Rapid Single Flux Quantum Very Large Scale Integration

Peter Alan Gross

Thesis presented in partial fulfillment of the requirements for **Master of Science in Engineering** at the University of Stellenbosch.

Advisor:

Prof. W.J. Perold

December 2002

Declaration,

I, the undersigned, hereby declare that the work in this thesis is my own original work, unless stated otherwise, and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature:

Date:

Abstract

Very Large Scale Integration (VLSI) of the Rapid Single Flux Quantum (RSFQ) superconducting logic family is researched. Insight into the design methodologies used for large-scale digital systems and related logistics are reviewed. A brief overview of basic RSFQ logic gates with in mind their application in a cell based layout scheme suited for RSFQ is given. A standard cell model is then proposed, incorporating these cells, on which, a library of low temperature superconducting (LTS) cells are laid out. Research is made into computer techniques for storing and manipulating large-scale circuit netlists. On this base, a method of technology mapping Boolean circuits to an RSFQ equivalent is achieved. Placements on-chip are made, optimized for minimum net length, routed and exported to a popular electronic mask format. Finally, the convergent technology fields of solid state cooling and high-temperature superconducting electronics (HTS) are investigated. This leads to a proposal for a low profile, low cost, HTS cryopackaging concept.

Opsomming

Grootskaalse integrasie (VLSI) van die *"Rapid Single Flux Quantum"* (RSFQ) supergeleidende familie van logiese hekke word uiteengesit. Insig in die ontwerpmetodes vir grootskaalse digitale stelsels en verwante aspekte word ondersoek. 'n Kort oorsig van basiese RSFQ logiese hekke word gegee, met hulle toepassing in 'n uitlegskema wat geskik is vir RSFQ. 'n Standaard sel model, wat bogenoemde selle insluit, word voorgestel en 'n selbiblioteek word uitgelê vir lae temperatuur supergeleidende bane. Ondersoek word ingestel na die manipulasie van die beskrywing van elektroniese bane en 'n manier om logiese Boolese baanbeskrywings om te skakel na fisiese RSFQ bane. Die fisiese plasing van selle word bespreek ten einde die verbindingslengte tussen selle te minimeer. Die finale uitleg word omgeskakel na 'n staandaard elektroniese formaat vir baanuitlegte. Die konvergerende tegnologievelde van "soliede toestand" verkoeling en hoë-temperatuur supergeleidende elektroniese bane word bespreek. Ten slotte word 'n nuwe tipe, lae profiel en lae koste kriogeniese verpakking voorgestel.

Acknowledgements

I would like to thank Prof. Perold for assisting in the research of this challenging topic. For what began as a seemingly insurmountable task ended up being something from which I've grown immeasurably.

Coenrad Fourie, thank you for your assistance on the finer points and your enthusiasm for RSFQ.

I thank the National Research Foundation for the assistance they gave by funding my studies.

Finally, to my family, thank you for the understanding and support you have given me to complete this thesis.

Contents

List of Figures

List of Tables

Nomenclature

Computer Aided Design	CAD	
Cell Based Integrated Circuit	CBIC	
Global Clock	CLK	
Input port of global clock on standard cell	CLK_{in}	
Output port of global clock on standard cell	CLK_{out}	
Coefficient of Performance	COP	
Electron charge	e	(1.602×10^{-19} C)
Electronic Design Automation	EDA	
Electronic Data Interchange Format	EDIF	
Characteristic Junction Frequency	f_c	
Finite State Machine	FSM	
Gigahertz	GHz	(1×10^{9})
Graphic User Interface	GUI	
Planck constant	h	(6.63×10^{-34} Js)
Hardware Description Language	HDL	
High Temperature Superconductor	HTS	
Critical Current	I_c	
Integrated Circuit	IC	
Josephson junction	JJ	
Josephson Transmission Line	JTL	
Low Temperature Superconductor	LTS	
Damping Resistance	R_n	

Rapid Single Flux Quantum	RSFQ	
Register Transfer Level	RTL	
Superconducting quantum interference device	SQUID	
Bias voltage	Vdd	
Input port of bias voltage on a standard cell	Vdd_{in}	
Output port of bias voltage on a standard cell	Vdd_{out}	
Very Large Scale Integration	VLSI	
Sinusoidal Global Clock Input 10mV	Vss	
Single Flux Quantum	Φ_0	(2.07×10^{-15} Wb)

Chapter 1

Introduction

Rapid Single Flux Quantum (RSFQ) is a superconducting family of logic that can realize circuits that switch in the sub-terahertz range and beyond [6][35]. Here, the prospect and implementation thereof is researched and discussed in a manner that is not limited to a single fabrication process, i.e. Hypres, but a broader range that could include *high temperature superconducting* (HTS) circuits as well.

VLSI stands for *very large scale integration* and refers to integrated circuits that contain more than 10^5 basic transistors or Josephson junctions. The type of circuits designed can be general-purpose integrated circuits like microprocessors, digital signal processors and memories.

A method for automating the layout of an RSFQ VLSI circuit is proposed and then developed, since purposefully written programmes performing this function, as of yet, are scarce. Before the methods and procedures for such a programme are covered, Chapter 1 reviews standard design techniques, methodologies and present logistic problems facing RSFQ. Chapter 2 develops from a raw library of RSFQ elements, a formal *standard logic cell* cast into which these elements are arranged. In Chapter 3, the netlist concept and data structures needed to manipulate large circuits in computer memory are developed from literature on the topic. Here, an RSFQ mapper is implemented to *technology map* a Boolean netlist to an RSFQ equivalent. Placement algorithms are then researched and tested in Chapter 4. Global and local routing procedures are developed in Chapter 5. Results of the algorithms employed are tested for three circuits and documented in Chapter 6. Then, in Chapter 7 new improvements in solid-state cooling are discussed with the potential to overcome the biggest hurdle to the implementation of RSFQ circuits.

The object of this thesis is to widen the pathway that leads to achieving the combination of RSFQ and VLSI. Many commercially available software packages exist that perform this task. It is hoped that the work undertaken here will contribute to this body of work at some later point in time.

1.1 Overview of VLSI design issues

In designing complex VLSI circuits from a given specification, primarily, the following are optimized for [1]:

- **Area** – Minimization of the circuit area is critical as it increases the *yield* of a circuit. The yield is the percentage of correct circuits on a chip. In addition, the ability to fit a circuit on fewer chips, leads to a more economical design.
- **Speed** – An increase in speed improves the attractiveness of a chip. Increasing the speed usually implies an increase in area due to parallelism. The design process should always consider the trade-off between *area* and *speed.*
- **Power Dissipation** – Although RSFQ logic elements dissipate considerably less power than transistor type elements, the level of power dissipation must be considered, especially since cooling at cryogenic temperatures becomes increasingly difficult.
- **Design time** – The design of an integrated circuit is an economic activity. A design satisfying a set of specifications should be available as soon as possible. CAD tools help to shorten the design time considerably as does the use of semi custom design.
- **Testability** – As a significant percentage of chips fabricated are expected to be defective, all of them have to be tested before being used in a product. It is important that a chip is easily testable as testing equipment is costly. This asks for the minimization of the time spent to test a single chip. Increasing the testability of a chip usually increases its area.

Two concepts that are helpful in the design of complex circuits are *hierarchy* and *abstraction* [1][8][26] and [3]. Hierarchy is used to describe a circuit over different levels of abstraction. Abstraction hides details of higher and lower levels (see figure 1.1). The use of abstraction allows for the reasoning out of a limited number of interacting parts at each level in a hierarchy. Each part is itself composed of interacting subparts at a lower level of abstraction. This decomposition continues until the basic building blocks (i.e. Josephson junctions) of a VLSI circuit are reached.

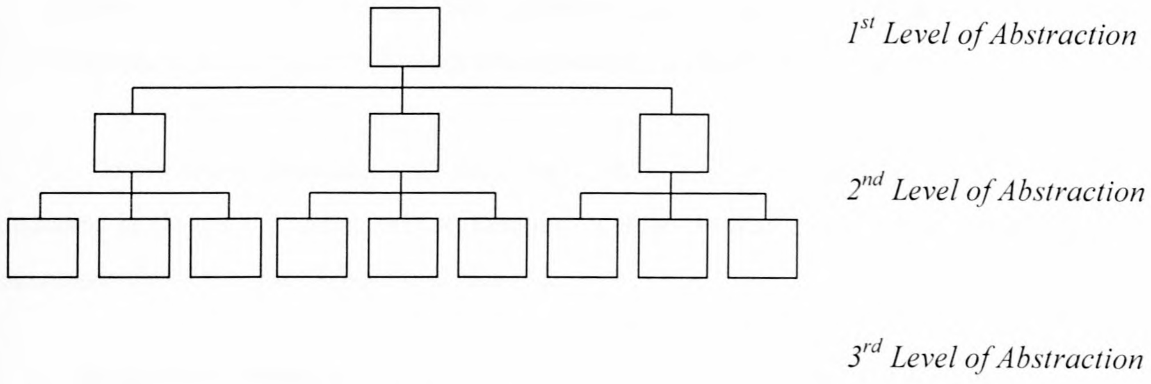


Figure 1.1. Hierarchy and Abstraction

1.2. The Design Domains

A hierarchy and abstraction model is insufficient to properly describe the VLSI design process. There is a consensus to distinguish three design domains, each with its own hierarchy. These being:

- **The behavioral domain** – This looks at a design from the perspective of a black box. A design with several junctions can be described by means of expressions in Boolean algebra or truth tables. At a higher, *register-transfer-level (RTL) [13]*, a circuit is seen as sequential logic consisting of memory elements (registers) and functions that compute the next state given the current memory state. The highest behavioral descriptions are algorithms that don't even refer to the hardware that will realize the computation described.

- **The structural domain** – The circuit is seen from a position of sub circuits. A description in this domain gives information on the sub circuits used and the ways in which they are interconnected. Each of these sub circuits has a description in the behavioral domain.

- **The physical domain** – A VLSI circuit always has to be realized on a chip, which is essentially two-dimensional. The physical domain gives information on how the subparts in the structural domain are located on a two-dimensional plane.

These three domains and their hierarchies can be visualized on a Y chart as depicted in figure 1.2. Each axis represents a design domain and the level of abstraction decreases from the outside to the center [1].

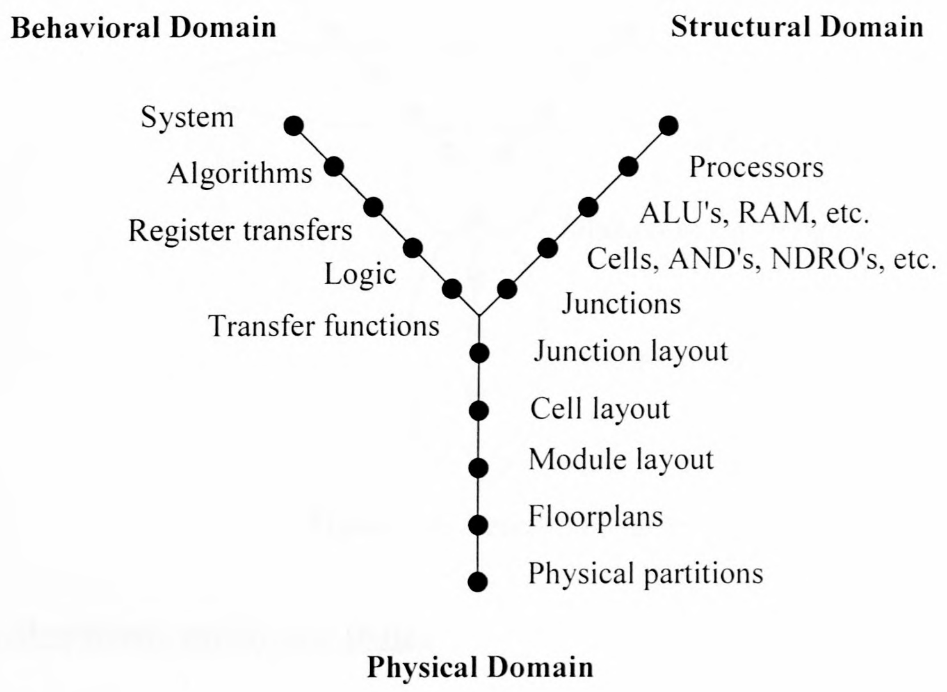


Figure 1.2. Y Chart adapted for Josephson junction type, large-scale circuits

1.3. Design Hierarchy

The Y-Chart can also be broken into a *design hierarchy* (see figure 1.3), allowing the chip designer to work on different levels of the same problem.

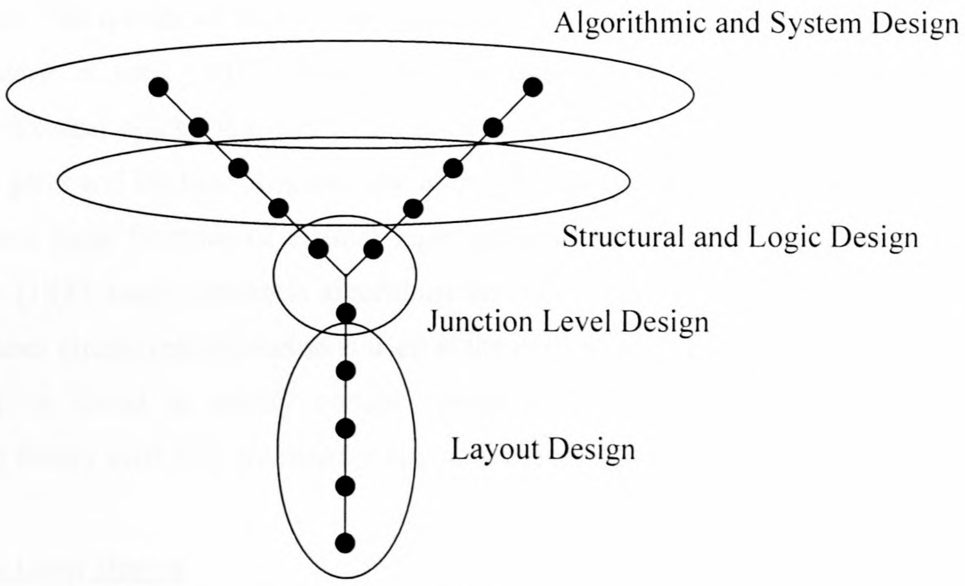


Figure 1.3. Design Hierarchy

1.3.1. Algorithmic and System Design

At the early stages of design, there is a need to formalize the specifications. The designer will work with the initial algorithm using a *hardware description language (HDL) [13],* allowing for a natural description of hardware [1][8]. Using a formal HDL to describe a circuit removes ambiguity from a specification that could be found in a natural language description, like English.

The HDL description can be used for design *simulation* and *synthesis* to generate an equivalent version of the circuit on a lower level using standard logic building blocks.

Other forms of capturing the behavioral description of a circuit also include *finite state machines (FSMs)*, which are useful in control-dominated applications [1]. However, most of these available tools convert the graphic information to equivalent HDL languages [1].

1.3.2. Structural and Logic Design

The complete design of a circuit can sometimes not be achieved by a high-level description alone. The quality of the circuits produced may be unacceptable. In this case, a *schematic editor* is used [1][8]. This *CAD* tool uses a graphic user interface to manipulate hierarchical blocks that can be expanded to lower-levels. The lowest levels consist of logic gates and latches. This tool also allows for simulation of the circuit.

In general, large libraries of distinct logic gates are available for realization of digital circuitry [1][8]. Logic synthesis algorithms do not deal with this library directly; instead, an abstract circuit representation is used at the early stages of synthesis. Once an estimate circuit is found to satisfy certain optimization constraints, the circuit is converted using library cells by a *technology-mapping* tool.

1.3.3. Junction-Level Design

Josephson junctions create the logic gates in RSFQ [6] and COSL [11] technology. At this level, to create these gates, a complete set of tools is needed, which are mostly simulation tools [1]. Simulations take place on the switch level (functional level), timing level (timing delay) and circuit level (nonlinear SPICE models).

For a full custom design it is important to be able to be able *extract* the circuit from its primarily 2D layout.

1.3.4. Layout Design

There are many different layout tools available to handle the editing of a multilayered, 2D circuit mask. In essence, the layout problem consists of sub blocks of a design that has a list of interconnections to be made [1]. These sub blocks are rectangles with terminals on their perimeter to which wires connect. The problem is to compose a layout of an entire integrated circuit.

Completed mask designs can be exported to popular formats like CIF, GDSII and DXF. Such mask files can then be sent to an integrated circuit foundry for fabrication.

1.4. VLSI Design Flow in Computer Aided Design Packages

The development of *integrated circuits (ICs)* is broken down into nine distinct steps [8], called the *design flow* as seen in figure 1.4. Each step involves specific algorithms and processes that are described below.

1. **Design Entry** – This relates to the *graphic user interface* (GUI) level of the CAD software tool package. Various tools are used that include text editors, schematic capturers, symbol libraries, simulation graphs, and message feedback windows among others.

2. **Logic Synthesis** – A netlist is produced from an *HDL* or *schematic entry* description defined in step 1. An HDL behavioral description is transformed into a structural equivalent, through a process of *synthesis.* A schematic entry is structural in nature already.

3. **System Partitioning** – For large systems where one netlist cannot fit onto a given IC area, the netlist is partitioned minimizing the number of interconnecting wires. There are many partitioning algorithms available. The Kernighan-Lin is one of the most popular with many derivatives - it is reviewed in chapter 4.

4. **Prelayout Simulation** – A useful tool for a circuit design is the ability to preliminary test a circuit netlist before implementing the lengthy layout process (steps 5 to 7). This saves time, considering that for circuits that are VLSI in nature optimum layout placements could take hours, if not days to complete.

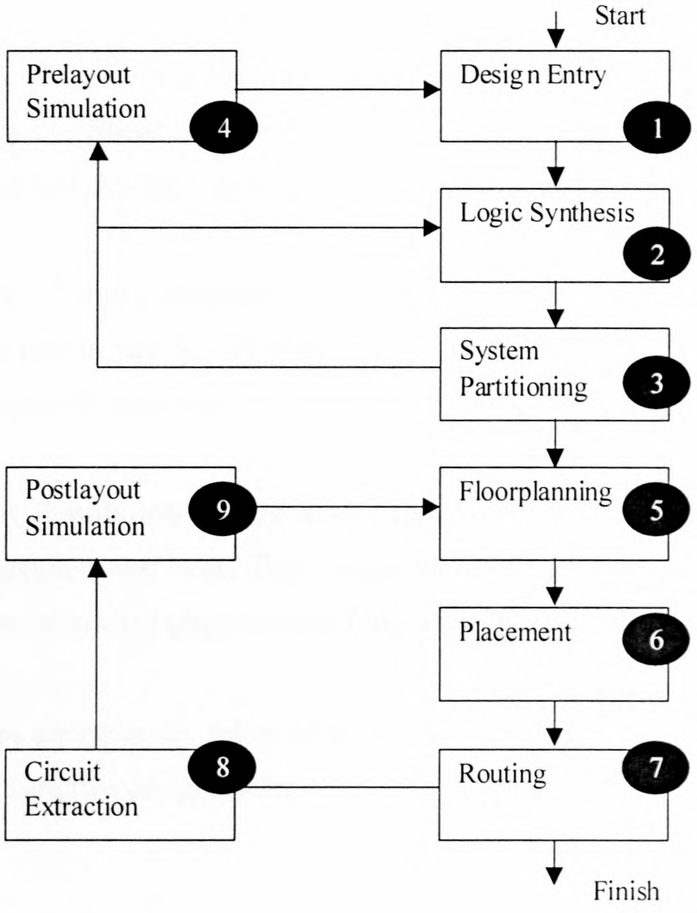


Figure 1.4. Application Specific Integrated Circuit Design Flow [8].

5. **Floorplanning** – The designed primary blocks of a large circuit contain logic gates in the order of thousands that are arranged according to requirements. CAD tools have been developed to automate the step and generally attempt to minimize interconnecting wire lengths or congestion.

6. **Placement** – The exact placement of cells in the primary blocks are decided here. There is a host of optimization techniques available that either attempt to minimize interconnecting wire length or area and maximize circuit speed. In this thesis, we implement simulated annealing and genetic algorithms to achieve these goals.

7. **Routing** – Connections are made between placed cells from step 6. Once a preliminary placement is made, the layout is first globally, then locally or detailed routed. The two sub-steps employ different algorithms.

8. **Extraction** – Circuit parameters are extracted from physical layout and sent to a simulation tool in step 9. This step is useful to the circuit designer in that it helps to verify whether placement will work according to initial specification.

9. **Postlayout Simulation** – This final step involves performing simulations with various waveform test beds. These tested waveforms can be compared with those in step 4 as to how the physical manifestation of a circuit changes its timing.

These steps are generally followed as per the flow diagram in figure 1.4. It should be noted that there are two design cycles.

1.5. Economics of Superconducting Electronics

An important consideration is the present cost penalty for using cryoelectronics as opposed to room-temperate alternatives in implementing an electronic circuit. The rapid adoption of superconductors is presently hampered by the high price of the cooling units required to reach cryogenic temperatures. A roadmap for the superconducting electronics [9] has been established similar to the semiconductor equivalent [10], both highlighting the same issue.

The problem is made apparent when a cooling unit for *low temperature superconductors (LTS)* presently would cost in the order of \$30,000 [30]; for *high temperature superconductors (HTS)* this would be in the region of \$10,000 [31].

Presently, LTS Nb/AlO$_x$/Nb [16] is the most advanced RSFQ technology to date with integration scales approaching the sub-micron region [32]. HTS on a sub-micron scale is thought to be something of the future [33]. However, the cost of fabricating a VLSI LTS or HTS IC is considered negligible in comparison to the peripheral cooling and packaging needed.

This thesis addresses this problem in a later chapter by highlighting advancements made in the field of solid state cooling [23][27], particularly thermoelectrics. By making use of nano-scale fabrication, a new breed of solid state cooling is made available based on concepts that have been known since 1821.

So as not to lose sight of a wider scope of emerging technologies that compete with RSFQ, the reader is urged to refer to [10].

Chapter 2

Library of Superconducting Logic Components

A library of LTS RSFQ logic cells are presented that have the ability to produce digital switches that operate at sub-terahertz speeds [6] [7]. The object of this chapter is to provide an overview of the basic nature and ability of RSFQ circuits without too much detail. Individual circuit SPICE models are available in Appendix B [4].

A differentiation is made between a *basic logic circuit* and a *standard logic cell* with respect to RSFQ. A *basic logic circuit* is an implementation of building blocks required for the general concept of RSFQ; *standard logic cells,* however, develop from the former to create a structured means of arranging such circuits on a large scale.

For a deeper understanding of the physical principles and design issues of RSFQ circuits, the reader is encouraged to refer to [6], [2], [7], [11] and [16].

2.1. RSFQ Basic Logic Circuits

The circuits below are optimized for high yield and are tested using Monte Carlo Analysis [4]. The basic principle of RSFQ was made widely known by Likharev et al. [6] and tested to have operating speeds as high as 770GHz [7]. The pulses used in these circuits adhere to the integral below.

2.2. Standard Logic Cell Template

For an organized method of arranging the physical layout of the above logic cells, a *standard cell [8]* template is used. This serves as a mold wherein the physical logic function of a superconductive RSFQ cell is arranged to convention.

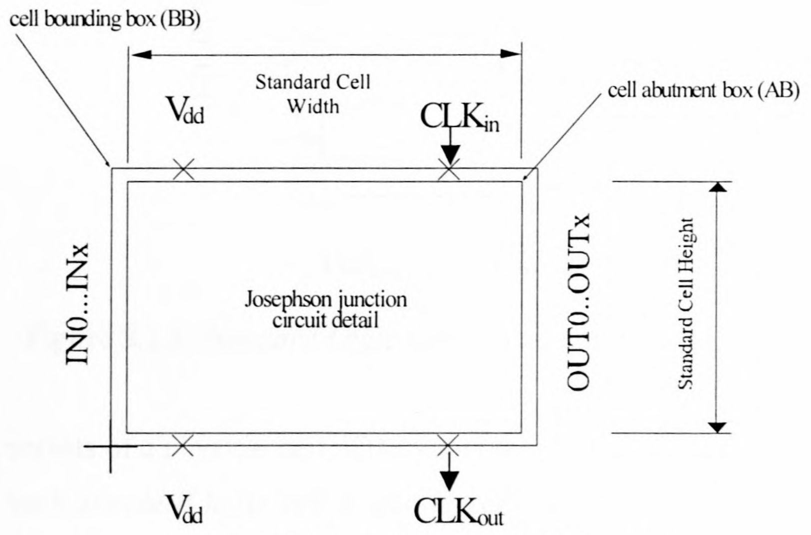


Figure 2.2.1. Standard Cell Layout

Inputs enter on the left or west orientation, while outputs exit on the right or east orientation. *Special ports* are considered the supply voltage V_{dd} and global pulse clock, *CLK*. The *Standard Cell Width* is a fixed quantity throughout a physical library, and the *Standard Cell Height* varies according to individual cell layouts.

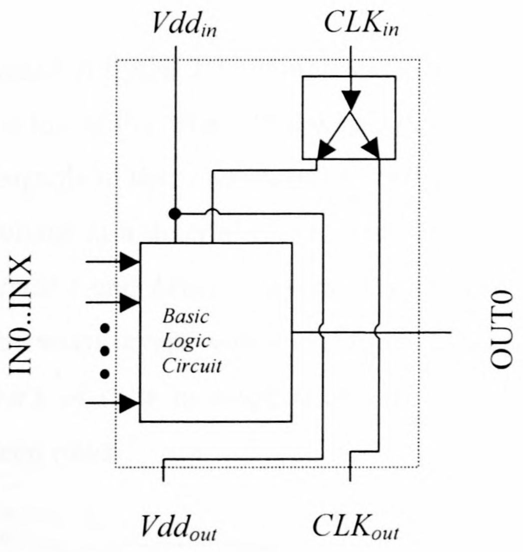


Figure 2.2.2. *Standard Logic Cell* containing *Basic Logic Circuit*

The contents of a physical cell will contain the circuit arrangement as seen in figure 2.2.2. For each *standard logic cell* a splitter cell must be included to repeat the clock signal CLK_{in} to the special output port CLK_{out}. The incoming Vdd_{in} supplies the basic logic circuits bias voltage and couples to the special output port Vdd_{out}. In this fashion, it can be seen that these standard logic cells can be stacked one on top of the other.

2.3. An RSFQ Cell Based Integrated Circuit (CBIC)

The process proposed in figure 2.3., comprises rows of the above defined standard logic cells stacked one on top of the other. At the top, *Row End Cells* or *terminator cells* supply the V_{dd} and *CLK* signals to the rows. A *power cell* may be included to increase the strength of the supply voltage at a determined position in the rows. Routing of the cells will take place on the *Metal 1* and *Metal 2* layers of a multi-layered fabrication process with connections made between layers with a *via.* A *spacer cell* of arbitrary height may be used to adjust positions of cells in a specified row. A *feedthrough cell* allows for spacing for routing between rows.

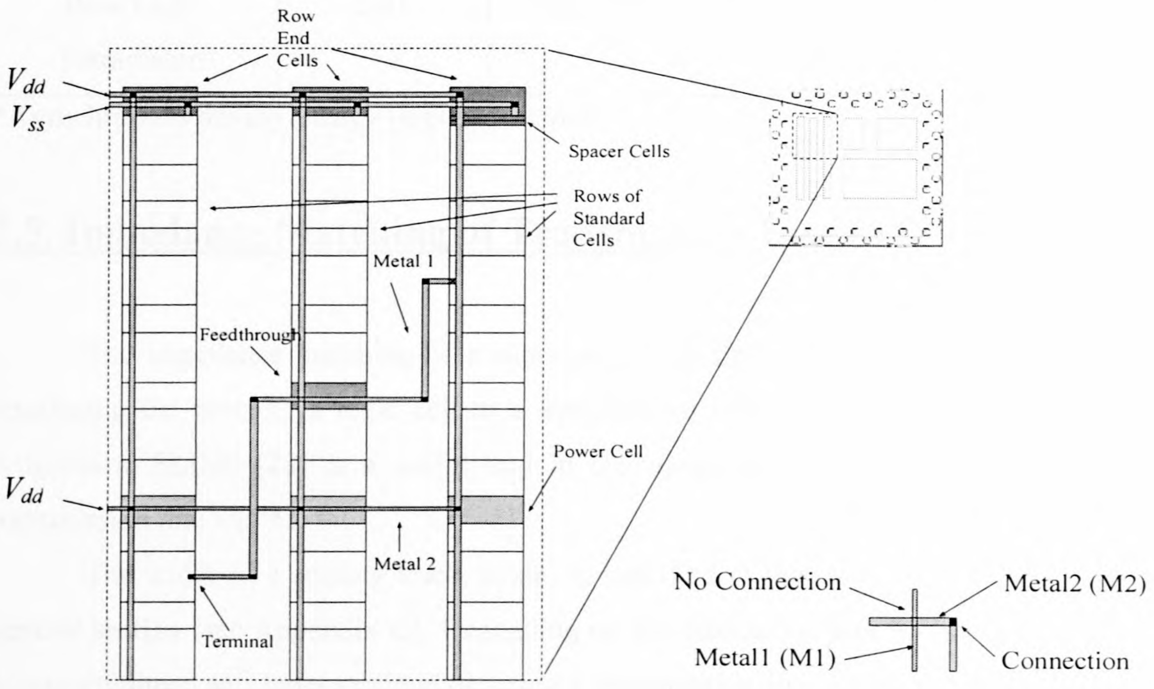


Figure 2.3. Cell based layout of standard logic cells.

Later chapters will be concerned with the automation of the arrangement of such a layout scheme. In conventional semiconductor layout strategies [8][1], the orientation of the standard logic cells is usually made in the horizontal plane. For the sake of consistency with a superconducting programmable gate array layout method [5], a vertical orientation was chosen.

Table 2.5. Parameters for Cells

Standard Logic Cell	Static Power Consumption [μW]	Actual Dimensions [μm]	Minimum Layout Area [μm^2]	DC bias [mV]	Latency (Clock to Output) [ps]
RSFQ AND-gate	7.26	215 x 193	41 495	2.6	35
RSFQ OR-gate	2.31	215 x 170	36 550	2.6	5.5
RSFQ NOT-gate	2.16	215 x 100	21 500	2.6	21
RSFQ XOR-gate	2.00	215 x 100	21 500	2.6	12
RSFQ Splitter-gate	3.10	215 x 70	15 050	2.6	10*
Row End/ Terminator	2.61	215 x 70	15 050	2.6	22*

*Asynchronous device latency (input-to-output)

2.5. Impedance Matching of Transmission Lines

The impedance matching of a superconducting line should be considered when interfacing the ports of a logic cell to a transmission line. The inductance calculation programme SLINE [28] is a useful tool in calculating, particularly, superconducting transmission line widths.

The width of a routing track, again, is specified in the *technology mapping file* general section (see Appendix C). Depending on the fabrication process being used, the design engineer will have to simulate using a transmission line model to find out what width of routing will be required to match ports of the standard cells employed.

Chapter 3

Netlists and Data Structures

To manually keep track of thousands of logic cells at a time is impossible for a design engineer who may have only months to complete a design. By using a computer to layout a VLSI circuit, tasks can be automated, tracked and simulated in a substantially reduced amount of time. In this chapter, a model for representing an electronic circuit in the memory of a computer is developed. In a later section of this chapter, a *technology mapping* step from Boolean logic to equivalent RSFQ is made.

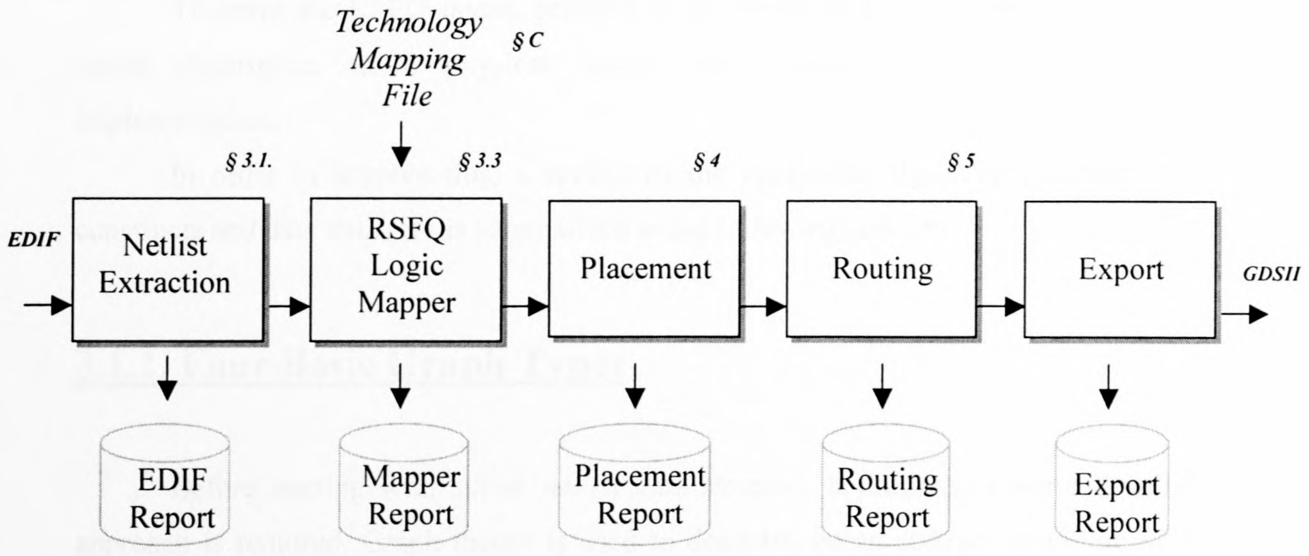


Figure 3.1 System overview

To integrate the previously developed layout scheme with existing electronic design methodologies the system below is proposed (see figure 3.1). This strategy accepts an *Electronic Data Interchange Format* ASCII file as input and processes it through what is termed the *layout process*, to export a physical mask to the GDSII format file. There are five distinct steps involved, namely: *Netlist Extraction, RSFQ Logic Mapper, Placement, Routing* and *Export.* Each of these issues is dealt with in various programme units written in Borland Delphi 6 [36].

3.1. Netlist Extraction

A means to store a structural circuit description and manipulate it using computer algorithms is the cornerstone to an EDA tool. This problem is often dealt with in industry and is well documented[1][8][26].

In tackling the RSFQ layout problem, the fact that entire design methodologies exist for conventional semiconductor type logic was considered. Instead of "reinventing the wheel" much effort was spent on implementing only what was needed. Adherence to industry standards was taken by support of the *Electronic Data Interchange Format EDIF* format [14].

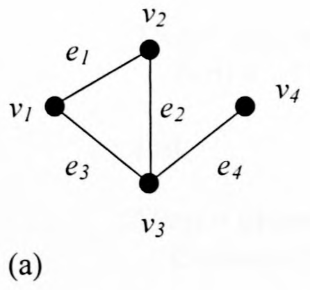
To solve the RSFQ layout problem is to transform a semiconductor logic type netlist description to a physical layout that comprises an equivalent RSFQ implementation.

In order to achieve this, a review of the applicable algorithms, mathematical constructs and data structure is summarized in the following sections.

3.1.1. Four Basic Graph Types

Before starting with actual *netlist* data structure definitions, a more simplistic approach is required. Graph theory is used to describe, on an abstract level, the *netlist* concept.

Graphs describe a set of objects and how they are connected to each other [1]. A *graph* is represented by a set of vertexes and edges $G(V, E)$. Figure 3.1.1. (a) shows an example graph where V = $\{v_1, v_2, v_3, v_4\}$ and E = $\{e_1, e_2, e_3, e_4\}$. A *directed graph* is sometimes useful in the description of a graph where in this instance, edges e_k have an added directional component to describe an endpoint v_k, see figure 3.1.1 (b).



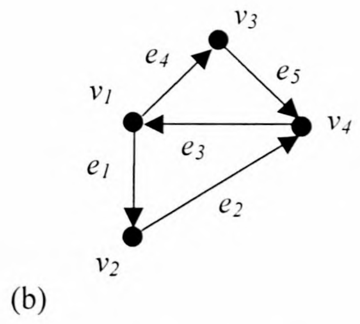


Figure 3.1.1. (a) Example Graph (b) Directed Graph

Graphs can be extended to have edges carry a weight term, w_k. This is analogous to a road map problem where vertexes represent cities; edges the connecting roads and the weight the length of these roads. This type of graph is called an *edge-weighted* graph. In some instances, it may be necessary to associate a weight with a vertex resulting in a *vertex-weighted* graph.

3.2. Data Structures to Represent Cell-Port-Net Graphs

The choice of a suitable data structure in the implementation of a graph algorithm is important in that it may directly affect the computational effort required solving a given problem. In the case of the RSFQ layout problem, *vertexes* become *cells* and *edges* become *nets*. Added to the basic definition of a cell is the need for ports. In a Pascal type expression, we describe these basic object structures in Figure 3.2.1

```
type PCell = ^TCell;
     PNet = ^TNet;
     PPort = ^TPort;

     TCell = object(TDynamicUnit)
         Ports : PPort;
         ...
     end;

     TNet = object(TDynamicUnit)
         PortPtr : TPortPtrDynamicList;
         ...
     end;

     TPort = object(TDynamicUnit)
         ConnectedNet : PNet;
         ...
     end;
```

Figure 3.2.1. Data structure for the representation of a Cell-Port-Net Graph

A netlist object is then used to hold the Cell-Port-Net structures in figure 3.2.2. A netlist object can represent either a *master cell* or an actual *netlist*. A master cell is a fundamental building block, such as an AND or OR gate. A netlist is a collection of cells, nets and *hierarchical* ports.

```
type PNetlist = ^TNetlist;
     TNetlist = object(UDynamicUnit)
        Cells : PCellsDynamicList;
        Nets : PNetDynamicList;
        Ports : PPortDynamicList;
     end;
```

Figure 3.2.2. Data Structure for the representation of a Netlist structure

The data structure symbols used to represent these objects are summarized in figure 3.2.3. The dots represent a pointer reference to the respective netlist objects. A cross through a dot implies a null reference, or generally the end of a list.

A dynamic linked list unit was developed (see Appendix D.13) to handle the memory allocation of these netlist, and other, objects. This was chosen over simply allocating arrays of netlist objects, due to the need for 10^5 or more netlist objects to be allocated in memory at a single time. Linked lists are generally slow in the time it takes to process a list of N nodes. The order of complexity is reduced by the introduction of *hashed linked lists* (see Appendix D.13.3).

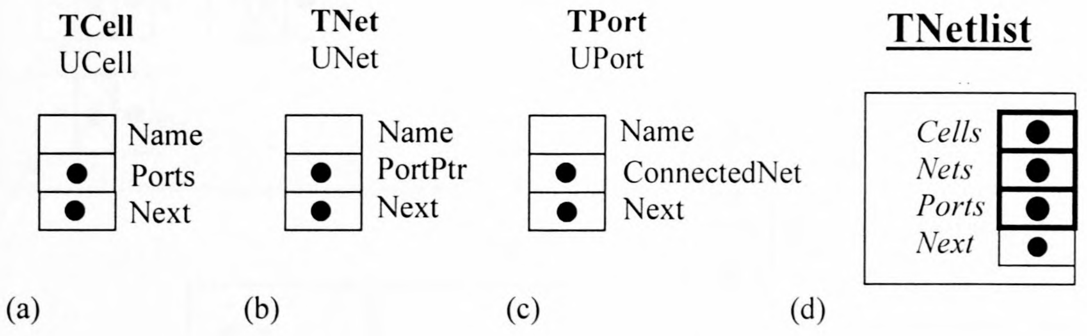


Figure 3.2.3. The visualization of the Cell-Port-Net and Netlist structures

A basic example circuit, named ABasicNetlist, can be seen in figure 3.2.4, containing three arbitrary cells, 7 nets, and 5 hierarchical ports. In an EDA design environment, this circuit in turn could be represented by a "black box" with five ports which exemplifies the prior mentioned concept of *hierarchy* and *abstraction*.

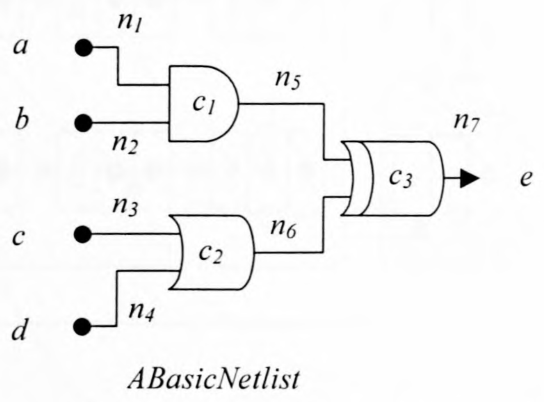


Figure 3.2.4. An Example Circuit (7 Nets, 5 Ports and 3 Cells)

The way in which ABasicNetlist is represented in internal memory of a computer is graphically depicted in figure 3.2.5.

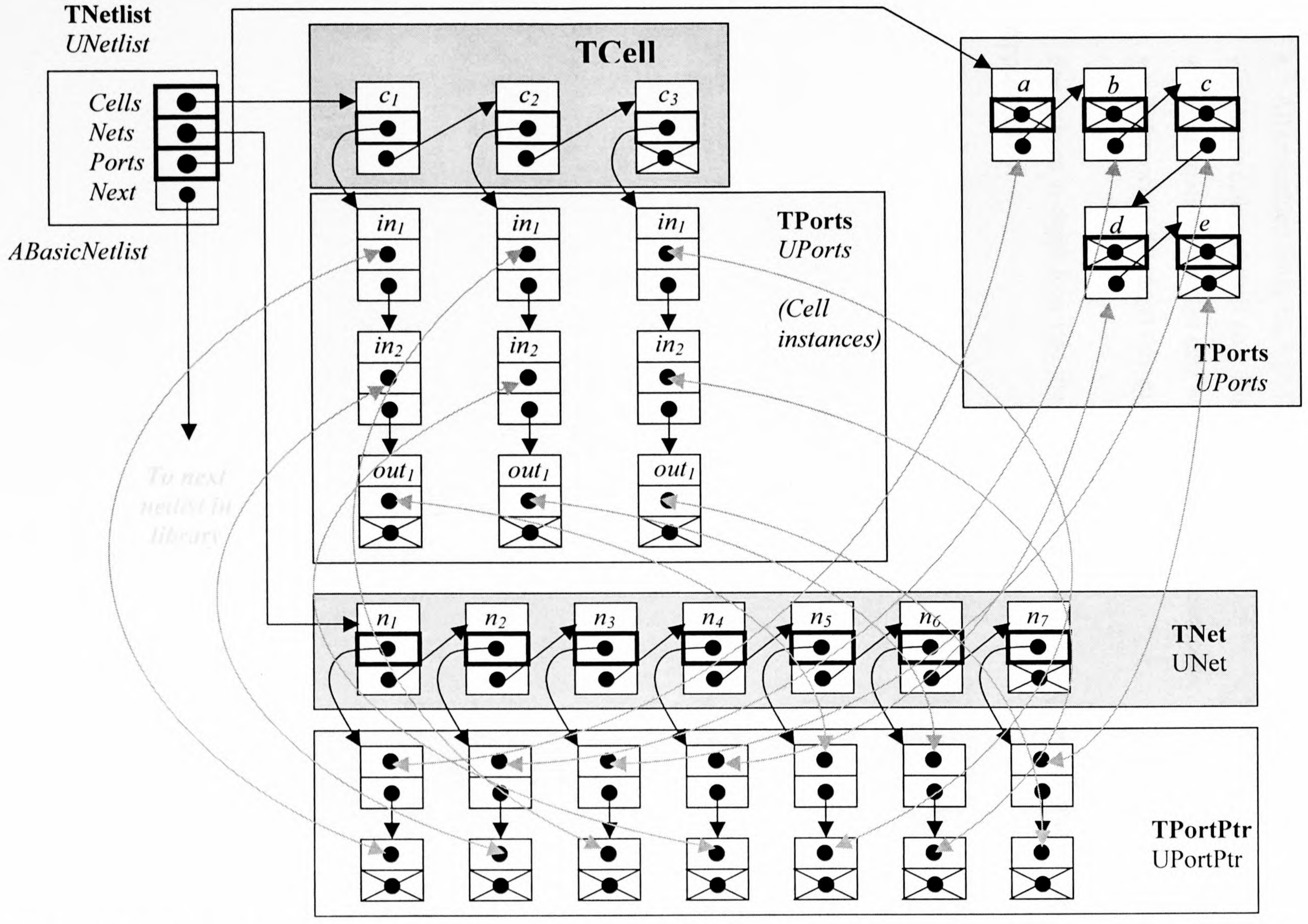


Figure 3.2.5. Data structure representation of circuit in figure 3.2.4. Single direction arrow lines imply a reference to a netlist object; dual direction arrow lines imply that both objects are aware of each other's reference.

The netlist is built by extracting the following data

- All embedded libraries, with their
- Master cells and their port declarations
- Cells containing netlist data with reference to master cells
- Reference to the target design cell

For a more in-depth look into the procedures used to extract the netlist, please refer to Appendix D.4.

3.4. RSFQ Logic Mapping

The extracted netlist is now mapped to an equivalent RSFQ netlist. In essence, this process transforms a semiconductor type Boolean logic circuit structure into an RSFQ equivalent. The mapping process is three fold: map Boolean cells to RSFQ equivalents, remove redundant cells and remove fan-out nets.

3.4.1. Map Cells to Technology

In the *technology mapping* file (see Appendix C), one is allowed to configure exact details of how the mapping process performs a one-to-one mapping of logic gates (see figure 3.4.1).

Physical dimensions, port names, port positions and references to the mask structures in a mask library are also mapped.

Special allowance is made for *special ports*, namely Vdd_{in}, Vdd_{out}, CLK_{in} and CLK_{out} located at the top and bottom edges of a standard cell, for which no equivalent exists in a Boolean logic gate.

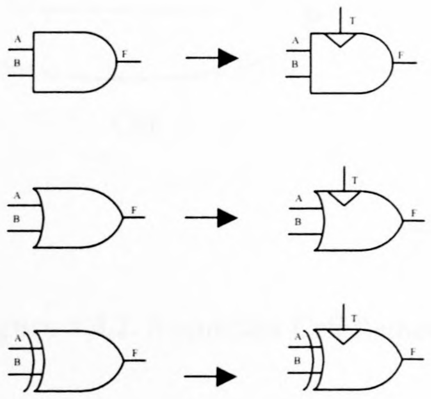


Figure 3.4.1. One-to-one mapping

3.4.2. Remove redundant cells

Certain cells in a netlist are redundant. Examples include single input AND and OR gates as well as DELAY cells (see figure 3.4.2). In the mapping process, these are cut from the netlist as they are likely residues from a synthesis process but an RSFQ equivalent cell would be meaningless.

The netlist is scanned for cells that meet this description and they are removed by the following algorithm

1. Locate redundant cell.
2. Add each outgoing net connection to the incoming net.
3. Remove references from both the incoming and outgoing nets to the redundant cell.
4. Remove the redundant cell belonging to the netlist.

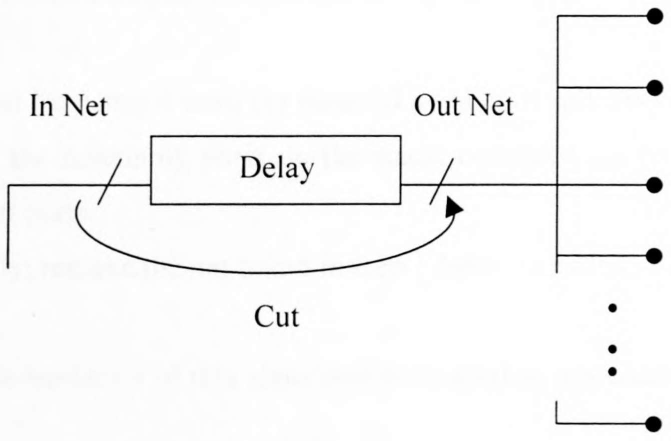


Figure 3.4.2. Redundant Cell Removal

The exact implementation of this algorithm is documented in Appendix D.5.1.

3.4.3. Remove fan-outs

Fan-outs have to be replaced by RSFQ splitter cell networks (see figure 3.4.3). For every splitter added to a fan-out network a delay will be brought into the timing of the netlist. It is advisable to carefully consider the design of a circuit where a large fan-out is required. The algorithm used to implement the creation of an RSFQ equivalent to a fan-out network is:

1. Locate a net that has a fan-out.
2. Locate the fan-out input port.
3. Create a LILO stack with the first entry containing the port found in step 2.
4. Create a new instance of a splitter cell and add it to the netlist.
5. Add a new net that connects the first port on the stack to the input of the new splitter and add it to the netlist.
6. Pop the port used in step 5.
7. Add the two output ports of the newly created splitter to the end of the port stack.
8. Repeat from step 4 until the required number of splitters have been created.
9. With the remaining ports on the stack, connect them to the original fan-out output ports.
10. Finally, remove the net found in step 1 from the netlist.

The implementation of this algorithm is covered in Appendix D.5.2. and is tested in Chapter 6.2.

3.3.4. Other mapping issues

Tristate® buffers and bus networks were considered in the mapping process. However, because an equivalent in RSFQ could not easily be found, an automated process for this type of mapping was omitted. A design engineer would still have the ability to layout *combinational* and s*equential logic* circuits, though.

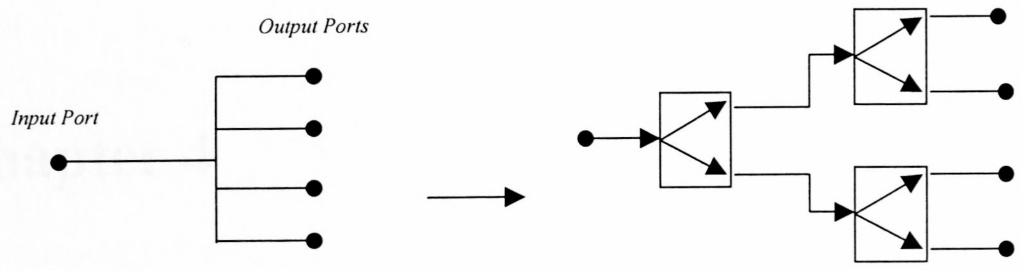


Figure 3.4.3. Mapping of a fan-out net to an equivalent splitter network

Now that an RSFQ netlist can be represented in the memory of a computer in a structured form, placement optimization and routing can be performed.

Chapter 4

Placement

The physical positioning of a large number of *standard logic cells* are optimized. This is an important step in solving the VLSI problem. Many optimization techniques exist [1][8][26]. Three are presented here. Two mayor categories of placement algorithms exist: *constructive* and *iterative* placement.

A constructive placement uses a *min-cut algorithm* or eigenvalue method. The min-cut uses successive partitioning of a netlist and then places the partitioned *bins* accordingly. Iterative placement involves an incremental improvement of an initial random arrangement and attempt to achieve the following goals [8]:

1. Minimize the total estimated interconnect length
2. Meet the timing requirements for critical nets
3. Minimize the interconnect congestion

4.1. Min-Cut Partitioning

Using the min-cut partitioning algorithm, a circuit is split into two sub circuits of near equal size while minimizing the number of nets that are connected to both sub circuits, The two sub circuits will each be placed in separate halves of the layout.

Depending on the cut being horizontal or vertical, the sub circuits are placed in upper and lower or left and right halves. It is assumed that, due to the fact that the number of nets crossing from one half to the other has been minimized, the number of *long wires* crossing from the halves have been reduced as well. We call this type of partitioning *bipartitioning* [1].

The *bipartitioning* is recursively applied until the partitioned sub circuits contain a specified minimum number of cells. Figure 4.1 has a three-stage bipartitioning of a given netlist.

The min-cut partitioning has two important tasks: to partition a graph and assign the partitions to their relative layout positions. Another factor to this algorithm is the ability to work with parts that already have a fixed position like inputs and outputs on the periphery of a chip.

The min-cut partitioning is a top-down approach where an entire circuit is decomposed, while another placement strategy, known as *clustering*, is a bottom-up approach.

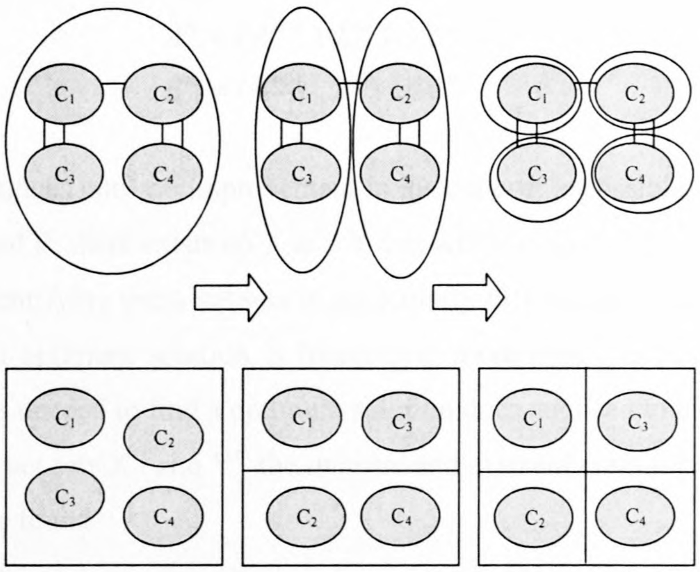


Figure 4.1. Partitioning example

4.1.1. The Kernighan-Lin Partitioning Algorithm

Many versions of the partitioning problem exist and there are many algorithms for each version, such as the following by Kernighan and Lin in 1970 [1][26]. The model takes a undirected graph *G(V, E)* which has 2n vertices (|V|=2n). Each edge (a, b) $\in E$ has a weight γ_{ab}. The problem is to find two sets A and B, with $A \cup B = V$, $A \cap B = \emptyset$ and $|A| = |B| = n$, that minimizes the cut cost: (4.1)

$$\sum_{(a,b)\in A\times B} \gamma_{ab}$$

The principle of the algorithm is to start with an initial partition consisting of sets A^0 and B^0, which will in general not have a minimum cost. Iteratively, subsets of both sets are isolated and interchanged. In the mth iteration, we will denote X^m to be the set isolated from A^{m-1} and Y^m to be the set isolated from B^{m-1}. The new sets A^m and B^m are then obtained as follows: (4.2)

$$\begin{aligned} A^m &= (A^{m-1} \setminus X^m) \cup Y^m \\ B^m &= (B^{m-1} \setminus Y^m) \cup X^m \end{aligned}$$

This continues until no improvement in the cut cost is possible. For a non-optimal partition set A and B, there exists no X and Y that will lead to an optimum solution in one *pass*, or step. Identifying these subsets is particularly difficult so subsets X^m and Y^m are found so that an optimum solution is found over more than one pass. Reportedly, the number of passes needed to find a optimum solution does not need to be more than 4.

To construct sets X^m and Y^m the *internal* and *external* costs for vertices in the sets A^{m-1} and B^{m-1} are found.

$$E_a = \sum_{y\in B^{m-1}} \gamma_{ay}, a \in A^{m-1} \qquad (4.3)$$

The external cost for vertex a ϵ A^{m-1} is a measure of the *pull* that the vertex experiences from the vertices in B^{m-1}. Similarly, the external cost E_b as well as the internal costs I_a and I_b can be defined:

$$E_b = \sum_{x \in A^{m-1}} \gamma_{bx}, b \in B^{m-1} \tag{4.4}$$

$$I_a = \sum_{x \in A^{m-1}} \gamma_{ax}, a \in A^{m-1} \tag{4.5}$$

$$I_b = \sum_{y \in B^{m-1}} \gamma_{by}, b \in B^{m-1} \tag{4.6}$$

The difference between internal and external costs gives an indication about the desirability to move the vertex. A positive value indicates a move to the opposite set, while a negative shows a preference to leave the vertex in the current set. The differences for the vertices in both the sets are given by D_a and D_b:

$$\begin{aligned} D_a &= E_a - I_a, a \in A^{m-1} \\ D_b &= E_b - I_b, b \in B^{m-1} \end{aligned} \tag{4.7}$$

The gain in cut cost, Δ, resulting from an interchange can then be expressed as:

$$\Delta = D_a + D_b - 2\gamma_{ab} \tag{4.8}$$

This placement technique was reviewed and implemented. However, it proved to be ineffective and cumbersome due to the nature in which cells would have to be recursively sub partitioned while still making allowance for interconnecting partitions.

4.2. Iterative Placement Improvements

The basis of the iterative placement improvement is a *grid* (see figure 4.2) that holds a preliminary position of an arrangement of cells to be optimized. By using a grid of pointer references to cells in a netlist, the algorithms employed can keep track of arrangements. Upon finding an optimal arrangement during a run, the programme can save the grid state for later recall.

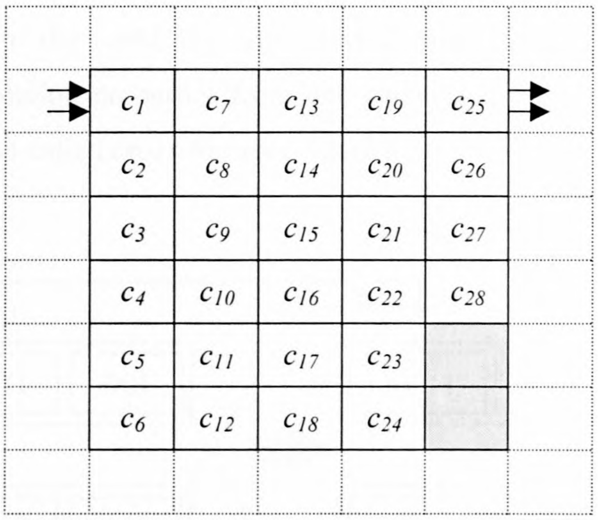


Figure 4.2. Initial Placement of an example 28 cell netlist with 4 Ports (2 Inputs and 2 Outputs).

Two methods for optimizing *total net length* are considered, namely genetic and simulation annealing. The total net length is calculated for all nets *E,* input port *i* and output *j*, as

$$\sum_{a \in E} (x_{ai} - x_{aj}) + (y_{ai} - y_{aj}) \qquad (4.9)$$

Only two calculations are needed per net since an RSFQ net can legally only have two connection points. This is based on the *Manhattan distance* algorithm [1][8].

4.2.1. Genetic Optimization

An optimization technique is introduced, by use of *genetic algorithms*. The method works with a fully specified solution *f*, in a set of feasible solutions *F* [1]. Simultaneously the algorithm keeps track of a set *P* of feasible solutions called the *population*. Using an iterative search process, the current population $P^{(k)}$ is replaced by the next one $P^{(k+1)}$ using a procedure that is characteristic for genetic algorithms.

A feasible solution $f^{(k+1)} \in P^{(k+1)}$ is generated from two feasible solutions $f^{(k)}$ and $g^{(k)}$, called the *parents* of the *child*, that are selected from $P^{(k)}$. $f^{(k+1)}$ is generated by *inheriting* parts of its solution properties from one parent and the other from a second parent under the operation called *crossover* (see figure 4.2.1).

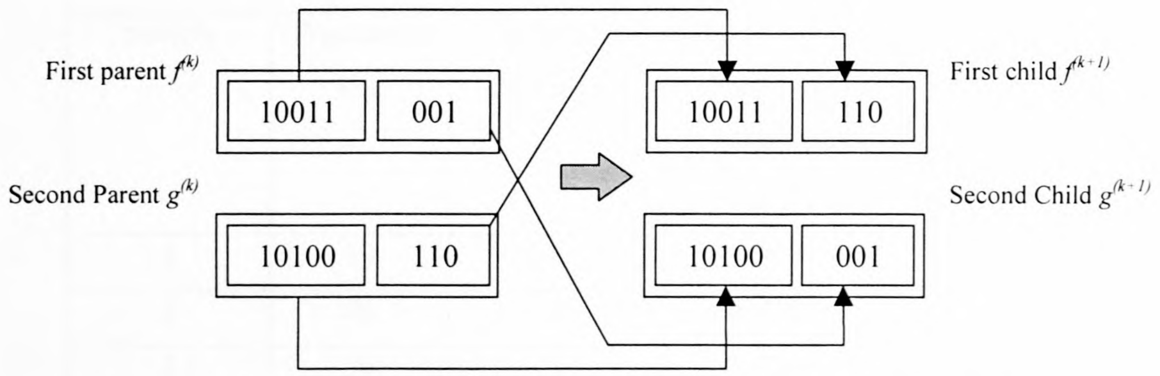


Figure 4.2.1. The Genetic Algorithm Generation Principle - Crossover

The operation assumes that a solution $f^{(k)}$ is represented by a string of values of a set length *n*, known as a *chromosome*. The *chromosomes* of a solution $f^{(k)}$ represent positions of cells in a solution.

The algorithm starts with a set of randomly generated solutions in the *population*. Parents are then selected to *crossover* based on a *fitness test*, which in this case is the total interconnecting net length. A *mutation* can be introduced to improve results that is realized by randomly swapping a determined percentage of *chromosomes* in a solution $f^{(k)}$.

This process is repeated until the *new population* has reached maximum size. Upon this, the old population bin is destroyed and replaced by the new offspring. Again, this process is repeated until the allowed number of generations is found. [1] [26].

The genetic algorithm was tested using a moderate sized finite state machine circuit, FSM. Numerous tests where conducted to gauge the performance of the optimization technique. The results are summarized in table 4.2.1 and figure 4.2.2 and 4.2.3. A mutation factor of 1% was chosen to improve results.

Table 4.2.1. Genetic Algorithm Test Cases (Mutation Factor = 1%)

FSM (cells=*67*, *nets =105*)			Results	
Sample	Population Size	Mutation Factor [%]	Best Result Total Net Length [μm]	Time [s]
1	10	1	100830	11
2	20	1	94122	18
3	50	1	89729	42
4	100	1	85057	85
5	200	1	78543	199
6	300	1	76132	322
7	400	1	77593	593
8	500	1	72856	868

Figure 4.2.3 plots the progress of the genetic optimization over the generations. In general, this technique is random since it starts from a randomly generated population pool. A larger population size leads to a greater optimization.

For the actual implemented genetic algorithm, refer to appendix D.6.1.

4.2.2. Simulated Annealing

Simulated Annealing takes an existing solution and then makes successive changes in a series of random moves[1][8][26]. An *energy function* determines whether a trial move is accepted or rejected. Minimums in the energy function correspond to a possible solution and the best solution is the *global minimum*.

Several *interchange* or *iterative exchange* methods exist and determine the random moves in a solution, namely [8]:

1. Pairwise interchange
2. Force-directed interchange
3. Force-directed relaxation
4. Force-directed pairwise relaxation

Normally only pairs of cells are interchanged by picking a source cell to be swapped with a destination cell. The *pairwise-interchange algorithm* follows these steps:

1. Select the source logic cell at random
2. Try all other logic cells in turn as the destination
3. Use a measurement method to decide whether to accept an interchange
4. Repeat process from 1.

The *neighborhood exchange algorithm* (see figure 4.2.4) is a modification of the pairwise interchange algorithm where the destination cells are in a neighborhood of cells the distance ε away of the source cell.

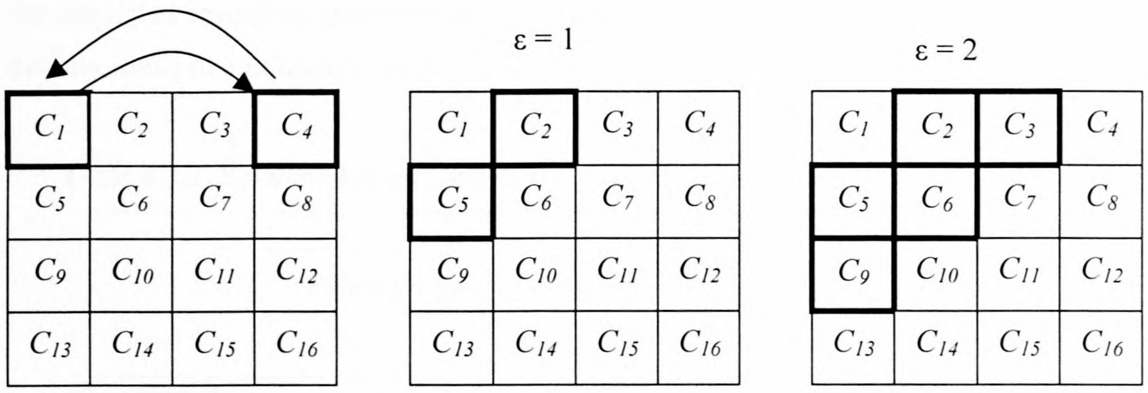

ε = 1
ε = 2
C_1 C_2 C_3 C_4
C_5 C_6 C_7 C_8
C_9 C_{10} C_{11} C_{12}
C_{13} C_{14} C_{15} C_{16}

Figure 4.2.4. (a) Pairwise Interchange (b) 1-Neighborhood of module 1 (c) 2-Neighborhood of module 2.

The *pairwise-interchange algorithm* with the *neighbourhood exchange algorithm* was chosen as the simulated annealing algorithm. Its implementation is found in appendix D.6.2.

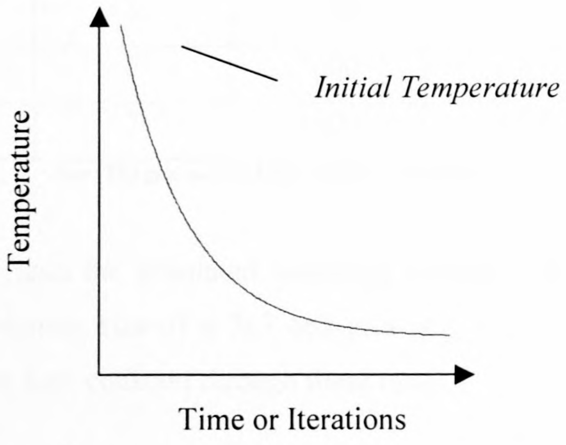


Figure 4.2.5. Cooling Schedule for Simulated Annealing Algorithm

The *temperature* value of the system is used (see figure 4.2.5) to moderate the likeliness that a swap will take place. A *cooling schedule* sets the speed in which the system cools. The temperature of the next iteration is based on the current temperature multiplied by the cooling schedule. As the *temperature* of the system approaches zero, the simulated annealing algorithm becomes *greedy* in that it only makes interchanges that directly result in a minimization of net length.

Table 4.2.2. Simulated Annealing Test Cases varied with Neighbourhood Size (ε)

FSM (cells=*67*, *nets =105*)				Results
Sample	ε	Initial Temperatue	Cooling Schedule	Best Result Total Net Length [μm]
1	1	1000	0.99	99793
2	2	1000	0.99	96548
3	3	1000	0.99	91226
4	4	1000	0.99	87401
5	5	1000	0.99	93438
6	6	1000	0.99	91903
7	7	1000	0.99	90035

Note: The grid on which the optimisation is taking place is 7x7.

Table 4.2.2. tests the simulated annealing algorithm for different neighbourhood sizes where the optimum size of a 7x7 cell grid is ε = 4. The initial temperature and cooling schedule are kept constant through these tests.

Chapter 5

Global and Local Routing

After the placement phase, the routing step can commence. On the outset, the problem appears complex, but this is simplified by subdividing it into two stages, namely *global routing* and *local or detailed routing* [8]. An intermediate step is needed as an adjustment between the two stages. Considered part of the global routing stage, cells and feedthroughs are repositioned according to their physical sizes and net routings through feedthroughs.

5.1. Global Routing

The global routing does not make any connections but lays a path for the detailed router to handle. Before global routing, the cells need to be in the optimized arrangement on the cell grid. A *feedthrough grid* is created (see figure 5.1) to assist the global router in keeping track of paths used to route a particular net uniquely identified by an *id*.

Each net is taken in turn and the shortest path calculated by assigning the net to the closest feedthroughs along a direct routing path. A *sequential routing* is performed, where congestion of channels isn't taken into account. Alternatively, an *order-independent routing* will take into account the congestion of particular channels and assign accordingly.

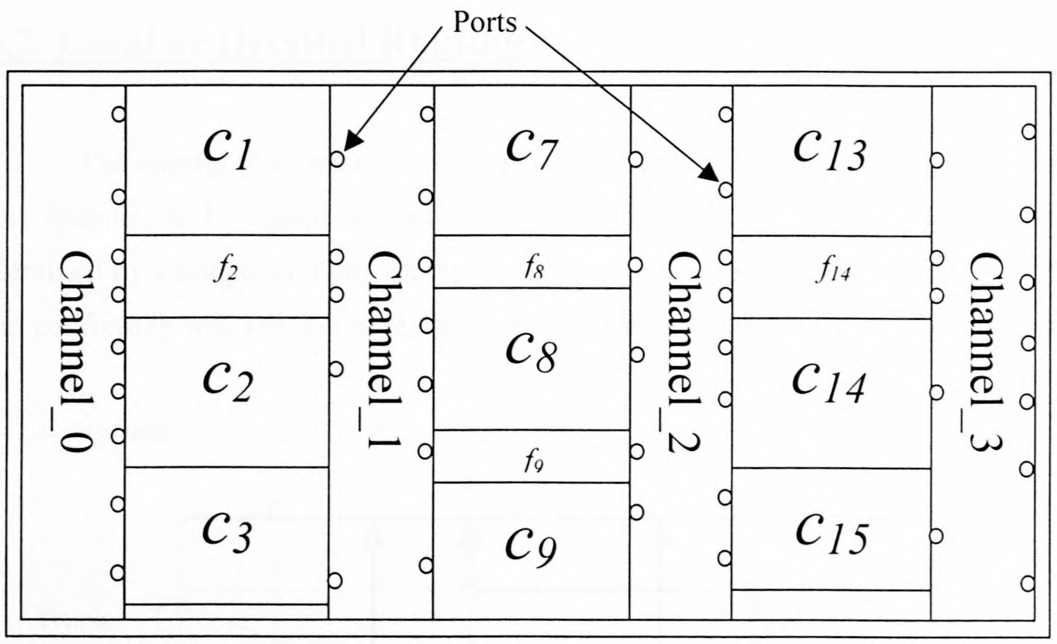


Figure 5.2. View of top left corner of placement after repositioning. Note that cell heights, feedthrough net assignments and channels dimensions have been taken into account.

Channels for local or detailed routing are now inserted between columns. In the channel creation process, the columns are again scanned and ports from the cells and feedthroughs inserted into channel port lists. Each of these ports has the unique *id* used in the routing process.

The implementations for the global routing algorithm are omitted in Appendix D, since it being mundane and lengthy in nature.

Improvements of the global routing can be made by including an iterative cycle that will make decisions concerning the routing through particular feedthroughs based on a minimization of channel congestions and other applicable factors.

5.2. Local or Detailed Routing

The routing of a channel is the last step needed in completing the layout problem. The channels to be routed are defined by a rectangular region with rows of terminals identified by a unique *id* along its top and bottom sides [26], [1] and [8] (see figure 5.3). For conformity with the literature, the channel is viewed in the horizontal orientation.

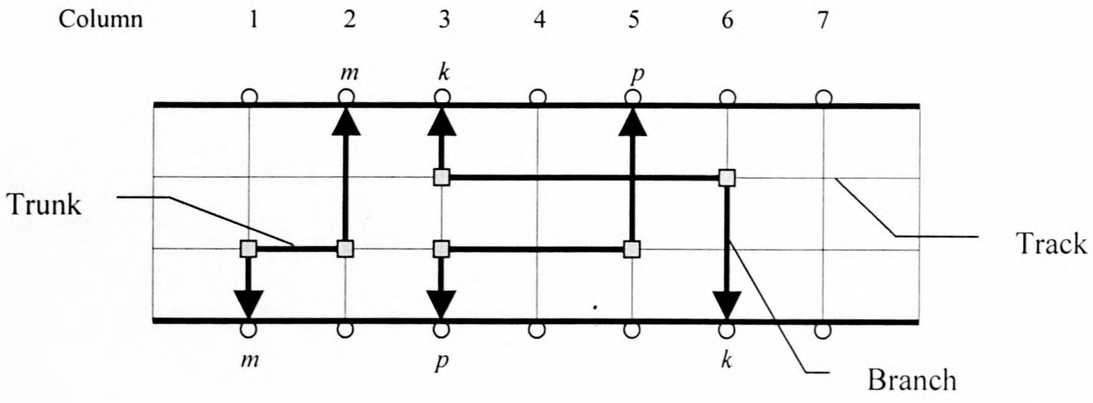


Figure 5.3. Channel, terminals (ports), trunks and branches viewed horizontally

There are many channel routing algorithms available [1][8][26]. Here we adopt the *left-edge algorithm* on which most channel router algorithms are based. This algorithm is suited for a two-layer routing, using one layer for *trunks* and the other for *branches* (see figure 5.3).

The left-edge algorithm proceeds as follows.

1. Sort the nets according to the leftmost edges of the nets horizontal segment
2. Assign the first net on the list to the first free track
3. Assign the next net on the list, which will fit, to the track
4. Repeat the process from step 3 until no more nets will fit in the current track.
5. Repeat steps 2-4 until all nets have been assigned to tracks
6. Connect the net segments to the top and bottom of the channel

The implementation of this algorithm can be found in Appendix (D.7). Here an improvement on the left-edge algorithm is made in that before assigning a trunk to a position, the trunks still to be assigned are scanned to ensure that a conflict doesn't arise. If two trunks compete for the same column, then an assessment is made as to which trunk should be placed first in the above algorithm.

An improvement to this algorithm can be the inclusion for adding a *dog-leg* [1] when two trunks compete, on both ends, for routing two in two channels.

6.1. Small Scale Test Circuit

A very small circuit was chosen to prove the effectiveness of the written software. This circuit has 4 cells, 9 nets, 5 input and 2 output ports (see figure 6.1.1.). Included will be the need for a splitter to be added on the net connected to port D.

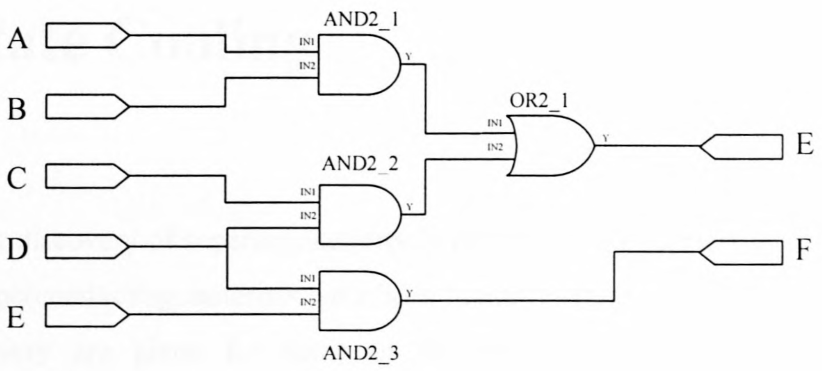


Figure 6.1.1. Small Scale Test Circuit with one fan-out connected to port D.

The layout for this circuit took approximately 10 seconds to complete. An intensive optimization wasn't necessary since we are more interested here in layout correctness. On close inspection of figure 6.1.2., the 5 input ports are located on the left side and the 2 output ports on the right side on the layout. A single splitter cell was added along with 3 terminators and 3 feedthroughs. The routing width was chosen at 6μm with a routing gap of 3μm.

Chapter 7

Solid State Cooling

Since the discovery of superconductivity in 1911 by Heike Kamerlingh Onnes [2], thousands of superconducting materials have been found. The transition temperatures and dates of discovery are given for some of the more important ones Figure 7.1. Collectively, the materials with transition temperatures above 23.2 K are referred to as *high-temperature superconductors (HTS)*. The metallic superconductors are usually called *low-temperature superconductors (LTS)*. Here a brief investigation is made into how high temperature superconducting electronics can be packaged and cooled at a lower cost.

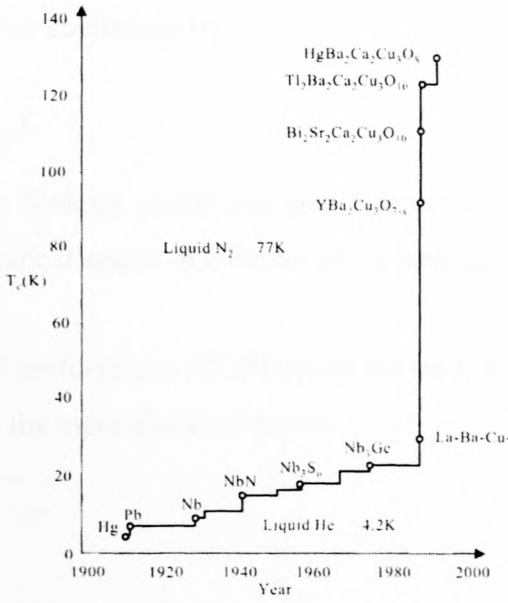


Figure 7.1. Transition temperatures of some important superconductors.

Thermoelectric coolers have been known to cool electronic components to 145K [17] using only cascaded Bi_2Te_3 n-type and p-type materials and 128K using a Bi_2Te_3 and BiSb materials [20]. An improved minimum temperature needs to be found below 70K in order to realize solid state cooled HTS RSFQ VLSI digital systems. The literature suggests the RSFQ may operate as high as 40K [24], although typical Josephson junction characteristics have been demonstrated at 70K [25].

7.1 Thermoelectric Cooling

The thermoelectric cooler is based on the Peltier effect which states that when an electrical current flows across a junction between two different materials, heat must be continuously added or subtracted at the junction to keep the temperature constant. The heat is proportional to the current flow and it's sign changes when the current is reversed.

$$q = \pi_{ab} I_{ab} \qquad (7.1)$$

where q is the rate at which heat is absorbed at the junction when the current I_{ab} passes from material a to b with π_{ab} being the Peltier coefficient.

For a small voltage difference between two junctions, the Seebeck voltage, discovered in 1821, is found to be linearly proportional to the temperature difference. This is related to the Peltier coefficient by

$$\pi_{ab} = Q_{ab} T \qquad (7.2)$$

where Q_{ab} is the Seebeck coefficient and T is the temperature at the junction. Figure 7.2 gives a schematic diagram of a Peltier effect heat pump.

The coefficient of performance (COP) of this device is defined by the amount of heat removed divided by the input electrical power.

$$\phi = q_c / P \qquad (7.3)$$

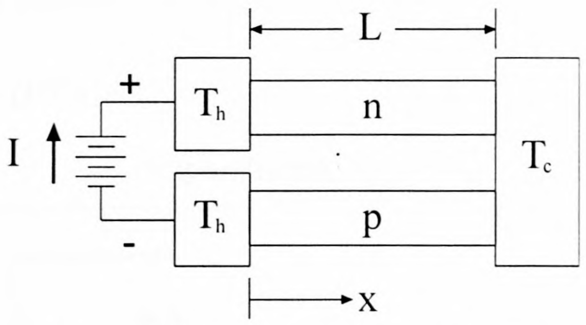


Figure 7.2. Simple thermoelectric cooler with branches n and p.

Three effects, namely the Peltier effect, Joule heating and thermal conduction, according to the following relation, determine the rate of heat removal from the cold reservoir

$$q_c = QT_cI - I^2R/2 - K\Delta T \tag{7.4}$$

where I is the applied current, R is the total resistance of the arms a and b, K is their thermal conductance and $\Delta T = T_h - T_c$. The applied voltage is given by

$$V = Q\Delta T + IR \tag{7.5}$$

from which the input power is

$$P = VI - (Q\Delta T + IR)I \tag{7.6}$$

where the first term is the Seebeck voltage term. The maximization of the Coefficient of Performance COP (equation 7.3), reveals devices using couples with a high value of Z_{ab} defined by

$$Z_{ab} = Q_{ab}{}^2/(RK) \tag{7.7}$$

deliver the optimum performance. The Z_{ab} parameter is called the figure of merit for the couple ab. The figure of merit for a single thermoelectric material is defined as

$$Z = Q^2/(\rho\kappa) \qquad [\deg^{-1}] \tag{7.8}$$

which is also written as the dimensionless figure of merit as,

$$ZT = Q^2 T / (\rho \kappa) \tag{7.9}$$

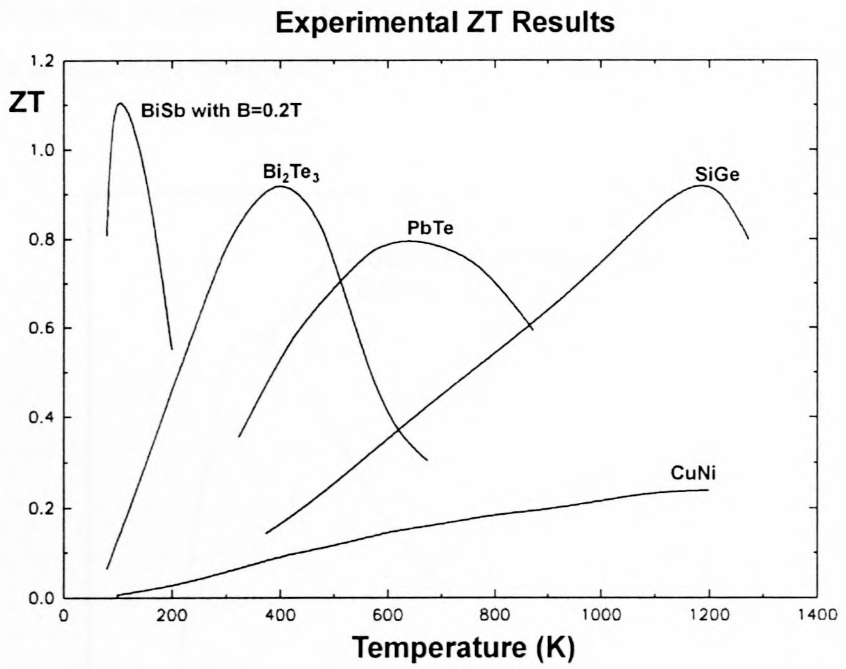


Figure 7.3. ZT parameters for various known thermoelectric materials[18], pre 1995.

Therefore, materials with a high Seebeck coefficient, a low resistivity and a low thermal conductivity are good candidates for application in thermoelectric devices. Figure 7.3 shows a plot ZT values over a wide temperature range for well-known thermoelectric materials pre 1995. It can also be shown that Z is high for materials with a high A-factor where

$$A \approx (\mu / \kappa_{ph})(m^* / m_0)^{3/2} \tag{7.10}$$

μ is the carrier mobility, κ_{ph} is the lattice component of the thermal conductivity, m^* is the density of states effective mass and m_0 the electronic mass.

7.1.1. Thermoelectric Efficiency

The heat pumping rate and COP of a couple is shown as a function of current in figure 7.4. Note that the maximum heat pumping rate q_{max} is much larger than the heat pumping rate at q_{cop} at the maximum COP.

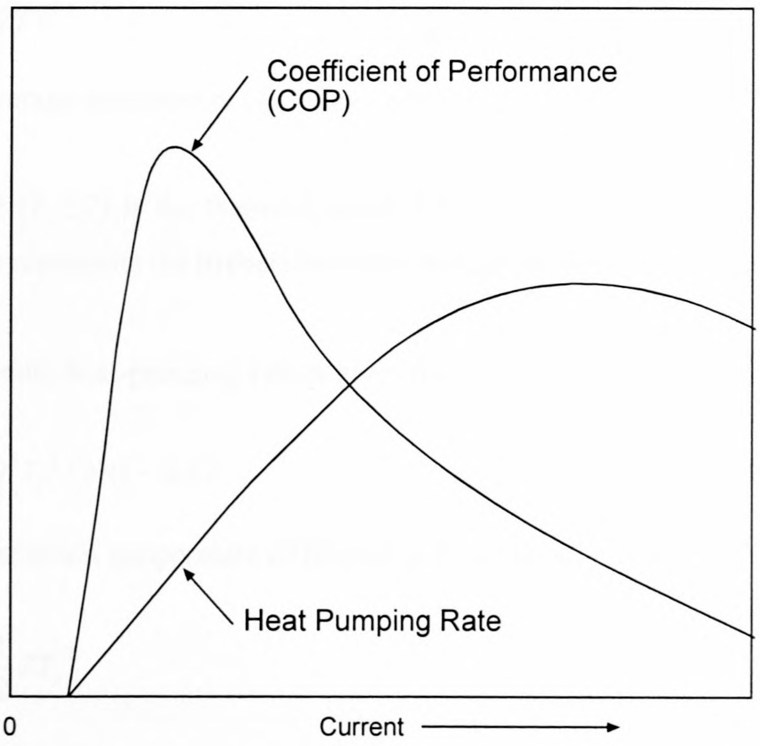


Figure 7.4. Variation of the coefficient of performance and heat pumping rate with current for a thermoelectric couple operating between given temperatures with fixed shape factors [17].

Above, the definition of the COP (equation 7.3) was given as

$$\phi = q / P = (QT_c I - I^2 R / 2 - K\Delta T)/(QI\Delta T + I^2 R) \tag{7.11}$$

The current at which the COP ϕ is a maximum is found by maximizing ϕ with respect to I. This results in a current

$$I_{COP} = Q\Delta T/(R(\omega - 1)) \tag{7.12}$$

and a maximum ϕ of

$$\phi_{\max} = (T_c/\Delta T)((\omega - T_h/T_c)/(\omega + 1)) \tag{7.13}$$

where

$$\omega = (1 + ZT)^{1/2} \tag{7.14}$$

T is the average temperature $(T_h + T_c)/2$ and Z is defined in equation 7.7.

The factor *($T_c/\Delta T$)* is the thermodynamic COP of a reversible Carnot cycle [17]. The second factor represents the irreversible heat conduction along the arms and the Joule loss.

The maximum heat-pumping rate is given by

$$q_{\max} = (Q^2 T_c^{\,2}/2R) - K\Delta T \tag{7.15}$$

and the maximum temperature difference is found to be

$$\Delta T_{\max} = \frac{1}{2} Z T_c^{\,2} \tag{7.16}$$

The lowest temperature on the cold face is determined using the equation

$$T_{c,\min} = ((1 + 2ZT_h)^{1/2} - 1)/Z \tag{7.17}$$

7.1.2. Fabrication of Thermoelectric Coolers

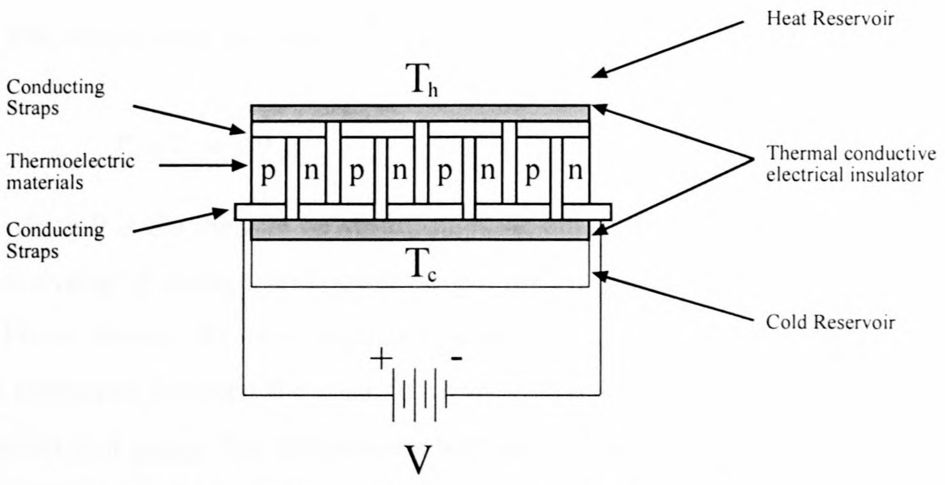


Figure 7.5. A single stage thermoelectric cooler

For N identical couples in series, the applied voltage is $V_n = NV$, the current $I_n = I$, power input $P_n = NP$, coefficient of performance COP $\phi_n = \phi$ and heat load $q_n = Nq$, where V, I, P, ϕ and q refer to the same quantities as for a single couple. Hence, the major advantage found here would be the increased amount of heat able to be removed from a cold reservoir.

7.1.3. Multistage Thermoelectric Units

Equation 7.17 shows that the maximum temperature difference that can be reached using a single-stage thermoelectric refrigerator is related to the figure of merit. Greater temperature differences are achieved using multistage cooling [23].

The coefficient of performance of an n-stage cascade is calculated assuming that each stage operates over $(T_h-T_c)/n$, where T_h is the hot side of the first junction, T_c is the cold side of the nth junction. It is assumed as an approximation that the coefficient of performance ϕ' for each stage is equal to $n(\phi_1 + 1/2) - 1/2$ where ϕ_1 is the coefficient of performance of a single stage working over the whole temperature range [23].

7.1.4. Heat Sink Choice

The temperature different, $T_2 - T_1$, across an element with a thermal resistance θ is

$$T_2 - T_1 = P\theta \tag{7.20}$$

where P is the thermal power through the unit [19]. Temperature difference is the electrical analog of voltage, and power or heat flow is the electrical analog of current.

For a device, the maximum temperature is given by $T_{j,max.}$. By definition, the thermal resistance between the case and heat sink is $\theta_{case\text{-}snk}$ and between the heat sink and ambient is $\theta_{snk\text{-}amb}$. The temperature between the device and the ambient can now be written as follows, when a heat sink is used.

$$T_2 - T_1 = P(\theta_{case-snk} + \theta_{snk-amb}) \tag{7.21}$$

The maximum safe power dissipation in a device is a function of:

$$P_{d,\max} = \frac{T_{j,\max} - T_{amb}}{\theta_{case-snk} + \theta_{snk-amb}} \tag{7.22}$$

The size of the heat sink becomes an issue if θ becomes less than 1 and air cooling needs to be employed. With air-cooling, θ can be reduced to 0.2 °C/W [17]. This implies that for a elegant packaging solution, $P_{d,max}$ must be less than 100W.

7.1.5. Future of Thermoelectric Materials

New thin-film superlattice materials are presently being invented and developed [18][22][12] with higher figures of merit than what has previously been attainable (see figure 7.7). The enhancement of these materials has allow for the potential to heat pump up to 700 Wcm^{-2} and localized cooling that occurs 23,000 times faster than with bulk materials [21].

The breakthrough occurs when a bulk thermoelectric material is arranged in a thin film, nanometer layered, superlattice. Due to the fact that the figure of merit, ZT (see

Table 7.1. Component List

Component	Description	Source	Cost	Availability
Fan	Force convection cooling of heat-sink	Electronic wholesalers	Medium	Very High
Heat Sink	Dissipates heat to environment	Electronic wholesalers	Low	Very High
Casing	MµMetal® vacuum sealed magnetic shielding	MµMetal®, USA	Low	High
Opto-coupling	Optically isolate cryogenic MCM from room-temperature environment.	Electronic wholesalers	Low	Very High
Microchip(s)	HTS Application Specific Integrated Circuit (ASIC)	Star Cryoelectronics	High	Very Low, custom made
Multiple Chip Module (MCM)	Cryogenic module to hold microchips.	Unknown	Medium	Very Low, custom made
Thin-film Thermoelectrics	New thin-film thermoelectric for cooling to cryogenic temperatures	RTI (Research Triangle Institute), License	High cost to set up process, materials low	Very Low, custom made

Chapter 8

Conclusion

The aim of this thesis was to develop software and research hardware that would allow for the realization of ultra high-speed Rapid Single Flux Quantum on a Very Large Scale of Integration. However, this would have little value if it were not a feasible undertaking.

The speed at which users of the technology will process information in the future will no doubt be awesome. Upon maturation, processor clock speeds reaching the sub-terahertz region should become the order of the day. All this without consideration of what possibilities quantum computing holds.

Sub-micron levels of fabrication are beginning to become commercially available for LTS and HTS through companies like Hypres and Star Cryoelectronics, respectively. The drive to HTS will see superconducting electronics being adopted by the larger consumer market. Here superconductors are sure to converge with other technologies like thermoelectrics and low profile cryopackaging technologies.

It is believed [29] that semiconducting electronics won't be able to exceed the 10GHz barrier due to delays incurred on interconnects by the resistance capacitance (RC) time constant. The resistance of a superconductor is zero and thus the fundamental limiting factor on its interconnecting delays is the speed of light.

References

[1] Sabih H. Gerez, "Algorithms for VLSI Design Automation", John Wiley & Sons, 1999

[2] Theodore Van Duzer and Charles W. Turner, "Principles of Superconductive Devices and Circuits", Prentice Hall PTR, 1999

[3] Sajjan G. Shiva, "Computer Design and Architecture", 3rd Edition, Marcel Dekker, Inc., 2000

[4] Coenrad Johan Fourie, "A 10GHz Oversampling Delta Modulating Analogue-to-Digital Converter Implemented with Hybrid Superconducting Digital Logic", MscEng Thesis, University of Stellenbosch, March 2001

[5] Peter Gross, "Development of a Rapid Single Flux Quantum Field Programmable Gate Array", B.Eng Thesis, University of Stellenbosch, November 2000

[6] K.K. Likharev and V.K. Semenov, "RSFQ Logic/Memory Family: A New Josephson-Junction Technology for Sub-Terahertz-Clock Frequency Digital Systems," IEEE Transactions on Applied Superconductivity, Vol.1, No.1, pp. 3-27, March 1991

[7] W.Chen, A.V.Rylyakov, Vijay Patel, J.E.Lukens and K.K.Likharev, "Rapid Single Flux Quantum T-Flip Flop Operating up to 770GHz", IEEE Transactions on Applied Superconductivity, Vol.9, No.2, June 1999

[8] Micheal John Sebastian Smith, "Application Specific Integrated Circuits", Addison-Wesley, 1999.

[9] H. Rogalla, "SCENET Roadmap for Superconductor Digital Electronics", Version 2.2, http://www.esas.org/Roadmaps/SDE-Roadmap, November 2001

[10] International SEMATECH, "International Technology Roadmap For Semiconductors", 2001, Personal Communication with Hypres.

[11] F.J.Rabie, "Superconducting COSL Building Blocks", MScEng Thesis, University of Stellenbosch, 1999

[12] US Patent 20010052234A1, Rama Venkatasubramanian, "Cascade Cryogenic Thermoelectric Cooler for Cryogenic and Room Temperature Applications", 21 March 2001.

[13] David Pellerin and Douglas Taylor, "VHDL Made Easy!", Prentice Hall PTR, 1996

[14] Electronic Design Interchange Format Version 2 0 0 (ANSI/EIA Standard 548-1988)

[15] Helman and Veroff, "Intermediate Problem Solving and Data Structures", Benjamin/Cummings Publishing Company, Inc., 1986

[16] Niobium Design Rules, Hypres Foundry, available on www.hypres.com

[17] J.H. Basson and E. Gaigher, "Thermoelectric Cooler Prestudy", CSIR Division of Microelectronics and Communications Technology, March 1989

[18] Cronin B. Vining, "Thermoelectric Technology of the Future", Defense Science Research Council Workshop, (http://www.zts.com), 1994

[19] Donald A. Neamen, "Electronic Circuit Analysis and Design", McGraw Hill, 1996

[20] W.M. Yim and A.Amith, "Bi-Sb Alloy for Magneto-Thermoelectric and ThermoMagnetic Cooling", Solid State Electronics, 1972, Vol.15, pp.1141-1165

[21] R.Venkatasubramanian, E.Siivola, T. Colpitts and B.O'Quinn, "Thin-film thermoelectric devices with high room-temperature figures of merit", Reseach Triangle Institute, (www.zts.com).

[22] Cronin B. Vining, "Summary Report on ICT99", (www.zts.com)

[23] H.J.Goldsmid, "Thermoelectric Refrigeration", Plenum Press, New York, 1964

[24] M. Huang, V. Komissinki, A.Yu. Kidiyarova-Shevchenko, M.Gustafsson, Eva Olsson, B.Högberg, Z.Ivanov and T.Claeson, "Small Scale Integrated Technology for HTS RSFQ Circuits", IEEE Transactions on Applied Superconductivity, Vol.11., No.1., March 2001.

[25] H.Shimakage, R.H.Ono, L.R.Vale and Z.Wang, "Interface-Engineered Josephson Junctions Optimized for High J_c", IEEE Transactions on Applied Superconductivity, Vol 11. No.2., June 2001.

[26] Sadiq M Sait and Habib Youssef, "VLSI Physical Design Automation, Theory and Practice", World Scientific, 1999

[27] Y.Hishinuma, T.H. Geballe and B.Y.Moyzhes, "Refrigeration by combined tunneling and thermionic emission in vaccum: Use of nanometer scale design", Applied Physics Letters, Volume 78, Number 17, April 2001.

[28] S.R. Whiteley, *SLINE Version 1.0,* June 1996, Available online at www.srware.com.

[29] Personnal commication with Hypres.

[30] ST405 Cryocooler 4.2K, Cryomech Inc., New York, USA, www.cryomech.com

[31] Cryotel Cryocooler 60K, Sunpower, Athens, Ohio, USA, www.sunpower.com

[32] Darren K. Brock, Alan M. Kadin, Alex F. Kirichenko, Oleg A. Mukhanov, Saad Sarwana, John A. Vivalda, Wei Chen and James E. Likens, "Retargeting RSFQ Cells to a Submicron Fabrication Process", available from www.hypres.com.

[33] Ivan Bozovic, "Atomic-Layer Engineering of Superconducting Oxides: Yesterday, Today, Tomorrow", IEEE Transactions on Applied Superconductivity, Vol.11, No.1, March 2001

[34] MμMetal® Megnetic Shielding Catalog and Design Guide, The MμMetal® Company, Inc., available through www.mushield.com.

[35] James D. Doss, "Engineer's Guide to High Temperature Superconductivity", John Wiley & Sons, 1989.

[36] Borland International Inc., 100 Borland Way, P.O. Box 660001, Scotts Valley, CA 95067-0001, *Borland Delphi version 6*

Appendix B

SPICE Library

The SPICE library used to perform the simulations of the RSFQ cells is included here. They have been modified for use with JSIM freeware SPICE simulator. These belong to the University of Stellenbosch SPICE library, developed by Coenrad Fourie [4] and optimised using specialised optimization software; tested using Monte Carlo simulations. The library includes the *rsfq_jtl, rsfq_div, rsfq_dro, rsfq_merge, rsfq_and, rsfq_or, rsfq_dc-sfq_conv* and the *rsfq_xor* cells.

```
* A=8 9=B 4=Vdd
.subckt rsfq_jtl 8 9 4
B0 2 0 jj1 area=0.25
B1 1 0 jj1 area=0.25
L0 6 5 1.98p
L1 6 2 0.132p
L2 7 1 0.132p
L3 8 6 1.98p
L4 3 5 0.132p
L5 7 9 1.98p
L6 5 7 1.98p
R0 4 3 14.3
R1 0 2 1.03
R2 1 0 1.03
.ends suny_jtl

* 5=A 4=B 12=C 2=Vdd
.subckt rsfq_div 5 4 12 2
B0 6 0 jj1 area=0.251
B1 3 0 jj1 area=0.355
B2 1 0 jj1 area=0.251
L0 6 7 0.053p
L1 9 8 0.053p
L2 10 1 0.053p
L3 8 11 0.132p
L4 5 3 0.053p
L5 5 8 1.16p
L6 7 4 1.98p
L7 7 9 1.64p
L8 10 12 1.98p
L9 9 10 1.64p
R0 0 6 0.71
R1 3 0 0.61
R2 11 2 8.36
R3 1 0 0.71
.ends rsfq_div

* 4=A 2=R 6=F 5=Vdd
.subckt rsfq_dro 4 2 6 5
B0 3 0 jj1 area=0.245
B1 1 0 jj1 area=0.27
B2 4 3 jj1 area=0.245
B3 2 1 jj1 area=0.27
L0 3 1 8.474p
L1 1 6 3.17p
R0 3 0 1.47
R1 1 0 0.863
R2 5 3 30
R3 4 3 0.72
R4 2 1 0.72
.ends rsfq_dro

* 9=B 4=C 13=A 7=Vdd
.subckt rsfq_merge 9 4 13 7
B0 8 6 jj1 area=0.225
B1 9 10 jj1 area=0.25
B2 5 6 jj1 area=0.225
B3 4 3 jj1 area=0.25
B4 2 1 jj1 area=0.25
L0 10 0 0.026p
L1 12 11 0.13p
L2 6 11 0.21p
L3 3 0 0.026p
L4 9 8 0.66p
L5 4 5 0.66p
L6 2 13 2p
L7 11 2 2.64p
L8 1 0 0.026p
R0 7 12 9.8
```

```
R1 8 6 1
R2 9 10 0.94
R3 5 6 1
R4 4 3 0.94
R5 2 1 0.71
.ends rsfq_merge

* A=15  B=5  F=20  Clk=10
Vdd=6
.subckt rsfq_and 15 5 20 10 6
B0 7 9 jj1 area=0.170
B1 8 0 jj1 area=0.41
B10 4 1 jj1 area=0.25
B2 14 17 jj1 area=0.25
B3 13 16 jj1 area=0.275
B4 15 14 jj1 area=0.25
B5 11 13 jj1 area=0.275
B6 12 9 jj1 area=0.170
B7 18 3 jj1 area=0.275
B8 5 4 jj1 area=0.25
B9 3 2 jj1 area=0.275
L0 9 8 0.1p
L1 17 0 0.026p
L10 1 0 0.026p
L2 16 0 0.026p
L3 14 13 8.5p
L4 13 12 3.2p
L5 19 8 0.026p
L6 8 20 4p
L7 3 7 3.2p
L8 4 3 8.5p
L9 2 0 0.026p
R0 7 9 1.1
R1 8 0 0.78
R10 6 4 30
R11 5 4 0.7
R12 3 2 0.85
R13 4 1 1.45
R2 14 17 1.45
```

```
R3 13 16 0.85
R4 11 13 0.7
R5 6 14 30
R6 15 14 0.7
R7 12 9 1.1
R8 6 19 20
R9 18 3 0.7
X0 rsfq_div 10 11 18 6
.ends rsfq_and

*  rsfq_or  A=1  B=2  F=3
Clk=4 Vdd=5
.subckt rsfq_or 1 2 3 4 5
X0 rsfq_merge 1 2 6 5
X1 rsfq_dro 6 4 3 5
R0 5 3 11
B0 3 0 jj1 area=0.25
R1 3 0 1.04
*X2 rsfq_jtl 7 3 5
.ends rsfq_or

*  16=SINUin   18=SQFout
10=Vdd
.subckt  rsfq_dc-sfq_conv  16
18 10
B0 8 7 jj1 area=0.171
B1 6 2 jj1 area=0.245
B2 4 5 jj1 area=0.148
B3 3 1 jj1 area=0.171
L0 15 4 1.27p
L1 7 3 0.29p
L10 2 0 0.13p
L11 1 0 0.18p
L2 5 3 0.69p
L3 16 15 3.35p
L4 15 8 1.29p
L5 9 17 0.08p
L6 6 18 2.11p
L7 17 6 1.74p
```

```
L8 7 17 1.13p
L9 4 0 3.59p
R0 8 7 1.39
R1 6 2 0.76
R2 10 9 6.42
R3 4 5 1.37
R4 3 1 1.12
.ends rsfq_dc-sfq_conv

*  6=A  4=B  1=F  5=CLK
22=Vdd
.subckt rsfq_xor 6 4 1 5 22
B0 9 7 jj1 area=0.171
B1 8 2 jj1 area=0.171
B2 3 10 jj1 area=0.193
B3 10 0 jj1 area=0.171
B4 6 7 jj1 area=0.245
B5 7 0 jj1 area=0.171
B6 5 10 jj1 area=0.221
B7 4 2 jj1 area=0.245
B8 2 0 jj1 area=0.171
L0 9 12 5.07p
L1 12 3 0.4p
L2 8 12 5.07p
L3 10 1 4.75p
R0 9 7 0.87
R1 8 2 0.87
R10 3 10 0.98
R11 10 0 1.32
R2 4 2 0.71
R3 6 7 0.71
R4 11 9 11.6
R5 22 11 23.1
R6 11 8 11.6
R7 7 0 0.87
R8 5 10 0.75
R9 2 0 0.87
.ends rsfq_xor
```

Appendix C

Technology Mapping File

A *Technology Mapping File* is used in specifying lower level parameters used in the *layout process*. This file provides the mapping information from the input EDIF netlist to RSFQ.

```
[General]                 * General
StandardWidth = 215       * Width of the standard cell
RouteWidth = 6       * Width of the routing tracks
RouteSpace = 6       * Distance between routing tracks
Gamma = 7                 * Distance between adjacent tracks in the same channel
CLKpos = 185              * Position of CLK special port in the horizontal plane
VDDpos = 43               * Position of VDD special port in the horizontal plane
ViaLayer = 3              * Layer number in GDSII file for via, i.e. I1b
HorzLayer = 6             * Layer number in GDSII file for Horizontal routing
VertLayer = 1             * Layer number in GDSII file for Vertical routing
VddLayer = 10        * Layer number in GDSII file for Vdd routing

[RSFQSPLITTER]            * RSFQ SPLITTER
Inputs   = 1              * Number of Inputs
Outputs  = 2              * Number of Outputs
Height   = 70             * Standard Cell Height
In1_Pos  = 43             * Input Position of In1
In1_Name = 'IN1'          * Name of Port In1
Out1_Pos = 12        * Output Position of Out1
Out1_Name = 'Y1'          * Name of Port Out1
```

```
Out2_Pos = 51                 * Output Position of Out2
Out2_Name = 'Y2'                      * Name of Port Out2
GDSStruct=SPLT_100_26* GDSII Struct

[TERMINATOR]                  * TERMINATOR CELL
Inputs   = 1
Outputs  = 1
Height   = 70
In1_Pos  = 5
In1_Name = 'Vddin'
Out1_Pos = 5
Out1_Name = 'Vddrep'
GDSStruct=TERM_100_26

[AND1]                        * REDUNDANT 1 INPUT AND CELL
Inputs   = 1
Outputs  = 1
Height   = 50
In1_Pos  = 25
In1_Name = 'IN1'
Out1_Pos = 25
Out1_Name = 'Y'
GDSStruct=                            * No GDSII Structure exists for AND1

[AND2]                        * TWO-INPUT AND CELL
Inputs   = 2
Outputs  = 1
Height   = 170
In1_Pos  = 30
In2_Pos  = 152
In1_Name = 'IN1'
In2_Name = 'IN2'
Out1_Pos = 143
Out1_Name = 'Y'
GDSStruct=AND2_100_26
```

```
[AND3]                    * THREE INPUT AND CELL#
Inputs   = 3
Outputs  = 1
Height   = 300
In1_Pos  = 50
In2_Pos  = 150
In3_Pos  = 250
In1_Name = 'IN1'
In2_Name = 'IN2'
In3_Name = 'IN3'
Out1_Name = 'Y'
Out1_Pos = 150
GDSStruct=AND3_100_26

[AND4]                    * FOUR INPUT AND CELL#
Inputs   = 4
Outputs  = 1
Height   = 430
In1_Pos  = 100
In2_Pos  = 200
In3_Pos  = 300
In4_Pos  = 400
In1_Name = 'IN1'
In2_Name = 'IN2'
In3_Name = 'IN3'
In4_Name = 'IN4'
Out1_Pos = 150
Out1_Name = 'Y'
GDSStruct=AND4_100_26
```

```
[DELAY]                    * REDUNDANT ONE INPUT DELAY CELL
Inputs  = 1
Outputs = 1
Height  = 50
In1_Pos = 25
In1_Name = 'IN1'
Out1_Pos = 25
Out1_Name = 'Y'
GDSStruct=

[DFF]                      * D-TYPE FLIP FLOP#
Inputs  = 4
Outputs = 1
Height  = 100
In1_Pos = 20
In2_Pos = 40
In3_Pos = 60
In4_Pos = 80
In1_Name = 'D'
In2_Name = 'CLK'
In3_Name = 'CLRN'
In4_Name = 'PRN'
Out1_Pos = 50
Out1_Name = 'Q'
GDSStruct=DFF_100_26

[INV]                      * INVERTER CELL
Inputs  = 1
Outputs = 1
Height  = 100
In1_Pos = 79
In1_Name = 'IN1'
Out1_Pos = 85
Out1_Name = 'Y'
GDSStruct=INV_100_26
```

```
[OR2]                          * FOUR INPUT OR CELL
Inputs  = 2
Outputs  = 1
Height  = 92
In1_Pos  = 38
In2_Pos  = 74
In1_Name = 'IN1'
In2_Name = 'IN2'
Out1_Pos = 73
Out1_Name = 'Y'

GDSStruct=OR2_100_26

[OR3]                          * THREE INPUT OR CELL
Inputs  = 3
Outputs  = 1
Height  = 120
In1_Pos  = 30
In2_Pos  = 60
In3_Pos  = 90
In1_Name = 'IN1'
In2_Name = 'IN2'
In3_Name = 'IN3'
Out1_Pos = 60
Out1_Name = 'Y'
GDSStruct=OR3_100_26

[OR4]                          * FOUR INPUT OR CELL
Inputs  = 4
Outputs  = 1
Height  = 150
In1_Pos  = 30
In2_Pos  = 60
In3_Pos  = 90
In4_Pos  = 120
In1_Name = 'IN1'
In2_Name = 'IN2'
```

```
In3_Name = 'IN3'
In4_Name = 'IN4'
Out1_Pos = 75
Out1_Name = 'Y'
GDSStruct=OR4_100_26

[OR6]                          * SIX INPUT OR CELL
Inputs   = 6
Outputs  = 1
Height   = 240
In1_Pos  = 20
In2_Pos  = 60
In3_Pos  = 100
In4_Pos  = 140
In5_Pos  = 180
In6_Pos  = 220
In1_Name = 'IN1'
In2_Name = 'IN2'
In3_Name = 'IN3'
In4_Name = 'IN4'
In5_Name = 'IN5'
In6_Name = 'IN6'
Out1_Pos = 120
Out1_Name = 'Y'
GDSStruct=OR6_100_26

[TRIBUF]                       * TRIBUF#
Inputs   = 2
Outputs  = 1
Height   = 40
In1_Pos  = 10
In2_Pos  = 30
In1_Name = 'IN1'
In2_Name = 'OE'
Out1_Pos = 50
Out1_Name = 'Y'
GDSStruct=TRIBUF_100_26
```

```
[XOR2]                    * TWO INPUT XOR CELL
Inputs   = 2
Outputs  = 1
Height   = 100
In1_Pos  = 37
In2_Pos  = 55
In1_Name = 'IN1'
In2_Name = 'IN2'
Out1_Pos = 84
Out1_Name = 'Y'
GDSStruct=XOR2_100_26
```

(#) Not yet implemented in RSFQ

Appendix D

Structure of RSFQ Layout Programme

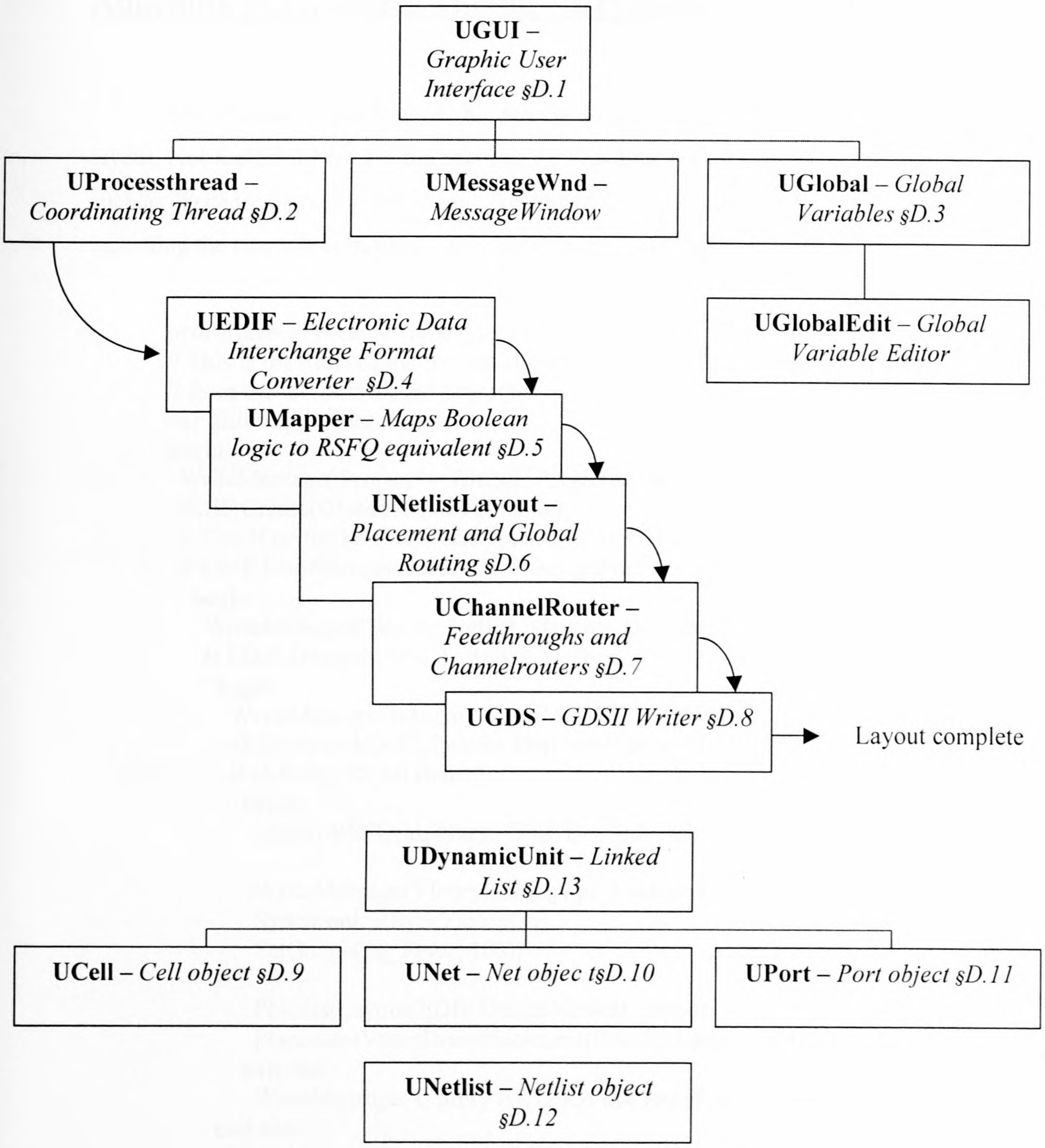

UGUI – Graphic User Interface §D.1
UProcessthread – Coordinating Thread §D.2
UMessageWnd – MessageWindow
UGlobal – Global Variables §D.3
UEDIF – Electronic Data Interchange Format Converter §D.4
UGlobalEdit – Global Variable Editor
UMapper – Maps Boolean logic to RSFQ equivalent §D.5
UNetlistLayout – Placement and Global Routing §D.6
UChannelRouter – Feedthroughs and Channelrouters §D.7
UGDS – GDSII Writer §D.8
Layout complete
UDynamicUnit – Linked List §D.13
UCell – Cell object §D.9
UNet – Net objec t§D.10
UPort – Port object §D.11
UNetlist – Netlist object §D.12

Figure D.1. RSFQ VLSI software programme overview of units denoted as Uxxxx.

Appendix D.2. Process Thread – UProcessThread.pas

The Execute procedure of the process thread controls the automation of circuit layout. Not included here are the routines for handling graphic interface synchronization such as *WriteMessage* and *SetGauge*. The thread allows the user to operate the GUI after executing the compile command, since the thread is made a background process.

```
procedure TProcessThread.Execute;
// This is the heart of the process thread. By using data that has been passed
// from the GUI the thread operates
var aLibrary : PLibrary;
begin
  WriteMessage('Project ' + Globals.ProjectName, pr_Medium);
  EDIF.Create(Globals.EDIFFilename);
  // Test if netlist has been loaded from EDIF file
  if EDIF.BuildSuccess and (EDIF.DesignNetlist <> nil) then
    begin
      WriteMessage('Testing Netlist Integrity', pr_Medium);
      if EDIF.DesignNetlist.TestIntegrity then
        begin
          WriteMessage('Integrity Test PASSED', pr_Medium);
          aLibrary := EDIF.Libraries.GetFromName('ALTERA');
          if aLibrary <> nil then
            begin
              MapToRSFQ(aLibrary, EDIF.DesignNetlist);

              WriteMessage('Floorplanning', pr_Medium);
              Synchronize(SyncFloorplan);
              SetGauge(gg_Floor, 100);

              PNetlistLayout(EDIF.DesignNetlist).Layout;
              PlacementView.DrawPlacement(PNetlistLayout(EDIF.DesignNetlist));
            end else
              WriteMessage('Library ALTERA not found', pr_Critical);
        end else
          WriteMessage('Integrity test FAILED', pr_Critical);
    end else
      WriteMessage('EDIF Netlist has not been loaded from file yet. Please specify
in Options.', pr_critical);
  GUI.ProcessStarted := False;
 end;
end.
```

Appendix D.3. Globals Variables – UGlobals.pas

The project file is a file of type TGlobalVariables. These variables are specified between the UGlobalEdit dialogue box, the Mapper file and run time programme decisions.

```
type TGlobalVariables = record

    Valid : Boolean;                  // Specifies whether globals are valid

    ProjectFileName,                  // Filename of the project
    ProjectName : String[255];        // Name of the project
    LibraryName : String[255];        // LibraryFilename
    EDIFFilename : String[255];       // Import EDIF Filename
    ExportFilename : String[255];     // Export GDSII Filename

    OptimizeTech : (GeneticAlg, Simulated); // Optimization technique?
    Genetic: TGeneticOpt;             // Technique parameters
    Simulated: TSimulatedOpt;

    CostFunction : (Area, Longest, AreaLongest);  // Cost Function

    StandardWidth, StandardHeight : Real; // Placement Grid cell dimensions

    Preplacement : record             // Floorplan variables
      GridSize : Real;
      NetWidth, NetHeight : Real;     //  Preplacement width and height
    end;

    Mapping : record                  // Technology mapping detail
      MappingFilename : String[255];
    end;

    Routing : record                  // Routing detail
      Gamma : Real;                   // Length before next track in channel
      Algorithm : (LeftEdge, Dynam);  // Routing algorithm used
      RouteWidth, RouteSpace : Real;  // Track width and spacing between
    end;

  end;
```

Appendix D.4. Electronic Data Interchange Format Importing – UEDIF.pas

In order to develop a portable platform from which to process the RSFQ VLSI design problem, the need to import and export Electronic Data Interchange Format (EDIF) data is necessary. Since many VHDL compiler programmes allow for the export of EDIF, it immediately allows the use of description languages.

The EDIF version 2 0 0 is described here and the algorithms developed to extract netlist information. The most important feature added to EDIF 3 0 0 is the ability to handle buses, bus rippers, and buses across schematic pages. EDIF 4 0 0 includes new extensions for PCB and multichip module (MCM) data. A complete description of the EDIF format is held by the **Electronic Industries Association (EIA) [14]**.

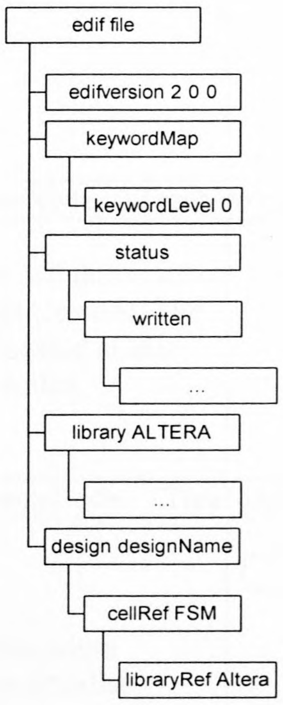
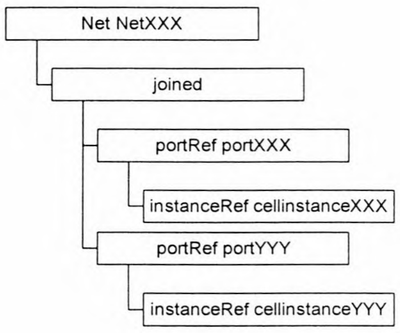


Figure B.2. FSM EDIF Example

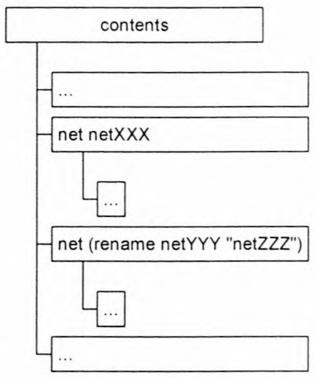
procedure ExtractNetConnections(NetNode : TTreeNode; aNet : PNet);

This procedure extracts net connections given a net node. The "joined" node is seen as a dummy node. The "portRef" node can have a child node "instanceRef" which refers to a cell instance within the current netlist. If the "portRef" child node is omitted, it implies that the port belongs to a interface port of the netlist



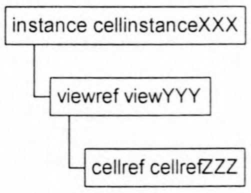
procedure ExtractNets(ContentsNode : TTreeNode; aNetlist : PnetlistLayout)

Given a netlist node, this procedure extracts the nets to aNetlist. The form of the net node is The nets are declared after the cell instance nodes. A rename node can be included.



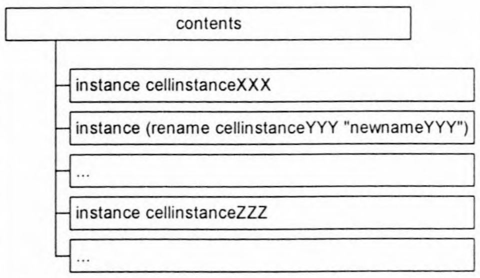
procedure ExtractCellRef(InstanceNode : TTreeNode; aCell : PCell)

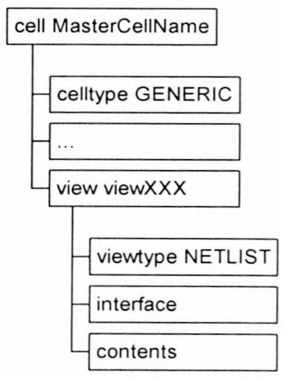
This procedure extracts the cell that is to be referenced to by an instance node. The reference is found and the passed aCells NetlistRef assigned to the netlist.

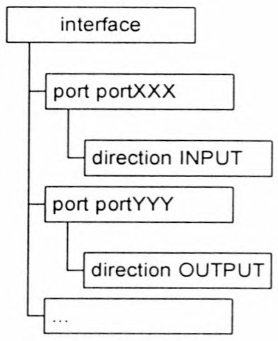


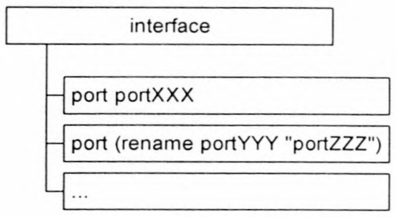
procedure ExtractInstances(ContentsNode : TTreeNode; aNetlist : PNetlistLayout)

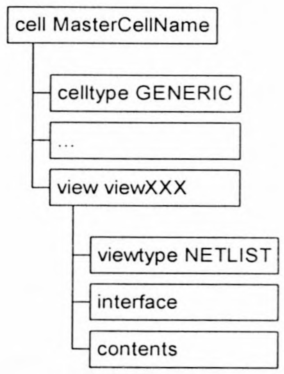
Given a netlist node, this procedure extracts the cell instances to aNetlist



procedure ExtractContents(NetlistNode : TTreeNode; aNetlist : PNetlistLayout)	
This scans the netlist or cell declaration for a contents node, and if it has one it extracts the cell instances and net connections.	 cell MasterCellName celltype GENERIC ... view viewXXX viewtype NETLIST interface contents

procedure ExtractPortType(PortNode : TTreeNode; var aPortType : TPortType)	
This procedure extracts the type of port attached to the interface node	 interface port portXXX direction INPUT port portYYY direction OUTPUT ...

procedure ExtractPorts(InterfaceNode : TTreeNode; aNetlist : PNetlistLayout)	
Given a netlist or cell view tree node, this procedure extracts the interface port information.	 interface port portXXX port (rename portYYY "portZZZ") ...

procedure ExtractInterface(NetlistNode : TTreeNode; aNetlist : PnetlistLayout)	
Given a netlist or cell tree node, this procedure finds the view node, then the interface. It then extracts the ports from the interface node	 cell MasterCellName celltype GENERIC ... view viewXXX viewtype NETLIST interface contents
procedure ExtractNetlists(LibraryNode : TTreeNode; aLibrary : Plibrary)	
Given a library tree node, this procedure scans through the library and extracts the cell or netlist information	
procedure ExtractLibrary(EDIFTree : TTreeNodes)	
This scans the first level of the EDIF Tree in search of libraries and extracts them.	
procedure GetDesignNetlist(EDIFTree : TTreeNodes);	
Get the reference to the design netlist	
function TEDIF.OpenEDIF(Filename : String; Tree : TTreeNodes) : Boolean;	
This is the coordinating procedure that will: 1. Read the EDIF from a file 2. Extract the read file into the tree Tree	

Below are extracts from the UEDIF unit file for the source code to the above procedures. A full source code listing would be too lengthy, approximately 814 lines.

```
procedure ExtractNetConnections(NetNode : TTreeNode; aNet : PNet);
 // This procedure extracts net connections given a net node. The form of the
 //  net node is
 //
 //    NetNode NetXXX
 //       |- joined
 //            |- portRef PortXXX
 //            |    |- instanceRef cellinstanceXXX
 //            ~
 //            ~
 //            |- portRef PortZZZ
 //
 // The "joined" node is seen as a dummy node. The "portRef" node can have a child
 // node "instanceRef" which refers to a cell instance within the current netlist.
 // If the "portRef" child node is omitted, it implies that the port belongs to
 // a interface port of the netlist

 var JoinedNode, PortRefNode, InstanceRefNode : TTreeNode;
    aPort : PPort;
    aCell : PCell;
    PortRefStr : String;
  begin
    // Locate dummy node, "joined"
    JoinedNode := FindFirstChildWith(NetNode, 'joined');
    if JoinedNode <> nil then
      begin
        // Locate first "portRef" child
        PortRefNode := FindFirstChildWith(JoinedNode, 'portref');
        while PortRefNode <> nil do
          begin
            // Get the name of the port to reference
            PortRefStr := GetNodeData(PortRefNode);
            // Find out whether this "portRef" node has a "instanceRef" node
            InstanceRefNode := FindFirstChildWith(PortRefNode, 'instanceref');
            if InstanceRefNode = nil then
              begin
              // This implies that the port is internal to the given netlist
                aPort := CurrentNetlist.PortInterface.GetFromName(PortRefStr);
              end else
              begin
              // This imples that the port is referenced by a netlist
                aCell                                                        :=
CurrentNetlist.Cells.GetFromName(GetNodeData(InstanceRefNode));
                aPort := aCell^.GetPort(PortRefStr);
              end;
            // Before adding the port to the net, test whether we are
          // referencing the same port
          if (aPort <> nil) then
            if (PortRefStr = aPort.Name) then
              begin
                // Add the port to the net
                aNet.AddPort(aPort)
              end
            else
              MessageDlg('Error: ExtractNetConnections on Net ' + aNet.Name, mtError,
[mbOk], 0)
            else
          MessageDlg('Error: ExtractNetConnections on Net ' + aNet.Name, mtError,
[mbOk], 0);
          // Search for the next sibling to match "portref"
          PortRefNode := FindNextSiblingFrom(PortRefNode.GetNextSibling, 'portref');
        end;
      end;
    end;

  procedure ExtractNets(ContentsNode : TTreeNode; aNetlist : PNetlistLayout);
  // Given a netlist node, this procedure extracts the nets to aNetlist.
  // The form of the net node is
  //
  //   contents
  //     ~
  //     ~
  //     |- net netXXX - ...
  //     |- net (rename netYYY "comment") - ...
  //     ~
  //     ~
  //     |- net netZZZ - ...
  //
  // The nets are declared after the cell instance nodes. A rename node can
  // be included

  var NetNode, RenameNode : TTreeNode;
     Net : PNet;
     NodeName : String;
    begin
     // Find the first "net" node off ContentsNode
     NetNode := FindFirstChildWith(ContentsNode, 'net');
     while NetNode <> nil do
       begin
         NodeName := GetNodeData(NetNode);
         // Test whether a rename node is present
         if Trim(NodeName) = '' then
        begin
         // If so, extract the renamenode
         RenameNode := FindFirstChildWith(NetNode, 'rename');
         if RenameNode <> nil then
          NodeName := GetRenameName(RenameNode)
         else
          begin
           MessageDlg('Error: ExtractNets', mtError, [mbOk], 0);
           NodeName := 'ERROR: Name not found';
          end;
        end;
       // Create a new net entity with Nodename
       Net := New(PNet, Create(NodeName));
       // Add the net the passed Netlist Nets list
       aNetlist.Nets.AddNode(Net);
       // Extract the connections the this net has to cell instances
       ExtractNetConnections(NetNode, Net);
       // Search for the next net node
       NetNode := FindNextSiblingFrom(NetNode.GetNextSibling, 'net');
      end;
  end;

procedure ExtractCellRef(InstanceNode : TTreeNode; aCell : PCell);
// This procedure extracts the cell that is to be referenced to by an
//  instance node. The reference is found and the passed aCell's
//  NetlistRef assigned to the netlist. The form of the instance node is
//
//  instance cellinstanceXXX
//      |- viewref viewYYY
//           |- cellref cellrefZZZ

var ViewRefNode, CellRefNode : TTreeNode;
   CellRefStr : String;
   NetlistRef : PNetlist;
  begin
   // Locate "viewref" and "cellref" nodes
   ViewRefNode := FindFirstChildWith(InstanceNode, 'viewref');
   CellRefNode := FindFirstChildWith(ViewRefNode, 'cellref');
   if CellRefNode <> nil then
     begin
       // Extract the reference to the cell/netlist
       CellRefStr := GetNodeData(CellRefNode);
       NetlistRef := CurrentLibrary.Netlists.GetFromName(CellRefStr);
       // Test whether the netlist/cell was found
       if NetlistRef <> nil then
         aCell.SetMasterCell(NetlistRef)
     else
       MessageDlg('Error: ExtractCellRef', mtError, [mbOk], 0);
    end else
      // If the cell instance is not to be referenced by any netlist,
      //  set the cells netlist to nil
      aCell.SetMasterCell(nil);
  end;

procedure ExtractInstances(ContentsNode : TTreeNode; aNetlist : PNetlistLayout);
// Given a netlist node, this procedure extracts the cell instances to aNetlist
// The form of the contents node is
//
//      Contents
//        |- instance cellinstXXX
//        |- instance (rename cellinstYYYY "comment")
//        ~
//        ~
//        |- instance cellinstZZZ
var InstanceNode, RenameNode : TTreeNode;
   CellInstance : PCell;
   RawRename, InstanceName : String;
  begin
   // Locate the instance node
   InstanceNode := FindFirstChildWith(ContentsNode, 'instance');
   while InstanceNode <> nil do
     begin
      // Extract the instance name
      InstanceName := GetNodeData(InstanceNode);
      // Test whether a rename node is present
      if Trim(InstanceName) = '' then
        begin
         // Locate rename node
         RenameNode := FindFirstChildWith(InstanceNode, 'rename');
         if RenameNode <> nil then
           begin
            // Extract the node name
            RawRename := GetNodeData(RenameNode);
            InstanceName := Copy(RawRename, 1, Pos(#32, RawRename)-1);
           end
         else
           MessageDlg('Error: ExtractInstances', mtError, [mbOk], 0);
        end;
      // Create a new cell instance
      CellInstance := New(PCell, Create(InstanceName));
      // Add the cell instance to the passes netlist
      aNetlist.Cells.AddNode(CellInstance);
      // Extract the netlist that this cell references
      ExtractCellRef(InstanceNode, CellInstance);
      // Search for the next instance
      InstanceNode := FindNextSiblingFrom(InstanceNode.GetNextSibling, 'instance');
    end;
  end;

procedure ExtractContents(NetlistNode : TTreeNode; aNetlist : PNetlistLayout);
// This scans the netlist/cell declaration for a contents node, and if it has
//  one it extracts the cell instances and net connections. The cell/netlist node
//  has the form:
//
//    cell cellname
//      |- celltype GENERIC
//      |- view viewXXX
//          |- viewtype NETLIST
//          |- interface
//          |    |- port ...
//          ~
//          ~- contents
var ViewNode, ContentsNode : TTreeNode;
  begin
    // Find the view node
    ViewNode := FindFirstChildWith(NetlistNode, 'view');
    if ViewNode <> nil then
      begin
        // Find the contents node
        ContentsNode := FindFirstChildWith(ViewNode, 'contents');
        if ContentsNode <> nil then
          begin
            // Extract the cell instance and net information
            ExtractInstances(ContentsNode, aNetList);
            ExtractNets(ContentsNode, aNetList);
          end;
      end;
  end;

procedure ExtractPortType(PortNode : TTreeNode; var aPortType : TPortType);
// This procedure extracts the type of port. The implementation here is
//
//    port portXXX
//      |  |- direction INPUT
//      |
//    port portYYY
//      ~  |- direction OUTPUT
//      ~
```

```
 var PortTypeNode : TTreeNode;
    Str : String;
  begin
    // Locate the direction node
    PortTypeNode := FindFirstChildWith(PortNode, 'direction');
    if PortTypeNode <> nil then
      begin
        // Retrieve the direction value
        Str := Trim(UpperCase(GetNodeData(PortTypeNode)));
        // Determine the port type
        if Str = 'INPUT'  then aPortType := pt_HierIn else
        if Str = 'OUTPUT' then aPortType := pt_HierOut else
        // If the type can't be found, report an error
        MessageDlg('Error: ExtractPortType - Invalid type', mtError, [mbOk], 0);
      end else
      begin
        // If the direction wasn't found report an error
        MessageDlg('Error: ExtractPortType - No type found, set to pt_None', mtError,
[mbOk], 0);
        aPortType := pt_None;
      end;
  end;

 procedure ExtractPorts(InterfaceNode : TTreeNode; aNetlist : PNetlistLayout);
 // Given a cell/netlist view tree node, this procedure extracts the interface port
 //  information. The interface node take the following form
 //
 //     interface
 //        |- port portXXX
 //        |- port (rename portYYY "portZZZ")
 //        ~
 //        ~

  var PortNode, RenameNode : TTreeNode;
     NewPort : PPort;
     PortStr : String;
     aPortType : TPortType;
  begin
   PortNode := FindFirstChildWith(InterfaceNode, 'port');
   while PortNode <> nil do
     begin
       // Extract port node
       if PortNode <> nil then
         begin
           RenameNode := FindFirstChildWith(PortNode, 'rename');
```

```
          if RenameNode <> nil then
            begin
              PortStr := GetRenameName(RenameNode);
            end else
            begin
              PortStr := GetNodeData(PortNode);
            end;
          // Extract port type from port node
          ExtractPortType(PortNode, aPortType);
          // Create a new port
          NewPort := New(PPort, Create(PortStr, nil, aPortType));
          // and ammend it to the PortInterface list
          aNetList.PortInterface.AddNode(NewPort);
          PortNode := FindNextSiblingFrom(PortNode.GetNextSibling, 'port');
        end;
      end;
    end;

  procedure ExtractInterface(NetlistNode : TTreeNode; aNetlist : PNetlistLayout);
  // Given a cell/netlist tree node, this procedure finds the view node, then the
  //  interface. It then extracts the ports from the interface node
    var ViewNode, InterfaceNode : TTreeNode;
      aPort : PPort;
    begin
      ViewNode := FindFirstChildWith(NetlistNode, 'view');
      if ViewNode <> nil then
        begin
          InterfaceNode := FindFirstChildWith(ViewNode, 'interface');
          if InterfaceNode <> nil then
            begin
              ExtractPorts(InterfaceNode, aNetList);

              // Add special ports
            // Add ClkIn, ClkOut and Vddin and Vddout ports
              //aNetlist.PortInterface.AddNode(New(PPort,      Create('Vddin',       aNetlist,
pt_Special)));
              //aNetlist.PortInterface.AddNode(New(PPort,      Create('Vddout',      aNetlist,
pt_Special)));
              //aNetlist.PortInterface.AddNode(New(PPort,      Create('CLKin',       aNetlist,
pt_Special)));
              //aNetlist.PortInterface.AddNode(New(PPort,      Create('CLKout',      aNetlist,
pt_Special)));

              aNetlist.Vddin := New(PPort, Create('Vddin', aNetlist, pt_Vddin));
              aNetlist.Vddout := New(PPort, Create('Vddout', aNetlist, pt_Vddout));
              aNetlist.CLKin := New(PPort, Create('CLKin', aNetlist, pt_CLKin));
        aNetlist.CLKout := New(PPort, Create('CLKout', aNetlist, pt_CLKout));
      end;
    end;
  end;

procedure ExtractNetlists(LibraryNode : TTreeNode; aLibrary : PLibrary);
// Given a library tree node, this procedure scans through the library and
//  extracts the cell/netlist information
  var NetListNode : TTreeNode;
    NewNetlist : PNetlistLayout;
  begin
   NetListNode := FindFirstChildWith(LibraryNode, 'cell');
   while NetListNode <> nil do
    begin
     // Create a new netlist
     NewNetList := New(PNetlistLayout, Create(GetNodeData(NetListNode)));
     // Assign this net list to the CurrentNetlist
     CurrentNetlist := NewNetlist;
     // Add the netlist to the current library
     aLibrary.AddNetlist(NewNetList);
     // Extract the netlist's interface
     ExtractInterface(NetListNode, NewNetList);
     // Extract the netlist's contents
     ExtractContents(NetlistNode, NewNetList);
     // Search for next cell/netlist
     NetlistNode := FindNextSiblingFrom(NetlistNode.GetNextSibling, 'cell');
    end;
    // Reset CurrentNetlist
    CurrentNetlist := nil;
  end;

procedure ExtractLibrary(EDIFTree : TTreeNodes);
// This scans the first level of the EDIF Tree in search of libraries and
// extracts them
  var ChildNode, LibraryNode : TTreeNode;
    NewLibrary : PLibrary;
  begin
   // Find the child of the edif tree
   ChildNode := EDIFTree.Item[0];
   // Locate the first library within the EDIFTree
   LibraryNode := FindFirstChildWith(ChildNode, 'Library');
   while LibraryNode <> nil do
    begin
     // Create a new library
     NewLibrary := New(PLibrary, Create(GetNodeData(LibraryNode)));
     // Set the current library to the newly created one
```

```
        CurrentLibrary := NewLibrary;
        // Add the newly created library to the libraries list
        Libraries.AddNode(NewLibrary);
        // Extract the netlists of the library
        WriteMessage('Extracting Library ' + NewLibrary.Name, pr_Medium);
        ExtractNetlists(LibraryNode, NewLibrary);
        // Search for the next library in the EDIFTree
        LibraryNode := FindNextSiblingFrom(LibraryNode.getNextSibling, 'library');
      end;
    // Reset CurrentLibrary
    CurrentLibrary := nil;
  end;

procedure GetDesignNetlist(EDIFTree : TTreeNodes);
var DesignNode, NetlistRefNode, LibraryRefNode : TTreeNode;
  aNetlist : PNetlist;
begin
  aNetlist := nil;
  DesignNode := FindFirstChildWith(EDIFTree.Item[0], 'design');
  if DesignNode <> nil then
    begin
      NetlistRefNode := FindFirstChildWith(DesignNode, 'cellref');
      if NetlistRefNode <> nil then
        begin
          LibraryRefNode := FindFirstChildWith(NetlistRefNode, 'libraryref');
          if LibraryRefNode <> nil then
            begin
              aNetlist                                                        :=
GetLibraryFromName(GetNodeData(LibraryRefNode))^.Netlists.GetFromName(GetNod
eData(NetlistRefNode));
            end;
        end;
    end;

    if aNetlist <> nil then
      DesignNetlist := aNetList
    else
      MessageDlg('Error: GetDesignNetlist - Design not found', mtError, [mbOk], 0);
end;

begin
  // Clear the library dynamic list
  Libraries.RemoveAll;
  // Extract libraries from the EDIFTree
  WriteMessage('Extracting Libraries', pr_Medium);
  ExtractLibrary(EDIFTree);
  GetDesignNetlist(EDIFTree);
  if DesignNetlist <> nil then
    begin
      with DesignNetlist^ do
      WriteMessage('Retrieving Target Netlist (' + IntToStr(Cells.Count) + ' Cells and ' +
IntToStr(Nets.Count) + ' nets)', pr_Medium);
      // Return a successful message
      ResultMsg := 'EDIF information extracted';
      BuildEDIF := True;
    end else
    begin
      // Return an unsuccessful message
      ResultMsg := 'EDIF information corrupt';
      BuildEDIF := False;
    end;
end;

function TEDIF.GetLibraryFromName(aLibraryName : String) : PLibrary;
// Returns a library in the EDIF given its name
  begin
    GetLibraryFromName := PLibrary(Libraries.GetFromName(aLibraryName));
  end;

function TEDIF.OpenEDIF(Filename : String; Tree : TTreeNodes) : Boolean;
// This is the coordinating procedure that will:
// 1. Read the EDIF from a file
// 2. Extract the read file into the tree Tree
var R : String;
begin
  // Return false by default
  OpenEDIF := False;
  // Execute commands
  if LoadEDIF(Filename) then
    if BuildEDIFTree(Tree, R) then
      // Return true if successful
      OpenEDIF := True;
end;
```

Appendix D.5. RSFQ Logic Mapper – UMapper.pas

Appendix D.5.1. Remove Delay Instances AND1, OR1 and DELAY

```
procedure RemoveDelays;
// Removes delay cells
var aCell  : PCell;
   inNet, outNet : PNet;
   aPortPtr : PPortPtr;
   PortIn, PortOut : PPort;
   aPort : PPort;
   s : String;
   EraseCells : TCellDynamicList;
   EraseNets  : TNetDynamicList;

begin
  // Scan through netlist
  while TheNetlist.Cells.ForEach(aCell) do
   // Identify DELAY cells
   begin
    s := PNetlist(aCell.MasterCell).Name;
    if (s = 'DELAY') or (s = 'AND1') or (s = 'OR1') then
     begin
      // Search for input/output ports and nets
      PortIn := aCell.Ports.GetFromName('IN1');
      PortOut := aCell.Ports.GetFromName('Y');
      inNet  := PortIn.Connected_Net;
      outNet := PortOut.Connected_Net;

      // Add outNet output connections to inNet
      while outNet.PortList.ForEach(aPortPtr) do
       begin
        case aPortPtr.Port.PortType of
         pt_Out:;
         pt_HierIn:
         pt_In, pt_HierOut:
          begin
           // Add output connections to inNet.
           inNet.AddPort(aPortPtr.Port);
           aPortPtr.Port.Connected_Net := inNet;
          end;
        end;
       end;
```

Appendix D.5. RSFQ Logic Mapper – UMapper.pas

Appendix D.5.1. Remove Delay Instances AND1, OR1 and DELAY

```
procedure RemoveDelays;
// Removes delay cells
var aCell  : PCell;
   inNet, outNet : PNet;
   aPortPtr : PPortPtr;
   PortIn, PortOut : PPort;
   aPort : PPort;
   s : String;
   EraseCells : TCellDynamicList;
   EraseNets  : TNetDynamicList;

begin
  // Scan through netlist
  while TheNetlist.Cells.ForEach(aCell) do
   // Identify DELAY cells
   begin
    s := PNetlist(aCell.MasterCell).Name;
    if (s = 'DELAY') or (s = 'AND1') or (s = 'OR1') then
     begin
      // Search for input/output ports and nets
      PortIn := aCell.Ports.GetFromName('IN1');
      PortOut := aCell.Ports.GetFromName('Y');
      inNet  := PortIn.Connected_Net;
      outNet := PortOut.Connected_Net;

      // Add outNet output connections to inNet
      while outNet.PortList.ForEach(aPortPtr) do
       begin
        case aPortPtr.Port.PortType of
         pt_Out:;
         pt_HierIn:
         pt_In, pt_HierOut:
          begin
           // Add output connections to inNet.
           inNet.AddPort(aPortPtr.Port);
           aPortPtr.Port.Connected_Net := inNet;
          end;
        end;
       end;
```

```
            // Remove PortIn from inNet
            inNet.PortList.RemovePort(PortIn);

            // Remove instance of DELAY cell
            TheNetlist.Cells.Remove(aCell);

            // Remove outNet
            TheNetlist.Nets.Remove(outNet);
          end;
        end;
      end;
```

Appendix D.5.2. Remove Fan-out

```
procedure RemoveFanout;
// Scans through the netlist and removes fanout nets

 procedure RemoveFanoutOnNet(aNet : PNet);
 // Adds splitters to the netlsit
 var i, j, k, FanOutCount : Integer;
    aSplitter : PCell;
    PortStack : Array of PPort;
    IN1, Y1, Y2 : PPort;
    aPortPtr : PPortPtr;
    FanPort, aPort : PPort;
    newNet : PNet;
    s : string;

  begin
    // Calculate number of splitters needed
    FanoutCount := aNet.PortList.Count - 1;

    // Find the fanout port
    FanPort := nil;
    while aNet.PortList.ForEach(aPortPtr) do
     with aPortPtr.Port^ do
       if (PortType = pt_Out) or (PortType = pt_HierIn) then
         FanPort := aPortPtr.Port;

    // Error testing
    if FanPort <> nil then
     begin
       // FANOUT REMOVAL ALGORITHM ------------------------------------------------

       // Create stack and add driver port
       SetLength(PortStack, 1);
```

```
PortStack[0] := FanPort;


// CREATE FANOUT NETWORK WITHOUT FANOUT NETS--------------------

i := 0; j := 0;
while j < FanOutCount-1 do
 begin
   // Initialise
   IN1 := Nil; Y1 := nil; Y2 := nil;

   // Create and add next splitter
   AddSplitterInstance(aNet.Name + '_' + IntToStr(j), aSplitter, IN1, Y1, Y2);
   inc(j);

   // Connect new splitter with a net
   newNet := New(PNet, Create(aNet.Name + '_S' + IntToStr(i)));

   // Find first available port on the stack
   aPort := PortStack[0];

   // Add this port and the IN1 of recently created splitter instance
   newNet.AddPort(aPort);
   newNet.AddPort(IN1);

   // Specify connected nets
   aPort.Connected_Net := newNet;
   IN1.Connected_Net := newNet;

   // Add the net to the netlist
   TheNetlist.AddNet(newNet);

   // Remove the topmost port from port stack non-destructively
   for k := 0 to Length(PortStack)-1 do
     PortStack[k] := PortStack[k+1];
   Setlength(PortStack, Length(PortStack)-1);

   // Add output ports Y1 and Y2 to stack
   SetLength(PortStack, Length(PortStack)+2);
   PortStack[Length(PortStack)-2] := Y1;
   PortStack[Length(PortStack)-1] := Y2;

   // Increment global counter
   inc(i);
 end;
```

```
        // CONNECT PORT TO FANOUT NETWORK ---------------------------------------
        k := 0;
         while aNet.PortList.ForEach(aPortPtr) do
          with aPortPtr.Port^ do
            if (PortType = pt_In) or (PortType = pt_HierOut) then
              begin
                // Create connecting fanout net
                newNet := New(PNet, Create(aNet.Name + '_SF' + IntToStr(k)));

                // Add ports to nets
                newNet.AddPort(aPortPtr.Port);
                newNet.AddPort(PPort(PortStack[k]));

                // Connect ports via a net
                Connected_Net := newNet;

                // Add the net to the netlist
                TheNetlist.Nets.AddNode(newNet);

                inc(k);
              end;

        // Destroy PortStack
        SetLength(PortStack, 0);
      end;

      // Remove original fanout net - it has been replaced
      TheNetlist.Nets.Remove(aNet);

    end;

// REMOVEFANOUT MAIN PROCEDURE ------------------------------------------------
var aNet : PNet;

begin
  // Scan netlist for all nets
  while TheNetlist.Nets.ForEach(aNet) do
    begin
      // Target nets that have a fanout, i.e. net count > 2
      if aNet.PortList.Count > 2 then
        begin
          // Add splitters to aNet
          RemoveFanOutOnNet(aNet);
        end;
    end;
end;
```

Appendix D.6. Netlist Layout – UNetlistLayout.pas

Appendix D.6.1.Genetic Algorithm

```
procedure TNetlistLayout.Genetic(Population_Size, TotalGenerations : Integer);
// This procedure uses Genetic Optimization to minimize the total interconnecting
// net length of a given layout.

var i, j, Generation : Integer;
   Population, NewPopulation : TDynamicList;
   newPheno, oldPheno, aPhenoUnit,
   Parent1, Parent2, Child, BestSolution, Result : PPhenoUnit;
   BestCost : Real;
   aNode : PDynamicNode;
   Longest : Real;

begin
 // Randomize
 Randomize;

 // Create population data structures
 Population.Create;

 /// Randomly fill the population pool
 for i := 1 to Population_Size do
   begin
    // Randomly generate the phenotype data
    CreateRandomPhenotype(aPhenoUnit);

    // Calculate the cost
    PhenoCost(aPhenoUnit);

    // Add the new pheno to the population group
    Population.AddNode(aPhenoUnit);
   end;

 // Create a result pheno unit
 CreateRandomPhenotype(Result);
```

```
// Make the initial best cost very high
BestCost := 1e12;

for Generation := 1 to TotalGenerations do
 begin
   // Create population data structures
   NewPopulation.Create;

     for i := 1 to Population_Size  do
      begin
        // Select the best parents within the population pool
        Parent1 := SelectParent(Population);
        Parent2 := SelectParent(Population);
        // Perfrom an ordered cross over to get next generation
        Child := OrderedCrossOver(Parent1, Parent2);
        // Mutate the child
        MutatePheno(Child);
        // Add the new best child to the new population pool
        NewPopulation.AddNode(Child);
      end;

   // Clear the "old" population pool
   Population.Destroy;
   Population.Create;

   // Transfer the new population to the "old" population pool
   while NewPopulation.ForEach(aNode) do
     begin
       oldPheno := PPhenoUnit(aNode);
       newPheno := New(PPhenoUnit, Create(oldPheno.Name));
       newPheno.Cost := oldPheno.Cost;
       newPheno.Phenotype := oldPheno.Phenotype;
       Population.AddNode(newPheno);
     end;

   NewPopulation.Destroy;

   // Find the best solution
   BestSolution := SelectBest(Population);

   // Test whether the best solution beats the best from previous generations
   if BestSolution.Cost < BestCost then
     begin
       PhenoCost(BestSolution);
       Longest := LongestNetConnection;
```

```
        Result^ := BestSolution^;
        BestCost := BestSolution.Cost;
      end;
    end;
  // Netlist to the best result
  PhenoCost(Result);

  ResultantPheno := Result^;

end;
```

Appendix D.6.2. Simulated Annealling Algorithm

```
procedure TNetlistLayout.SimulatedAnnealing;

var e, i, j, k, l, srchwdth, SourceCol, SourceRow, DestCol, DestRow : Integer;
    Done : Boolean;
    InitialPl, BestPl, Temp : Real;

begin

 Randomize;

 // Initialise
 CurrentTemperature := Globals.Simulated.StartTemp;
 i := 0; j := 0;
 e := Globals.Simulated.NeighbourSize;
 srchwdth := 2*e + 1;

 InitialPl := 1e30;
 BestPl := InitialPl;

 // Main simulated annealling loop
 while CurrentTemperature > 0.1 do
  begin
   i := 0;
   while i < Cells.Count do
    begin
    SourceCol := i mod GridColumns; SourceRow := i div GridColumns;
    j := 0;
    Done := False;
    while (j < srchwdth*srchwdth) and not Done do
     begin
      DestCol := SourceCol - e + j mod srchwdth;
      DestRow := SourceRow - e + j div srchwdth;
```

```
        if (DestCol >= 0) and (DestCol <= GridColumns) and
          (DestRow >= 0) and (DestRow <= GridRows) and not
          ((DestCol = SourceCol) and (DestRow = SourceRow)) then

          begin
            if Accept(SourceCol, SourceRow, DestCol, DestRow) then
              begin
                Swap(SourceCol, SourceRow, DestCol, DestRow);
                Done := True;
              end;
          end;
        inc(j);
      end;
      inc(i);

      Temp := NetlistArea;

      if Temp < BestPl then
      begin
        BestGrid := Grid;
        BestPl := Temp;
      end;
    end;
  end;
```

Appendix D.7. Channel Router – Left Edge Algorithm – UChannelRouter.pas

```
procedure TChannelRouter.LeftEndAlgorithm;
var f : TConstraint;
  i, j : Integer;
  V : Boolean;
  D, Gamma : Real;

begin
 // Minimum distance between tracks in same column
 Gamma := Globals.Routing.Gamma;

 D := Gamma;

 AddConstraints;
 SetLength(ChannelSolution, 0);

 while Length(Constraints) > 0 do
  begin
   // Find first net
   i := FindConstraintAfter(-1e12);
   while (Length(Constraints) > 0) and (i >= 0) do
    begin
     // Scan to see if there are any constraints in the other direction
     V := false; j := 0;
     while (j <= Length(Constraints)-1) and not V do
      begin
       if j <> i then

       V := ((abs(Constraints[i].Min - Constraints[j].Min)<D) and
           ((Constraints[i].MinSide = Right) and (Constraints[j].MinSide = Left)))
           or
          ((abs(Constraints[i].Max - Constraints[j].Max)<D) and
           ((Constraints[i].MaxSide = Right) and (Constraints[j].MaxSide = Left)))
           or
          ((abs(Constraints[i].Max - Constraints[j].Min)<D) and
           ((Constraints[i].MaxSide = Right) and (Constraints[j].MinSide = Left)))
           or
          ((abs(Constraints[i].Min - Constraints[j].Max)<D) and
           ((Constraints[i].MinSide = Right) and (Constraints[j].MaxSide = Left)));
          inc(j);
      end;
      f := Constraints[i];

      if not V then
        begin
          // Add track to solution
          SetLength(RowSolution, Length(RowSolution)+1);
          RowSolution[High(RowSolution)] := f;
          RemoveConstraint(i);
          i := FindConstraintAfter(f.Max + Gamma);
        end else
          // Otherwise search for next track
          i := i+1;

    end;
    SetLength(ChannelSolution, Length(ChannelSolution)+1);
    ChannelSolution[High(ChannelSolution)] := RowSolution;
    SetLength(RowSolution, 0);
  end;
 Width   :=   (Length(ChannelSolution)+1)   *   (Globals.Routing.RouteWidth   +
Globals.Routing.RouteSpace);
end;
```

Appendix D.8. GDSII Writer – UGDS.pas

```
unit UGDS;

interface

uses UDynamicUnit, SysUtils, Math;

// GDS Format record constants

const
  gds_HEADER       = $0002; // 2-byte integer
  gds_BGNLIB       = $0102; // 12 2-byte integers
  gds_LIBNAME      = $0206; // ASCII string
  gds_REFLIBS      = $1F06; // 2 45-character ASCII strings
  gds_FONTS        = $2006; // 4 44-character ASCII strings
  gdS_ATTRTABLE             = $2306; // 44-character ASCII string
  gds_GENERATIONS           = $2202; // 2-byte integer
  gds_FORMAT       = $3602; // 2-byte integer
  gds_MASK         = $3706; // ASCII string
  gds_ENDMASKS     = $3800; // No data
  gds_UNITS        = $0305; // 2 8-byte floats

  gds_ENDLIB       = $0400; // No data
  gds_BGNSTR       = $0502; // 12 2-byte integers
  gds_STRNAME      = $0606; // Up to 32-characters ASCII string
  gds_ENDSTR       = $0700; // No data

  gds_BOUNDARY     = $0800; // No data
  gds_PATH         = $0900; // No data
  gds_SREF         = $0A00; // No data
  gds_AREF         = $0B00; // No data
  gds_TEXT         = $0C00; // No data
  gds_NODE         = $1500; // No data
  gds_BOX          = $2D00; // No data
  gds_ENDEL        = $1100; // End of element

  gds_ELFLAGS      = $2601; // 2-byte integer
  gds_PLEX         = $2F03; // 4-byte integer
  gds_LAYER        = $0D02; // 2-byte integers
  gds_DATATYPE     = $0E02; // 2-byte integer
  gds_XY           = $1003; // Up to 200 4-byte integer pairs
  gds_PATHTYPE     = $2102; // 2-byte integer
  gds_WIDTH        = $0F03; // 4-byte integer
  gds_SNAME          = $1206; // Up to 32-character ASCII string
  gds_STRANS         = $1A01; // 2-byte integer
  gds_MAG            = $1B05; // 8-byte float
  gds_ANGLE          = $1C05; // 8-byte float
  gds_COLROW         = $1302; // 2 2-byte integers
  gds_TEXTTYPE       = $1602; // 2-byte integer
  gds_PRESENTATION  = $1701; // 2-byte integer
  gds_ASCII          = $1906; // Up to 512-character string
  gds_NODETYPE       = $2A02; // 2-byte integer
  gds_BOXTYPE        = $2E02; // 2-byte integer


type TByteArray = array of byte;

   TGDSRec = record
    RecLength : Word;
    RecType   : Word;
    Data      : Array of Byte;
   end;

   PGDSElement = ^TGDSElement;
   TGDSElement = object(TDynamicNode)
    RecType : Word;
    Data : Array of Byte;
    constructor Create;
   end;

   TGDSElementDynamicList = object(TDynamicList)
    function ForEach(var aGDSElement : PGDSElement) : Boolean; virtual;
    function GetFromName(aName : String) : PGDSElement; virtual;
   end;

   PGDSStruct = ^TGDSStruct;
   TGDSStruct = object(TDynamicNode)
    Elements : TGDSElementDynamicList;
    constructor Create(aName : String);
    procedure Rectangle(X1, Y1, X2, Y2 : Real; Layer : Word);
    procedure Reference(RefName : String; X, Y : Real);
    procedure Path(X1, Y1, X2, Y2 : Real; Width : Real; Layer : Word);
    destructor Destroy;
   end;

   TGDSHeader = object(TDynamicNode)
    HeaderElements : TGDSElementDynamicList;
   end;
```

```
TGDSStructDynamicList = object(TDynamicList)
 function ForEach(var aGDSStruct : PGDSStruct) : Boolean; virtual;
 function GetFromName(aName : String) : PGDSStruct; virtual;
end;

PGDSWriter = ^TGDSWriter;
TGDSWriter = object
 GDSLibrary : TGDSStructDynamicList;

 LibraryF, OutputF : File of Byte;

 constructor Create(LibraryFilename, OutputFilename : String);
 function CreateNewStruct(aName : String) : PGDSStruct;

 function ReadGDSRec(var GDSRec : TGDSRec) : Boolean;

 function  WriteStructToOutput(aStruct : PGDSStruct) : Boolean;
 procedure WriteGDSRec(GDSRec : TGDSRec);

 destructor Destroy;

end;
```

Appendix D.9. Cell Object – UCell.pas

```
type PCell = ^TCell;
   TCell = object(TDynamicNode)
    // Cell Variables
    MasterCell : Pointer;          // Reference to a master cell
    Ports : TPortDynamicList;  // List of input and output ports
    CellX, CellY : Real;           // The cells X, Y coordinates
    Owner : Pointer;               // Netlist that owns the cell, used in partitioning

    Clkin, Clkout,                 // CLK and Vdd special ports
    Vddin, Vddout : PPort;
    Tag : Integer;                 // Used in genetic optimisation

    // Cell Methods
    constructor Create(CellName : String);        // Create a new cell
    procedure AddPort(aPort : PPort);             // Add a port to port list
    function GetPort(PortName : String) : PPort;  // Find port
    procedure SetMasterCell(Master : Pointer);   // Set master cell
    destructor Destroy;                           // Destroy cell
  end;

  TCellDynamicList = object(THashDynamicList)
    function GetFromName(aName : String) : PCell;
    function ForEach(var aCell : PCell) : Boolean;
  end;
```

Appendix D.10 Net Object – UNet.pas

```
type PPortPtr = ^TPortPtr;
   TPortPtr = object(TDynamicNode)
    Port : PPort;              // Port that is referenced
    constructor Create;
    destructor Destroy;
   end;

   TPortPtrDynamicList = object(TDynamicList)
    function RemovePort(aPort : PPort) : Boolean;
    function ForEach(var aPortPtr : PPortPtr) : Boolean; virtual;
    function GetFromName(aName : String) : PPortPtr; virtual;
   end;

// Define the net structure

type PNet = ^TNet;
     TNet = object(TDynamicNode)
    PortList : TPortPtrDynamicList; // List of PortPtr's

    constructor Create(aNetName : String);
    procedure AddPort(aPort : PPort);
    procedure SimpleDraw(Canvas : TCanvas);
    procedure Draw(Canvas : TCanvas);

    function GetNetLength : real;           // Return the total net length
    function GetLongestConnection : Real; // Return longest net connection
    destructor Destroy;
   end;

type PNetDynamicList = ^TNetDynamicList;
   TNetDynamicList = object(THashDynamicList)
    function GetFromName(aName : String) : PNet;
    function ForEach(var aNet : PNet) : Boolean;
   end;
```

Appendix D. 11 Port Object – UPort.pas

```
type PPort = ^TPort;
     TPort = object(TDynamicNode)

     // Port Variables
     Parent : Pointer;                  // Pointer to parent cell
     Connected_Net: Pointer;            // Pointer to connected net
     Id : Integer;                      // An integer ID used in routing
     Orientation: TOrientation;         // Port orientation to parent cell
     Orient_Offset      : Real;         // Offset according to orientation
     PortType        : TPortType;       // Type of Port
     PortX, PortY       : Real;         // Coords relative to parent cell
     Partition : Pointer;               // Parent's parent
     SPICEPort : Integer;               // Index to SPICE port

     // Port Methods
     constructor Create(aPortName : String; aParent : Pointer; aPortType :
TPortType);
     function IsPort(aPortName : String) : Boolean;
     procedure GetPortCoords(var AbsPortX, AbsPortY : Real);
     procedure Draw(Canvas : TCanvas);
     function IsReceiverPort : Boolean;
     function IsDriverPort : Boolean;
     function IsSpecial : Boolean;

     destructor Destroy;
    end;

    TPortDynamicList = object(TDynamicList)
     function ForEach(var aPort : PPort) : Boolean; virtual;
     function GetFromName(aName : String) : PPort; virtual;
    end;
```

Appendix D.12. Netlist Object – UNetlist.pas

```
unit UNetlist;

interface

uses UNet, UPort, UCell, UDynamicUnit, Graphics, Classes, Math, UGDS,
SysUtils;

type PNetlist = ^TNetlist;
  TNetlist = object(TDynamicNode)
   // Variables
   Cells : TCellDynamicList;            // Cells contained in netlist
   Nets  : TNetDynamicList;             // Nets
   PortInterface : TPortDynamicList;    // Interface Ports

   Clkin, Clkout, Vddin, Vddout : PPort; // Special ports

   NetlistWidth, NetlistHeight : Real;  // Netlist Dimensions in micrometres
   GDSStructName,                  // Reference to the GDS physical equivalent
   SPICEStructName : String;           // Reference to the SPIE structurename

   // Methods
   constructor Create(aNetlistName : String);
   function  AddCell(aCell : PCell) : Boolean;
   function  AddNet(aNet : PNet) : Boolean;
   function  TestIntegrity : Boolean;
   function  TotalNetlength : Real;
   function  LongestNetConnection : Real;
   destructor Destroy;
  end;
```

Appendix D.13. Dynamic Lists – UDynamicUnit.pas

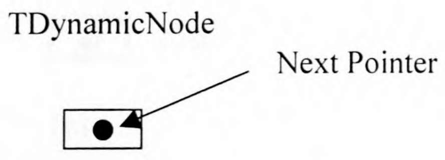
In order to handle vast collections of data, a unit was written to allow the dynamic memory allocation of data nodes in a list. The binary tree method was considered due to its improved searching speed. It was found that the added complexity is unnecessary since hashed linked lists could decrease the search order of complexity by a satisfactory factor.

Appendix D.13.1. Dynamic Nodes

At the center of the dynamic list unit is a dynamic node. Figure D.13.1 declares it as an object that has inheritance properties to be adopted by cell, net, port and other object types. It contains two variables, one to hold the name identity of the node, the other to reference the next node in a list.

```
type PDynamicNode = ^TDynamicNode
     TDynamicNode = object
       Name : String;
       Next : PDynamicUnit;
     end;
```

(a)



(b)

Figure D.13.1. Object Pascal declaration of TDynamicNode (a) and its graphic representation (b)

Table D.13.1. : TDynamicNode Methods

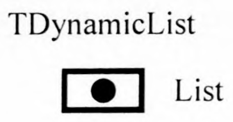
Method Name	**Description**	**Order of Complexity**
Create	Initializes the dynamic node. Must be called before attempting to add it to a list.	*1*
Destroy	Destroys the contents of the node.	*1*

Appendix D.13.2. Dynamic Lists

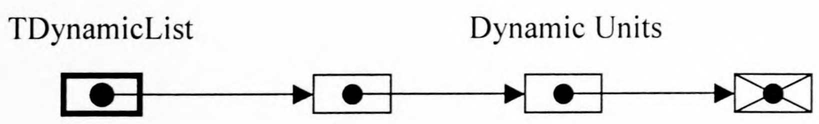
The dynamic nodes are handled by a dynamic list object. This object has methods that facilitate the manipulation of a node list, referenced by its *List* variable (see Figure D.13.2.). *List* points to the first dynamic node in the *list*. The following nodes are located by referencing the *next* pointers of these nodes (See figure D.13.2.c)

```
type PDynamicList = ^TDynamicList;
   TDynamicList = object
    List : PDynamicNode;
    constructor Create;
    procedure AddNode(aNode : PDynamicNode);
    procedure AddNodeAtEnd(aNode : PDynamicNode);
    function Remove(aNode : PDynamicNode) : Boolean;
    function RemoveAll : Boolean;
    function Index(i : Integer) : PDynamicNode;
    function Count : Integer;
    function GetIndex(aNode : PDynamicNode) : Integer;
    function GetFromName(aName : String) : PDynamicNode;
    function ForEach(var aNode : PDynamicNode) : Boolean;
    destructor Destroy;
   end;
```

(a)



(b)



(c)

Figure D.13.2. Pascal declaration of TDynamicList (a), its graphic representation (b), and an example dynamic list with dynamic nodes (c)

Table D.13.2. : TDynamicList Methods

Method Name	**Description**	**Order of Complexity**
Create	Initializes the dynamic list and must be called before attempting to manipulate a list.	*1*
AddNode	Adds a node to the beginning of the list (FIFO). Order of complexity is 1.	*1*
AddNodeAtEnd	Adds a node to the end of the list (LIFO).	*1*
Remove	Removes the specified node.	*N*
RemoveAll	Removes all nodes from the list, but does not deallocate nodes.	*N*
GetIndex	Scans the list and returns the index of a node. ***Note:*** Has no meaning in a hashed list	*N*
GetFromName	Scans the list and returns a pointer reference to the specified node.	*N*

ForEach	Scans through a list sequentially and returns the next node in the list.
	
	usage:
	
	while aDynamicList.ForEach(aDynamicNode) **do**
	  begin
	    aDynamicNode.Name := ….
	  end;
	
	Note: A ForEach procedure cannot be used inside itself.
		N

Destroy	Destroys list by calling each nodes destroy method.	*N*

Appendix D.13.3. Hashed Dynamic Lists

In order to decrease the order of complexity of the seach method, GetFromName, a hashing object is decended from the dynamic list object and contains a "hash list" of dynamic lists. The hashing function works by taking the sum of the ASCII values of the desired node name to locate its hash list. In this manner the order of complexity is decrease by the number of hash lists (H).

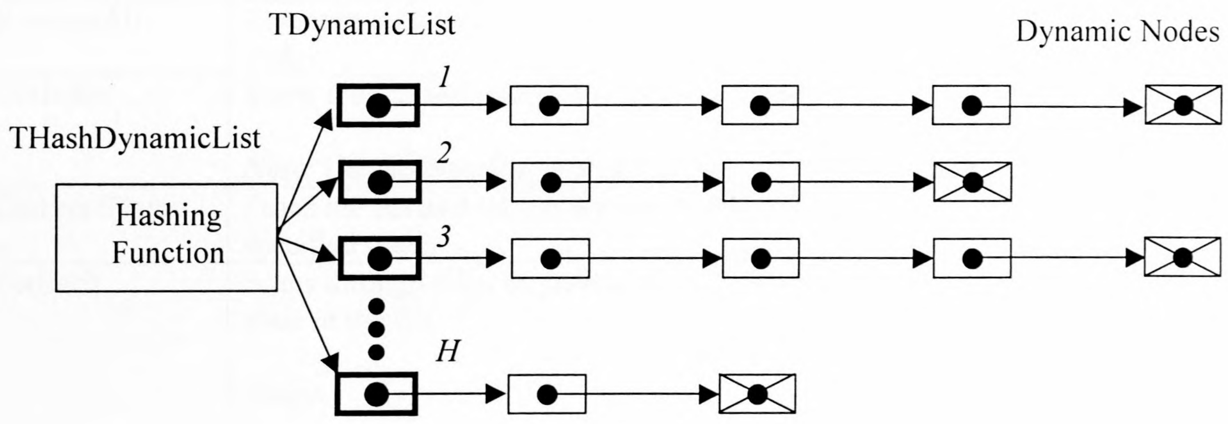


Figure D.13.3. A hashed dynamic list making use of a hashing function to decrease the order of complexity by a factor of H.

Table D.13.3. : THashDynamicList Methods

TDynamicList and THashDyanmicList		
Method Name	**Description**	**Order of Complexity**
Create(Hashsize)	Initialises a hashed dynamic list of H lists. Must be called before attempting to manipulate a hashed list.	*1*
AddNode	Adds a node to the beginning of the list (FIFO). Order of complexity is 1.	*1*
AddNodeAtEnd	Adds a node to the end of the list (LIFO).	*1*
Remove	Removes the specified node.	*N*
RemoveAll	Removes all nodes from the list, but does not deallocate nodes.	*N*
GetIndex	Scans the list and returns the index of a node. ***Note:*** Has no meaning in a hashed list	*N*
GetFromName	Scans the list and returns a pointer reference to the specified node.	*N/Hashsize*

ForEach	Scans through a list sequentially and returns the next node in the list.
	
	usage:
	
	while aDynamicList.ForEach(aDynamicNode) **do**
	  begin
	    aDynamicNode.Name := ....
	  end;
	
	Note: A ForEach procedure cannot be used inside itself.
		N

Destroy	Destroys list by calling each nodes destroy method.	*N*

```
unit UDynamicUnit;

interface

type PDynamicNode = ^TDynamicNode;
   TDynamicNode = object
    Name : String;
    Next : PDynamicNode;
    constructor Create(aName : String);
    destructor Destroy;
   end;

type PDynamicList = ^TDynamicList;
   TDynamicList = object

    List : PDynamicNode;    // Holds the pointer to the first DynamicNode

    constructor Create;

    procedure AddNode(aNode : PDynamicNode);
    procedure AddNodeAtEnd(aNode : PDynamicNode);
    function Remove(aNode : PDynamicNode) : Boolean;
    function RemoveAll : Boolean;
    function Index(i : Integer) : PDynamicNode;
    function Count : Integer;
    function GetIndex(aNode : PDynamicNode) : Integer;

    function GetFromName(aName : String) : PDynamicNode;
    function ForEach(var aNode : PDynamicNode) : Boolean;

    destructor Destroy;

    private

    Last : PDynamicNode;          // Holds the pointer to the last DynamicNode in a list

    NodeCount : Integer;         // Holds the count of nodes in a list
    ForEachFlag : Boolean;        // Holds the state of a ForEach loop
   end;

   THashDynamicList = object(TDynamicList)

    constructor Create(HashSize : Integer);

    procedure AddNode(aNode : PDynamicNode);
    procedure AddNodeAtEnd(aNode : PDynamicNode);
    function Remove(aNode : PDynamicNode) : Boolean;
    function RemoveAll : Boolean;
    function Index(i : Integer) : PDynamicNode;
    function Count : Integer;
    function GetFromName(aName : String) : PDynamicNode;
    function GetIndex(aNode : PDynamicNode) : Integer;
    function ForEach(var aNode : PDynamicNode) : Boolean;
    function GetHashList(Name : String) : PDynamicList;
    destructor Destroy;

    private

    HashList : Array of PDynamicList;

    HashMod : Integer;
    HashForEach : Boolean;
    CurrentForEach : Integer;

  end;
```