Reconnetics could be placed in an embedded system, which is Java capable. Small embedded versions of Java are available which fit within these architectures, both in software form and as a "Java processor"[23]. Such systems can easily contain all the necessary software and hardware components to fully run the system as it stands, even outside a PC architecture. This includes the appropriate functionality for internet operation, utilising a TCP/IP stack.

The issue of transformation of circuits during run-time to fit available space is an important one. It must obviously be done quickly if real time organisation is to be sensibly achieved. The three methods looked into here have worked well, although there could be more work done in this area. Keeping designs constrained inside block areas, which are easily fitted together, is the quickest and least demanding in terms of processing time. Geometric transformation requires more processing time but generally is capable of working relatively speedily at run-time. Inter-mesh methods, also proposed here, allow the most usage of resources with little wastage but can be more demanding in terms of complexity and processing time.

As previously noted, there needs to be some way of remembering high-level mappings and definitions when reconnecting to a device, at some later time after its initial processes have been loaded. The method proposed for this is two new commands – loadMap and saveMap, within the DCP language, which create and store such a map that includes information on variables and process loading and position detail. This map could in theory be either stored on the server or client nodes. The map could be a  question of reprocessing the same DCP script, allowing for change which has taken place while supervision has been absent, or some form of document separately generated. This document could easily take the form of XML script, specially devised for this purpose. A description could be present in XML of the

[23] Such as the Lightfoot™ Java Processor by Digital Communication Technologies (DCT).

loaded circuits, actions taken, results to that execution point and other such data. It may also contain information on how far a DCP script was processed, for example, before disconnection. XML has several benefits for this kind of venture, the main ones being its simple text format for easy transmission between machines and its style of extensibility. A simple example would be the run-time system connecting to a device at some remote point, deploying processes, then disconnecting. At some future point the run-time system reconnects – needing to know the basic layout and details – the XML script would then be loaded (possibly using the remote IP address as its identifier in terms of file name) to give the additional information. The DCP can then be processed from a previous point.

More work could be done to realise the possibility of increasing functionality of MHPs on a software level. These have been named intelligent Mobile Hardware Processes (iMHP). There are two main issues here; adding functionality, which is supportive of the entity in its relationship with its environment and software functionality, which is a result of its partitioning between the two levels. Supportive functionality improves its ability to communicate and persist within a software/hardware system, where it may be so well contained it is more or less autonomous. Such an entity requires little in terms of a run-time system and is close to the idea of a mobile agent .

There is also the idea of incorporating pure software processes in the entity format, as perceived here. These entities can communicate and interact with other MHPs but have no hardware instantiation. A problem here could be that such processes are not locked in terms of clock time with the hardware processes. A way of dealing with this is either to provide some kind of software synchronisation, or a small amount of instantiated circuit which is capable of supplying a division of the clock timing to the software. Depending on the application, such synchronisation may be critical. If not, simple methods of flagging data exchange may suffice.

This leads to the idea of virtual components, in particularly, virtual channels. These channels, in the same way as physical hardware channels or wires, provide communication between processes. The processes may be software, or hardware, or combinations of both. Virtual resources are implemented in software, taking the place of hardware resources, which may be difficult to route or simply non-existent. Again, such software resources may be difficult to synchronise to external hardware, without some form of signal from the clocks in use, although in certain configurations this may not be a problem (for example, where the clock signal is derived from the CPU system clock or a physical link exists). Virtual channels can be implemented to link from the hardware to a software process, synchronous or asynchronous. Such a channel could be made to hold the process, through handshaking protocols, until its communicating entity picks up the information (as in occam), or be attached to a buffer, which may be part of the channel itself.

It would also be possible to simulate other needed resources, if required, such as memory or logic.

# 8. Conclusions

This work has shown the progression from a static-based engineering process to one which is dynamic and mobile. The goals of this project were initially cited as to provide a fluid, dynamic approach to hardware manipulation in several key areas:

- At design-time, through mixed methods of hardware design input.
- A means of encapsulating hardware entities in the software domain for transport and software presence/intelligence.
- At delivery time through mobility of component objects.
- At run-time through communication and interaction.

The system developed has answered each of these requirements, through a combination of theory, verbal argument and empirical demonstrations. We have seen how:

- The Reconnetics capture system enables *any* design from *any* source to be utilised. It does this by being able to manipulate resources at the most fundamental level, that of the bitstream.
- Java can be used as a suitable language for the encapsulation of the hardware implementation and software functionality can be added, in both support of hardware and to add extra capabilities.
- Extrinsic (over networks) and intrinsic (within devices) mobility are both viable and useful abilities.

- A run-time system such as Reconnetics can be used as a managed high-level tool to manipulate and interact with the mobile design components in real-time.

Reconfigurable technology will become more prominent in mainstream goods and research tools. This demand could feed back into the development of better and faster reconfigurable devices. Many companies are working on devices which marry conventional hardware and FPGA regions. The currently available Virtex–II Pro, from Xilinx, contains four IBM PowerPC® cores and very large amounts of configurable area including extremely fast communications transceivers. Such systems could benefit from the techniques related here. It would be possible to envisage an OS which is similar to the Reconnetics run-time system, which partitions programs, as needed, to software or hardware implementation, depending on its requirements. Where programs in systems such as .NET are translated into an in-between language, prior to targeting native code, in the same way the end target could just as easily be reconfigurable hardware.

It is quite possible that systems that are embedded entirely on one chip will become capable of reconfiguring themselves, given enough memory. Such a system could be simply swapping cached configurations, or this may be more complex, as with Reconnetics, which can handle any size of entity, big or small.

Networked distribution of hardware updates is set to become an important feature of electronic equipment of the near future and with this, the need for systems which are capable of achieving it efficiently and with flexibility. The system presented here can do this and provides many opportunities for experimentation with remote reconfiguration. A distributed approach may be used for fault correction, system update, or interactive methods of load-on-demand.

The development of the Reconnetics system has also pointed out an important area of research, relating to hybrid design objects which have functionality on both hardware and software levels. We have seen the benefits here of containing these processes and allowing them a degree of software functionality, which acts as an interface. We have also seen it may be possible to extend the overall functionality, by incorporating code that is in addition to its hardware support. This partitioning is contained in one entity, which becomes a hybrid software/hardware object, capable of being transferred from one device to another.

Reconfigurable computing will continue to open new grounds of inspiration and research, the fruits of which are already becoming evident in commercial and military applications. It was the object of this work to further add to the field by asserting the flexibility of the medium within the remit of the above points.

# Glossary

Application-Specific Integrated Circuit (ASIC)

A device whose initial stages of manufacture are design independent and the final photographic mask is design dependent.

Block Random Access Memory (BRAM)

A small memory (usually kbits in size) unit on the surface of a FPGA.

Built-In Self-Test (BIST)

The extra circuitry added that enables self-diagnostic checks.

Complex Programmable Logic Device (CPLD)

These programmable logic devices increased the complexity and density of logic on the chip beyond the capabilities of the PLA. This was done by creating a collection of such units on the same device with a programmable interconnect structure allowing a rich variety of design possibilities.

Configurable Logic Block (CLB)

A discrete programmable unit on the surface of a FPGA, made up of logic, memory and switches.

dynamically reconfigurable hardware

A circuit, when implemented on a chip, can be customised on-the-fly while remaining resident in the system.

fault tolerance

A quality that describes a system’s resistance to failure. A fault-tolerant system is one that is designed to self-correct and continue under many circumstances. The term can be used to describe the ability of a state machine to force itself into a known state whenever an unforeseen condition or hardware glitch is encountered [41].

Field Programmable Gate Array (FPGA)

A programmable logic device (chip) that can be programmed in the field for a particular application. All chip manufacturing processes are independent of the particular circuit being implemented, so it is more generic and cheaper than standard cell devices. Logic gates and other resources, such as memory, are laid out in a fixed and structured way on the silicon. This does mean, however, that circuit density will not be as great as standard cell devices.

finite state machine

*See state machine*

flip-flop

An edge sensitive memory device.

floorplan

This is an area on a silicon chip, not including the input, output and bi-directional buffers around its periphery.

formal verification

A method of mathematically verifying the logic synthesised from a hardware model. Formal verification is the process of building a mathematical model of logic contained in an HDL model and comparing it with the actual synthesised logic, using specialised algorithms. This is particularly useful in the verification of large complex systems, where excessive functional verification vectors would be needed, using a simulation verification technique.

functional test vectors

The input stimuli used during simulation to verify an HDL model operates functionally as intended.

gate level

A low-level behavioural model of a circuit, described in terms of gate primitives from a technology-specific library, or possibly from some generic technology, independent of gates.

glue logic

Logic used to interface more complex circuits together.

Hardware Description Language

A software computer language used for the purposes of modelling and designing hardware circuits. These can be structural, describing the design in terms of its connections and resources, or behavioural, in terms of an algorithm determining its functionality.

logic synthesis

The process of optimising boolean equations at the logic level, mapping them to a technology-specific library of cells and then optimising at the gate level using timing and area information from the cells in the technology library.

Look-Up Table (LUT)

A block of memory which can be used to store various logic functions, accessed through an index.

netlist

A file containing the representation of a design at the cell level in VHDL, Verilog or EDIF etc. The cell level is also the gate level if all cells are gate level cells. A netlist file contains a list of cells, usually from a technology specific library and identifies how the cells are interconnected.

optimisation

A general term used to describe the process of improving the structural configuration of a circuit model, given certain area, timing and possibly power constraints.

partitioning

1. The process of dividing a design into smaller pieces, either through the HDL design of concurrent hardware modules or, by using a synthesis tool to automatically partition a flattened netlist.
2. The process of dividing a design solution between hardware and software components, according to efficiency based on the constraints of resources, or the problem itself.

place and route

The process of transforming the gate-level representation of a circuit into a program file that may be used to program an FPGA device. There are two stages involved in this: placing the required logic into logic cells and to route the signals that connect the logic cells and I/O via the interconnect wires.

process

An entity, which may be hardware or software, possessing its own thread of control.

programmable interconnect

Refers to the routing wires and multiplexors between logic cells and I/O, which can be programmed to connect such units together.

Programmable Logic Array (PLA)

The first programmable logic devices containing a two-level structure of AND and OR gates with user programmable connections. Any logic function up to a certain level of complexity could be accommodated.

Programmable Logic Device (PLD)

A device in which the logic functionality is determined by the user.

propagation delay

The delay of a signal passing from one point in the circuit to another. A propagation delay may be: a delay passing along a wire in the physical circuit on the chip, the delay of a signal being passed through a cell, or the total delay through multiple cells and associated wires. Propagation delays are determined by cell drive capability and capacitive loading. Capacitive loading consists of the input capacities of cells connected to the drive cell and the total capacitance on the interconnecting wire network.

register

A memory device containing more than one latch or flip-flop that are all clocked from the same signal.

register transfer level (RTL)

The model of a circuit described in a hardware description language that infers memory devices.

RTL synthesis

The process of converting an HDL model described at the register transfer level (RTL), to the logic level and then to the gate level performing combinational logic optimisation at each stage. Register transfer level synthesis does not optimise (add or remove) registers. This definition of RTL synthesis encompasses logic level synthesis, logic level optimisation and gate level optimisation.

state machine

The model of a circuit, or its hardware implementation, that cycles through a predefined sequence of operations (states).

Static Random Access Memory (SRAM)

A type of RAM used in FPGA which does not require periodic refresh and therefore is generally easier to implement. The memory stays intact while power is applied, until it is overwritten with a new value.

state map

A map of flip-flop values, holding a set of states.

structure map

A map of the physical resources.

synthesis

A general term used to describe the process of converting the model of a design, described in an HDL, from one behavioural level of abstraction to a lower, more detailed, behavioural level.

test bench

*See test harness.*

test harness

A test harness is an HDL model used to verify the correct behaviour of a hardware model. Normally written in the same HDL language as the hardware model being tested. A test harness will:

- instantiate one or more instances of the hardware model under test,
- generate simulation input stimuli (test vectors) for the model under test and collate output responses (output vectors),
- compare output responses with expected values and possibly automatically give a pass or fail indication.

tri-state logic

Electronic logic which includes a third state, high-impedance, usually signified by the symbol Z. A wire in this state can be considered as disconnected, allowing it to be driven by logic sources.

# Acronyms

| | |
|---|---|
| ALU | Arithmetic and Logic Unit |
| ABEL | Advanced Boolean Equation Language |
| ASIC | Application Specific Integrated Circuit |
| BIST | Built-In Self-Test |
| BRAM | Block Random Access Memory |
| CAD | Computer Aided Design |
| CAE | Computer Aided Engineering |
| CFB | Configurable Function Block |
| CLB | Configurable Logic Block |
| CPLD | Complex Programmable Logic Device |
| CPU | Central Processing Unit |
| CSP | Communicating Sequential Processes |
| DARPA | Defence Advanced Research Projects Agency (USA) |
| DCP | Design Control Program |
| DES | Design Engine/Supervisor |
| DMA | Direct Memory Access |
| DRC | Design Rule Check |
| DRL | Dynamic Reconfigurable Logic |
| DSP | Digital Signal Processing |
| EDIF | Electronic Design Interchange Format |
| EPROM | Electronically Programmable Read Only Memory |
| EEPROM | Electrically Erasable and Programmable Read Only Memory |
| EDA | Electronic Design Automation |
| FIR | Finite Impulse Response |

| | |
|---|---|
| FPAA | Field Programmable Analogue Array |
| FPGA | Field Programmable Gate Array |
| FSM | Finite State Machine |
| GA | Genetic Algorithm |
| GCLK | Global Clock |
| HDL | Hardware Description Language |
| iMHP | Intelligent Mobile Hardware Process |
| I/O | Input/Output |
| IC | Integrated Circuit |
| IEEE | IEEE Institute of Electrical and Electronics Engineers |
| IOB | Input/Output Block |
| ISP | In-System Programmable |
| JHDL | Java Hardware Description Language |
| KCPSM | Constant (K) Coded Programmable State Machine |
| LUT | Look-Up Table |
| MCM | Multi-Chip Modules |
| MHP | Mobile Hardware Process |
| NCD | Native Circuit Description |
| NGD | Native Generic Database |
| PAL | Programmable Array Logic |
| PAR | Place and Route |
| PCB | Printed Circuit Board |
| PCF | Physical Constraints File |
| PCI | Peripheral Component Interconnect |
| PLA | Programmable Logic Array |
| PLD | Programmable Logic Device |
| PSM | Programmable Switch Matrix |
| RAM | Random Access Memory |
| RTL | Register Transfer Level |
| RTR | Run-Time Reconfigurable |
| ROM | Read Only Memory |
| SPLD | Simple Programmable Logic Device |
| SRAM | Static Random Access Memory |
| TCL/TK | Tool Command Language/Tool Kit |

| | |
|---|---|
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| UCF | User Constraints File |
| VHDL | VHSIC Hardware Description Language |
| VLSI | Very Large Scale Integration |
| XHWIF | Xilinx Hardware Interface |
| XNF | Xilinx Netlist File |
| XML | Extensible Mark-up Language |

# References

[1] A. Aho, R. Sethi and J.D. Ullman, *Compilers Principles, Techniques and Tools*, Addison Wesley, 1986.

[2] A.W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, 1998.

[3] P.J. Ashenden, *The Student's Guide to VHDL*, Morgan Kaufmann, 1998.

[4] P. Bellows and B. Hutchings, *JHDL – An HDL for Reconfigurable Systems*, *Proceedings of the IEEE Symposium Field Programmable Custom Computing Machines*, 1998.

[5] D. Van den Bout, *The Practical Xilinx Designer Lab Book*, Prentice Hall, 1999.

[6] J. Burns et al., *A Dynamic Reconfiguration Run-Time System*, *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, 1997.

[7] K. Compton and S. Hauck, *Reconfigurable Computing: A Survey of Systems and Software*, Dept. of Electrical and Computer Engineering, Northwestern University, IL, http://www.ece.nwu.edu/~kati/publications.html, 2000.

[8] K. Compton and S. Hauck, *Configuration Caching Techniques for FPGA*, Dept. of Electrical and Computer Engineering, Northwestern University, IL, http://www.ece.nwu.edu/~kati/publications.html, 2000.

[9] A. DeHon, *Reconfigurable Architectures for General-Purpose Computing*, Ph.D. Thesis, Massachusetts Institute of Technology, 1996.

[10] R. Enzler, *The Current Status of Reconfigurable Computing*, Technical Report, Swiss Federal Institute of Technology, Electronics Laboratory, 1999.

[11] J. Ferber, *Multi-Agent Systems*, Addison-Wesley Longman, 1999.

[12] E. R. Harold, *Java Network Programming*, O'Reilly, 2000.

[13] S. Hauck, *The Chimaera Reconfigurable Functional Unit*, *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, 1997.

[14] J.R. Hauser, *Augmenting a Microprocessor with Reconfigurable Hardware*, Ph.D. Thesis, University of California, 2000.

[15] A. Hoare, *Communicating Sequential Processes*, Englewood Cliffs, Prentice Hall, 1985.

[16] INMOS, *Occam 2 Reference Manual*, Englewood Cliffs, Prentice Hall, 1988.

[17] S. Kung, *VLSI Array Processors*, Prentice Hall, USA, 1988.

[18] D. Lea, *Concurrent Programming in Java Design Principles and Patterns*, Addison Wesley, 2000.

[19] E. Lemoine and D. Merceron, *Run Time Reconfiguration of FPGA for Scanning Genomic Databases*, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 90–98, 1995.

[20] W. Luk and S. Guo, *Visualising Reconfigurable Libraries for FPGAs*, *Proceedings of the 31st Asilomar Conference on Signals, Systems and Computers*, pages 389-393, IEEE Computer Society, 1998

[21] W. Luk, T. Wu and Ian Page, *Hardware-software Co-design of Multidimensional Programs*, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 82–90, 1994.

[22] P. Lysaght and J. Dunlop, *Dynamic Reconfiguration of FPGAs*, *More FPGAs*, Abingdon EE & CS Books, England, pages 82-94, 1994.

[23] P. Lysaght et al., *Prototyping Environment for Reconfigurable Logic*, *Proceedings of 5th International Workshop FPL '95*, 1995.

[24] D. MacVicar and S.Singh, *Accelerating DTP with Reconfigurable Computing Engines*, *Proceedings of the 8th International Workshop on Field Programmable Logic and Applications*, Volume 1482 of *Lecture Notes in Computer Science, pages 391–395*, 1998.

[25] W.H. Mangione-Smith and B. Hutchings, *Configurable Computing: The Road Ahead*, *Proceedings of the Reconfigurable Workshop 1997*, Geneva, Switzerland, pages 81–96, 1997.

[26] M. Mano and C.R. Kime, *Logic and Computer Design Fundamentals*, Prentice Hall, 2000.

[27] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin and B. Hutchings, *A Reconfigurable Arithmetic Array for Multimedia Applications*, *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 135–143, 1999.

[28] N. McKay and S.Singh, *Dynamic Specialisation of XC6200 FPGAs by Partial Evaluation*, Field Programmable Logic and Applications, Tallinn, Estonia. Springer-Verlag. 1998.

[29] N. McKay and S. Singh, *Debugging Techniques for Dynamically Reconfigurable Hardware*, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1999*.

[30] N. McKay and S. Singh, *Debugging Techniques for Dynamically Reconfigurable Hardware*, *Proceedings of 8th International Workshop Field Programmable Logic and Applications* (FPL 1998), Spinger-Verlag, 1999.

[31] S.J. Metsker, Building Parsers with Java, Addison Wesley, 2001.

[32] E. Mirsky and A. DeHon, *MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources*, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, 1998.

[33] Z. Navabi, *VHDL Analysis and Modelling of Digital Systems*, McGraw-Hill, 1998.

[34] J. Nelson, *Programming Mobile Objects with Java*, Wiley, 1999.

[35] I. Page and W. Luk, *Compiling Occam into FPGAs*, *FPGAs*, W.Moore and W.Luk, Eds., Abingdon EE&CS Books, England, 1991.

[36] I. Page, *Compiling Video Algorithms into Hardware*, Advice97, EDA Ltd, London, 1997. Reprinted Embedded Systems Engineering, September 1997.

[37] M. Rencher and B. Hutchings, *Automated Target Reconfiguration on Splash 2*, J. Arnold and K.L Pocek Eds., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 192-200, Napa, CA, 1997.

[38] S. Scalea, *A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing*, Sanders (Lockheed Martin), Hudson, 1998.

[39] R. Sidhu, S. Wadhwa, A. Mei and V.K. Prasanna, *A Self-Reconfigurable Gate Array Architecture*, *Proceedings of the 10th International Workshop on Field Programmable Logic and Applications*, 2000.

[40] S. Singh and R. Slous, *Accelerating Adobe Photoshop with Reconfigurable Logic, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 216–225, 1998.

[41] S. Sinha et al., *Tunable Fault Tolerance for Runtime Reconfigurable Architectures*, *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines*, page 185, 2000.

[42] K. Skahill, *VHDL for Programmable Logic*, Addison Wesley, 1996.

[43] D.J. Smith, *HDL Chip Design*, Doone Publications, 1999.

[44] A. Thompson, *Hardware Evolution*, Springer-Verlag, 1998.

[45] M. Vasilko and G. Benyon-Tinker, *Design Visualisation for Dynamically Reconfigurable Systems*, in *Proceedings of 10th International Workshop Field Programmable Logic and Applications (FPL 2000)*, Springer-Verlag, pages 131–140, 2001.

[46] M. Vasilko, *Automatic Temporal Floorplanning with Guaranteed Solution Feasibility*, in *Proceedings of 10th International Workshop Field Programmable Logic and Applications (FPL 2000)*, Springer-Verlag, pages 656–664, 2001.

[47] J.F. Wakerly, *Digital Design*, Prentice Hall, 2000.

[48] N. Weaver et al., *Object-Oriented Circuit Generators in Java*, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[49] J.L. Weber, *Using Java 2 Platform (Special Edition)*, pages 1123–1126, Que, 2000.

[50] M. Wirthlin and B. Hutchings, *Sequencing Run-Time Reconfigured Hardware with Software*, *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, FPGA 1996, pages 122–128, 1996.

[51] M. Wirthlin and B. Hutchings, *Improving Functional Density Through Run-Time Constant Propagation*, *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, FPGA 1997, pages 86–92, 1997.

[52] Xilinx, Home Page, `URL: http://www.xilinx.com`

[53] Xilinx, *Dynamic Reconfiguration*, Xilinx Application Notes, XAPP093, Xilinx Inc. 1998.

[54] Xilinx, *Technical Overview*, Xilinx Application Notes XAPP097, Xilinx Inc. 2000.

[55] Xilinx, *Using the Virtex Block SelectRAM+ Features*, Xilinx Application Notes XAPP130, Xilinx Inc. 2000.

[56] Xilinx, *8-Bit Microcontroller for Virtex Devices*, Xilinx Application Notes XAPP213, Xilinx Inc. 2000.

[57] Xilinx, *Virtex 2.5V Field Programmable Gate Arrays Data Sheet*, Xilinx Inc. 2001.

# Appendix

# Appendix I: Language Reference

**Quick Reference:-**

| | |
|---|---|
| clear() | Reset all variables and connections |
| clock(on/off) | Turn the clock on or off |
| do { ... } while ( ) | Loop construct |
| device(device, canvas, IP/localhost) | Initialise a connection to a device |
| frequency(n MHz) | Set the board clock frequency |
| gclk(n) | Set all active processes to a specific clock |
| if () { ... } else { … } endif | Conditional construct |
| inputON(varName) | Turn an input ON |
| inputOFF(varName) | Turn an input OFF |
| inputVal(varName, value) | Set a bus or single to an integer value |
| link(varName, varName) | Connect a bus or single wire between processes |
| linkDefault(varName) | Link to a default IOB |
| linkIOB(varName, side, index, unit) | Link to an IOB |
| linkPad(varName, padName) | Link to a named Pad |
| load(procName) | Load a process, auto-position |
| loadAt(procName, row, col) | Load a process at row, col |
| message(String) | Print a message to the screen |
| process(name, type) | Associate a process with a type |
| print(var) | Print the contents of a variable to the screen |
| readBack() | Forces a readback of the device |
| readHex(row, col, width, filename) | Reads an Intel Hex format file into a BRAM |
| recapture(procName, newName) | Recaptures the process |
| reload(procName) | Load both structure and state-map of a process |
| repeat(n) { ... } endrep | Loop n times |
| reset() | Reset the device |
| resetBram(row,col,width) | Reset all values inside a BRAM to zero |
| setBram(row,col,offset,width,val1...valn) | Set values inside a BRAM |
| stateMapAt(procName, row, col) | Load the state-map of a process at a specific position |
| step(n) | In step mode, step the clock n times |
| tree(varName) | View a connection tree starting at varName |
| unload(procName) | Unload a process |
| unlink(varName) | Unlink a bus or single wire between processes |
| var(name, procName, portID, n) | Define a variable, n wires wide |
| wait(varName, n) | Wait for a specific location to reach n |
| write(filename) | Write the current configuration to a file |
| // | Comment line |

**clear**

---

Clears all current variables and disconnects from the device.

Example:

```
clear();
```

**clock**

---

Switches the device’s clock on or off.

Example:

```
clock(on);
```

Turns the device’s clock on.

**do { ... } while ()**

---

Allows repetition of the enclosed block until a specific state is reached.

```
do  {

     message(“Trying again...”);
     step(1);

} while (pulsecounter_out != 4);
```

The enclosed block which includes a message to the user and a board clock step is executed until `pulsecounter_out` reaches four.

**device**

Initialises a connection to a device. The first parameter is the device type, followed by the canvas used and finally the connection description which can be an IP address or `localhost`.

The canvas type is usually a “blank” bitstream for a particular device type but can be an already developed design to begin with.

Example:
```
device(xcv300, null300GCLK1, localhost)
```

**frequency**

Determines the on-board clock frequency. The exact range is determined by the device’s specifications. The number supplied is a floating point value in MHz.

Example:
```
frequency(1.7);
```

Sets the frequency of the on device clock to 1.7 MHz.

**gclk**

Sets all currently placed processes to a specific global clock.

Example:
```
gclk(0);
```

Any used logic blocks are set to global clock 0.

**if () { ... } else { … } endif**

This is a conditional construct. If the condition is fulfilled the block or single statement is executed.

Example:

```
if (pulsecounter_out == 10) {

     message(“I have had 10 pulses”);

}  endif;
```

If `pulsecounter_out` becomes equal to 10 then the message is output to the screen.

**inputON**

Turns a defined locations input point on.

Example:

```
inputON(pulsecounter_in);
```

The input point `pulsecounter_in` is set open for input.

**inputOFF**

Turn a defined locations input point off.

Example:

```
inputOFF(pulsecounter_in);
```

The input point `pulsecounter_out` is closed.

**inputVal**

---

Allows a defined location's inputs to be set with a specific value, the size of which is determined by its data width.

Example:

```
inputVal(pulsecounter_in, 1);
```

The port `pulsecounter_in` is set to one; which if the width of such location was one wire would be the maximum value that it could hold.

**link**

---

Creates a link between a designated point(s), defined by `var`.

Example:

```
link(counter_out, pulsecount_in);
```

The output at position `counter_out` is linked to `pulsecount_in`. The number of wires linked is determined by the definition of that `var` port.

**linkDefault**

---

Links a named group or single point to its original point. This may be a I/O block, for example. Allows designs to be instantiated as originally intended.

Example:
```
linkDefault(counter_out);
```

The port `counter_out` is linked to its original design location; probably of type IOB.

**linkIOB**

---

LinkIOB uses the JBits model to connect points. IOBs are in this instance referenced by side, index and unit.

Example:

```
linkIOB(counter_out, left, 4, 2);
```

An IOB pad at left side, 4 position and unit 2 is linked to `counter_out`. This command was added to allow JBits model coordinates.

**linkPad**

---

This command allows a single point to be connected to a I/O pad, as identified by its name, defined by Xilinx. This command is very device specific; currently only the XCV300 and XCV1000 are supported.

Example:

```
linkPad(counter_out, AK21);
```
The pad `AK21` (defined by Xilinx as a device I/O position) is linked to `counter_out`.

**load**

---

This command loads a previously defined process into the device matrix at a point decided by the DES. Basically, this point is decided by an algorithm which takes into consideration such aspects as the density of routing and proximity of required resources.

Example:

```
load(counter);
```

Loads the pre-defined process “counter” on the board’s matrix, using auto-placement to work out a suitable position.

**loadAt**

---

Loads a named process at row, column in the device.

Example:

```
loadAt(counter, 10, 11);
```

Loads the process `counter` to a base position of row 10 and column 11. An error condition is given should the resources of the required surface area not be met.

**loadStateAt**

---

Allows a state map to be positioned as a template over a process. This may result in the process being in a specific state, although this depends largely on the way the process is designed.

Example:

```
loadStateAt(my_old_process, 10,10);
```

**message**

Sends a message to the screen.

Example:

```
message(“The process is finished”);
```

**process**

Associates the name of a process class (stored in the pool) with a process.

Example:

```
process(mycounter, count4);
```

A new process is initialised called `mycounter` of process class `count4`.

**print**

Writes a defined locations value to the screen.

Example:

```
print(pulsecounter_out);
```

Outputs the value contained at `pulsecounter_out` to the screen.

**readBack**

Forces a read back of the device which brings the current Reconnetics canvas in line with the state of an external device.

Example:

```
readBack();
```

**readHex**

Read a file in Intel Hex format (output from most assemblers) into a BRAM. Useful for supplying programs or data to instantiated processors or similar designs.

Example:

```
readHex(0, 0, 16, “myprog.hex”);
```

**recapture**

This command allows a placed process to be recaptured. The process is recompiled as a new process and added back into the design pool.

Example:

```
recapture(oldProcessName, newProcessName);
```

**reload**

This command will load a recaptured process using automatic placement. Both the state map and the structural map are used.

Example:

```
reload(my_old_process);
```

**reset**

---

Resets the device to its default state.

Example:

```
reset();
```

Clears the currently connected device of used resources and initialises them to their default state.

**resetBram**

---

Resets the content in the block RAM (BRAM).

Example:

```
resetBram(0, 0, 8);
```

The example resets the BRAM of data width 8 to all zeros.

**repeat(n) { ... } endrep;**

---

Repeats a section of code n times.
Example:

```
repeat(5) {
     message(“looping”);
     step(1);
} endrep;
```

The enclosed block which includes a message to the user and a board clock step is executed five times.

**setBram**

---

Sets the content in the block RAM (BRAM).

Example:

```
setBram(0, 0, 0, 8, 65, 66, 67, 68, 69, 70, 71, 72, 73);
```

This example sets the BRAM at row=0, col=0, at offset 0, data width 8, to the values following.

**stateMapAt**

---

The state map is loaded in at the row and column positions.

Example:

```
StateMapAt(my_proc, 10, 10);
```

After being recaptured the process `my_proc` has a structural and a state representation, this command loads in the flip-flop states to positions with an origin of 10, 10.

**step**

---

By setting the clock off and using this command, processes can be checked for correct functioning. A way to do this would be to create watch points over the board and print the values at these locations to the screen.

Example:

```
step(2);
```

Steps the board twice.

**tree**

---

Outputs a tree of connections to the screen. Use for diagnostic route checking.

Example:

```
tree(counter_out);
```

Gives a text-based output of a hierarchical tree of connections which lead from `counter-out`.

**unload**

---

Shuts down a process and releases associated resources.

Example:

```
unload(counter);
```

The defined and loaded process `counter` is extracted from the board, wiring and logic resources are released.

**unlink**

---

Unroutes all nets associated with a given, defined point.

Example:

```
unlink(counter_out);
```

The wires connected to `counter_out` are unrouted and resources released.

**var**

---

Defines a point which may be linked to, observed or interacted with.

Example:

```
var(myreg, counter, data_out, 4);
```

The first parameter is the name that is used to reference that specified point or points in future commands. The remaining parameters identify the exact location. In this case a process called `counter`, with a set of output points, that have a width of 4.

**write**

---

The current bitstream is written to a file. Allows a configuration to be analysed with other tools, or for archive purposes.

Example:

```
write(“myconfiguration.bit”);
```

The file `myconfiguration.bit` is created and the current configuration bitstream is written to it.

//

---

This is the indicator of a line comment.

Example:

```
// This program is a test
```

## Operators

Several operators can be used for logical tests:

| | |
|---|---|
| `==` | equals |
| `!=` | not equal |
| `<` | less than |
| `>` | greater than |
| `<=` | less than or equal to |
| `>=` | greater than or equal to |

## Appendix II: Design Examples

*Pulsecounter VHDL code*

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY pcount IS
PORT
(

     clkin : in STD_LOGIC;
     data_in : in STD_LOGIC;
     count : out std_logic_vector(3 downto 0)

);
END pcount;

ARCHITECTURE pcount_arch OF pcount IS

signal count_v : std_logic_vector(3 downto 0);

BEGIN

     PROCESS (clkin,data_in)
     BEGIN
          IF (clkin'event AND clkin='1') THEN
               IF (data_in='1') THEN
                         count_v <= count_v + "0001";
               END IF;
          END IF;
     END PROCESS;
     count <= count_v;
END pcount_arch;
```

*Pulsecounter UCF file*

```
NET count<3>     LOC =  P53;
NET count<2>     LOC =  P52;
NET count<1>     LOC =  P50;
NET count<0>     LOC =  P49;
NET data_in      LOC =  P48;
NET clkin        LOC =  P89;
```

*Pulsegenerator VHDL code*

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.all;

ENTITY seq IS
PORT
(

     clkin : in STD_LOGIC;

     count  : out std_logic_vector(3 downto 0)

);
END seq;

ARCHITECTURE seq_arch OF seq IS

signal count_v : std_logic_vector(3 downto 0);

BEGIN
     PROCESS (clkin)
     BEGIN
        IF (clkin'event and clkin='1') THEN
                     count_v <= count_v + "0001";
        END IF;
     END PROCESS;

     count<=count_v;

END seq_arch;
```

*Pulsegenerator UCF file*

```
NET count<0>  LOC=P53;
NET count<1>  LOC=P54;
NET count<2>  LOC=P55;
NET count<3>  LOC=P56;
```

*Pulsecounter Handel-C code*

```
set clock = external "P89";

void main(void)
{
     unsigned int 4 count;


     interface bus_in(unsigned int 1) data_in() with { data =
          { "P48" }};

     interface bus_out() output(count) with { data =
          { "P49","P50","P52","P53" }};


     while (1) {
          if (data_in.in==1) {
               count++;
          }


     }

}
```

Equivalent Handel-C code for the `Pulsecounter` demo. Note: A UCF file should still be included for capture, exactly the same as in the VHDL code.

## Appendix III: Reconnetics Class Descriptions

### Main System

`bramPoint`

Holds details on used BRAM site.

`clbList`

Contains methods for searching and manipulating a list of CLBs.

`clbObject`

A sub-matrix containing all the resource details for a process. This sub-matrix may be manipulated to provide a new shape to resource coverage and layout.

`clbPoint`

Contains details and methods for handling a single CLB unit.

It can contain its original position data as well as new information.

`clbPort`

Holds data on a single wire, such as, name, direction, the CLB it belongs to, the number of wires in this group etc.

`conPort`

Contains details on a net of wires from a single source to one or many sinks.

`connectServer`

Handles communication over TCP/IP to devices.

`copyBits`

Methods for transferring resources within or between bitstreams.

`DES`

Design Engine/Supervisor is the main engine at run-time.

Executes commands as found in DCP script supplied by the user.

`fpgaObject`

An object of this type is a generated program which will build its circuit anywhere on a device matrix.

`grabObject`

Starting point for capture of a process held within a bitstream.

`handleCLB`

Copies CLB resources from one area to another, within a single bitstream or between two.

`intelHex`

Contains tools for loading Intel Hex format files, used mainly for loading BRAMs with data.

`IOS`

Utility for reporting details of a bitstream, such as, I/O used, and routing.

`ioScan`

Finds IO points and their associated source/sink points.

`jObjectifier`

Main class for capture and production of  Java code from a bitstream for building an MHP.

`jRecap`

Run-time version of `jObjectifier`; isolates a single process from many even when routed, scans resources used and generates equivalent Java code.

`jStateCap`

Run-time capture of the flip-flops within a named processes area. Produces a state-map in the form of a Java program which can rebuild it in a new bitstream. Similar technique to the structural capture within `jRecap`.

`loadObject`

Holds information on a referenced `fpgaObject` which has been instantiated. Such detail is at a higher level than the circuit detail, so is held in this object. It may hold the origin and name of the process, for example.

`myClassLoader`

Class loader for dynamic loading of classes at run-time. Adapted from dynamic classloader in Weber, Using Java 2 Platform [49].

`padInterface`

Methods for associating specific detail held on a device with references used in a DCP.

`padPin`

Holds data on a used

pin.

`parser`

Tools for parsing DCP script.

`preCapture`

Precapture system.

`process`

Holds information on a process when defined within the DCP, contains a reference to a `loadObject`.

`resetBit`

Resets a bitstream, ensuring all logic and routing is free and unused.

`runDosProcess`

Starts up separate DOS processes as required, used for dynamic compiling of Java at run-time.

`runReconnetics`

Main starting point for the system.

`scanBram`

Ascertains what BRAM resources are used.

`scanBits`

For finding used resources.

`setFlips`

This will set specified flip-flops to values. Used by `stateObject` to reconstruct a state map.

`shapeShift`

Geometrical transformer, can change the shape or compress

`stateObject`

An object of this class contains flip-flop state data generated by `jStateCap`. Calling its build method allows its reconstruction in a new bitstream.

`treeHandler`

Contains methods for manipulating and investigating routing.

`var`

Class for defining locations of a device matrix as `vars`.

a process held within a bitstream.

## Appendix IV: System Requirements

Reconnetics minimum requirements:

200 MHz PC + 64Mb

Internet connection for remote communication

Java 1.2

Xilinx JBits package 2.8

Design tools and hardware synthesis software. (Such as Xilinx Foundation, Handel-C etc.)

## Appendix V: KCPSM Xilinx Processor Details

KCPSM is a 8-bit microcontroller for the Virtex, VirtexE and Spartan II devices. It occupies just under 35 Virtex CLBs – less than 10% of the smallest XCV 50 device and less than 0.6% of the XCV 1000 device. A single block RAM is used to form a ROM store for programs. The performance is around 25 to 35 MIPS range depending on device speed grade.

It requires no external support and can therefore be totally embedded into a Virtex or Spartan device.

It features a 16-bit instruction word and an 8-bit data path. There are 16 general purpose registers and various flags. The KCPSM has 256 input ports and 256 output ports. A separate bus is available for easy address decoding, together with READ and WRITE strobe lines. An interrupt line is also available to initiate interrupt software routines.

The assembler supplied with KCPSM can output an EDIF file, describing and initialised with a program ready for implementation. An even easier way to program is to use Reconnetics to download a Intel hex file to the BRAM when a design is instantiated.

The KCPSM processor is supplied as in EDIF format, which can be easily placed in a design. The download also includes full manual as well as assembler and support applications.

Download from www.xilinx.com as XAPP213 [56].