A MEMETIC ALGORITHM IMPLEMENTATION ON A FPGA FOR VLSI CIRCUIT PARTITIONING

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

STEPHEN COE

In partial fulfilment of requirements

for the degree of

Masters of Science

August, 2004

ABSTRACT

A MEMETIC ALGORITHM IMPLEMENTATION ON A FPGA FOR VLSI CIRCUIT PARTITIONING

Stephen Coe
University of Guelph, 2004

Advisor:
Dr Shawki Areibi
Dr Medhat Moussa

During the last decade, the complexity and size of circuits have been rapidly increasing, placing a stressing demand on industry for faster and more efficient CAD tools for VLSI design. One major problem is the computational requirements for optimizing the place and route operations of a VLSI Circuit. Thus, this thesis investigates the feasibility of using Reconfigurable Computing platforms to improve the performance of CAD optimization algorithms for the VLSI circuit partition problem. The proposed Reconfigurable Computing Genetic Algorithm architecture achieved a 5x speedup over conventional software implementation while maintaining 85% solution quality. Furthermore, a Reconfigurable computing based Memetic Algorithm improved upon this solution while using a fraction of the execution time required by the conventional software based approaches.

This thesis also investigates the tradeoff of developing Reconfigurable computing solutions using a high-level language (Handel-C) vs a low-level language (VHDL).

Implementing a Local Search algorithm in VHDL produced speedups of nearly twice that of the Handel-C implementation while requiring five times more development time. This speedup is a result of optimizing the VHDL architecture to target the specific FPGA hardware.

Acknowledgements

My sincere thanks go to Dr Shawki Areibi and Dr Medhat Moussa for their support and advice throughout this research. Without their help, this work would never have been possible.

To

my family and friends

whose love and encouragement helped accomplish this thesis.

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Approach 4

1.3 Contributions 4

1.4 Thesis Outline 6

2 Background/Literature Review **7**

2.1 Field Programmable Gate Array (FPGA) 8

2.1.1 FPGA Internal Design 9

2.2 Hardware Development Languages 11

2.2.1 Random Number Generators (RNG) 14

2.3 Reconfigurable Computing 16

2.3.1 Hardware/Software Co-design 16

2.3.2 Reconfigurable Algorithms 18

2.4 VLSI CAD Tools 19

2.4.1 Circuit Partitioning 21

2.4.2 Benchmarks 24

2.4.3 Genetic Algorithms 24
2.4.4 Local Search 31
2.5 Hardware Accelerators for CAD 34
2.6 Hardware Based Genetic Algorithms 36
2.6.1 Hardware/Software Co-Design Approaches 36
2.6.2 Pure Hardware Genetic Algorithm Implementations 37
2.6.3 Synthesized Hardware Genetic Algorithms 40
2.6.4 High-Level Hardware Implementation 41
2.7 Summary 44

3 A Genetic Algorithm Processer **45**

3.1 Hardware Design 45
3.1.1 Architecture Specifications and Constraints 47
3.1.2 Genetic Algorithm Architecture Overview 47
3.2 Create-Population-Module (CP-M) 55
3.2.1 Init-Population-Submodule (IP-SM) 56
3.2.2 Repair-Chromosome-Submodule (RC-SM) 59
3.2.3 Fitness-Calculation-Submodule (FC-SM) 64
3.3 Population-Reproduction-Module (PR-M) 68
3.3.1 Select-Parent-Submodule (SP-SM) 68
3.3.2 Cross-Parent-Submodule (CP-SM) 73
3.3.3 Mutate-Chromosome-Submodule (MC-SM) 77
3.3.4 Replace-Population-Submodule (RP-SM) 81
3.3.5 Copy-Parents-Submodule (CoP-SM) 84

3.4 Simulation and Verification . . . 86
3.4.1 Performance Analysis and Tuning . . . 89
3.4.2 Design Enhancement . . . 91
3.5 Computational Results . . . 94
3.5.1 Effect of Generation Size on Solution Quality . . . 98
3.5.2 Effect of Crossover Rate on Solution Quality . . . 100
3.5.3 Effect of Mutation Rate on Solution Quality . . . 102
3.5.4 Effect of Population Size on Solution Quality . . . 103
3.5.5 Effect of Balancing criteria on Solution Quality . . . 105
3.6 Summary . . . 105

4 Local Search and Memetic Architecture 107

4.1 Basic Local Search Procedure . . . 108
4.2 Hardware Design . . . 114
4.2.1 Local Search Memory Management . . . 115
4.3 Local Search Design and Architecture . . . 117
4.3.1 Partition-Update-Module (PU-M) . . . 118
4.3.2 Select-Next-Neighbourhood-Move-Module(SNNM-M) . . . 121
4.3.3 Data-Replicator-Module (DR-M) . . . 125
4.3.4 Search-Loop-Module (SL-M) . . . 129
4.3.5 Data-Update-Module (DU-M) . . . 133
4.3.6 Apply-Best-Move-Module (ABM-M) . . . 136
4.4 VHDL vs Handel-C implementation of Local Search Architecture . 139
4.4.1 Memory Management . . . 139

4.4.2 Resources 142
4.4.3 Delay Calculations of Local Search Architecture 142
4.4.4 Timing Results of Local Search Architecture 145
4.4.5 VHDL vs Handel-C: A comparison 145
4.5 Simulation/Verification of Local Search Architecture 146
4.5.1 Performance Analysis & Tuning 147
4.5.2 Design Modifications of Local Search Architecture 150
4.6 Computational Results of the Local Search Architecture 152
4.7 A Memetic Algorithm Hardware Accelerator 154
4.7.1 Memetic Algorithm Registers 156
4.7.2 Computational Results 162
4.8 Limitation of Hardware Implementation 163
4.9 Summary 166

5 Conclusions and Future Directions 168

5.1 Hardware CAD algorithms 169
5.2 Hardware Development Languages 170
5.3 Future Work 170

A Genetic Algorithm Module Pin Descriptions 174

B Local Search Module Pin Descriptions 185

C Genetic Algorithm Experimental Results 193

D Local Search and Memetic Algorithm Experimental Results 200

Bibliography **206**

List of Tables

2.1 Parallel Commands for Handel-C 12

2.2 Handel-C vs VHDL Resources 14

2.3 Benchmark Statistics 25

2.4 Statistical Connectivity of Benchmarks 25

2.5 Results of running the regression Problem[Mart01] 42

2.6 Results of running the XOR Problem[Mart01] 43

3.1 Handel-C constant definitions 51

3.2 Register Description 53

3.3 Memory Usage 56

3.4 Software/Hardware timing 88

3.5 Genetic Algorithm Software Profile 89

3.6 Genetic Algorithm Design Comparison 95

3.7 Base Case parameters for Handel-C Genetic Algorithm 96

3.8 Hardware improvement over Software 98

4.1 Local Search Software Profile 115

4.2 Priority State Machine Truth Table 141

4.3 VHDL vs Handel-C Local Search Resources 142
4.4 VHDL Memory Read Timing 144
4.5 VHDL Memory Write Timing 144
4.6 Execution Time of Development Languages 145
4.7 Local Search Software vs Hardware 147
4.8 Local Search Technique Comparison 152
4.9 New Local Search Software Profile 153
4.10 Memetic Algorithm Registers 157
4.11 Base Case parameters for Handel-C Memetic Algorithm 157

A.1 Signal Description of IPS 175
A.2 Signal Description of RCS 176
A.3 Signal Description of FCS 177
A.4 Signal Description of FCS (Con't) 178
A.5 Signal Description of SPS 179
A.6 Signal Description of CPS 180
A.7 Signal Description of CPS (con't) 181
A.8 Signal Description of MCS 182
A.9 Signal Description of RPS 183
A.10 Signal Description of CoPS 184

B.1 Signal Description of PUM 186
B.2 Signal Description of SNNMM 187
B.3 Signal Description of DRM 188
B.4 Signal Description of DRM (Con't) 189

B.5 Signal Description of SLM 190
B.6 Signal Description of DUM 191
B.7 Signal Description of ABMM 192

C.1 Hardware vs Software Comparison 194
C.2 Affect of Generation Size 195
C.3 Affect of Crossover Rate 196
C.4 Affect of Mutation Rate 197
C.5 Affect of Population Size 198
C.6 Affect of Balancing Difference 199

D.1 Affect of Difference Size on Local Search Algorithm 201
D.2 Exhausted Memetic Algorithm: Effect of Number of Random Individuals 202
D.3 Intermediate Memetic Algorithm: Effect of Number of Random Individuals 203
D.4 Intermediate Memetic Algorithm: Effect of Generation Size between Local Search 204
D.5 Exhausted Memetic Algorithm: Effect of Number of Iterations of Local Search 205

List of Figures

1.1 Overall Design Approach 5

2.1 IC Technology 8

2.2 FPGA Structure 9

2.3 Simple CLB 10

2.4 FPGA Internals 11

2.5 Handel-C Read Memory Access Signals [Celo03a] 13

2.6 Performance vs Flexibility of today's hardware 17

2.7 Interconnect delay vs Gate delay [Kang03] 20

2.8 Example of Circuit Partitioning 22

2.9 Genetic Algorithm mating process 26

2.10 One-Point Crossover 28

2.11 Uniform Crossover 29

2.12 An architecture for local search proposed by Abramson et.al. [Abra97] 32

2.13 Search neighbors by flipping all nets one by one 33

2.14 Protein folding problem: performance scaling as a function of FPGA size [Shac01] 40

3.1 Parallel Flow Genetic Algorithm 46
3.2 Genetic Algorithm Block Diagram 48
3.3 Netlist representation of a single net 49
3.4 Example of Chromosome Data 49
3.5 Example of Netlist Data 50
3.6 Handel-C constant definitions 52
3.7 Genetic Algorithm Memory Map 54
3.8 Create-Population-Module (CP-M) 57
3.9 Init-Population-Submodule Signal Diagram 58
3.10 Init-Population-Submodule Block Diagram 59
3.11 Internal design to Init-Population-Submodule 60
3.12 Stored chromosome data 60
3.13 Repair-Chromosome-Submodule Signal Diagram 61
3.14 Repair-Chromosome-Submodule Block Diagram 62
3.15 RC-SM Internal Repair Logic 64
3.16 Fitness-Calculation-Submodule Signal Diagram 65
3.17 Fitness Calculation Submodule Block Diagram 67
3.18 Fitness Compare Logic [Sitk95] 68
3.19 Internal Fitness Layout 69
3.20 Population Reproduction Module (PR-M) 70
3.21 Select-Parent-Submodule Signal Diagram 71
3.22 Select Parents Submodule Block Diagram 72
3.23 Masking Random Number 73
3.24 Selection of Unique Individuals 74

3.25 Crossover-Process-Submodule Signal Diagram 75
3.26 Cross-Parent-Submodule Signal Diagram 76
3.27 Crossover Combinational Logic 77
3.28 Mutation-Chromosome-Submodule Signal Diagram 78
3.29 Mutate-Chromosome-Submodule Block Diagram 79
3.30 Bit Mutation 80
3.31 Replace-Population-Submodule Signal Diagram 82
3.32 Replace Population Submodule Block Diagram 83
3.33 Copy-Parents-Submodule Signal Diagram 85
3.34 CoP-SM Block Diagram 86
3.35 Software vs Hardware comparison graph 87
3.36 New Netlist Storage using Integer Values 92
3.37 Bit Lookup using Integer Values 92
3.38 Population Reproduction with Block Rams 93
3.39 Parallel Pipeline Architecture 94
3.40 Hardware vs Software Results 97
3.41 Effect of Number of Generations on Mean Objective Value 99
3.42 Effect of Number of Generations on Standard Deviation 99
3.43 Effect of Number of Generations on Execution time 100
3.44 Effect of Crossover Rate on Mean Objective Value 101
3.45 Effect of Crossover Rate on Execution time 101
3.46 Effect of Mutation Rate on Mean Objective Value 102
3.47 Effect of Mutation Rate on Execution Time 103
3.48 Effect of Population Size on Mean Objective Value 103

3.49 Effect of Population Size on Standard Deviation 104
3.50 Effect of Population Size on Execution Time 104
3.51 Parallel Pipeline Architecture 105
3.52 Parallel Pipeline Architecture 106

4.1 Local vs Optimum Solution 108
4.2 Local Search Block Diagram 109
4.3 Local Search Data 111
4.4 Determining Feasible Move as only cut nets 112
4.5 Update Partition Data process 113
4.6 Local Search implemented in Hardware 116
4.7 Local Search Memory Map 117
4.8 Local Search Pipeline 118
4.9 Partition-Update-Module Signal Diagram 120
4.10 Partition-Update-Module (PU-M) Block Diagram 121
4.11 Update Logic for Partition Data 122
4.12 Determining Feasible Move 122
4.13 Select-Next-Neighbourhood-Move-Module Signal Diagram 123
4.14 Select-Next-Neighbourhood-Move-Module (SNNM-M) Process Flow 124
4.15 Select neighbourhood move 125
4.16 Data-Replicator-Module Signal Diagram 127
4.17 Data-Replicator-Module Block Diagram and flow 128
4.18 Applying Neighbourhood move to Solution Copy 129
4.19 Search-Loop-Module Signal Diagram 131

4.20 Search-Loop-Module (SL-M) Block Diagram and flow 132
4.21 Data-Update-Module Signal Diagram 134
4.22 Data-Update-Module Block Diagram 135
4.23 Apply-Best-Move-Module Signal Diagram 137
4.24 Apply-Best-Move-Module (ABM-M) Block Diagram 138
4.25 Priority State Machine 140
4.26 Memory Communication 141
4.27 Handel-C Parameter Passing 143
4.28 Handel-C vs VHDL Timing 146
4.29 Local Search Update Timing 149
4.30 Bit Net Representation 149
4.31 Parallel Partition-Data-Update 151
4.32 Timing Comparison of Local Search Modifications 153
4.33 Effect of Balancing Size on Local Search design 154
4.34 Memetic Algorithm Architecture Block Diagram 155
4.35 Exhaustive Memetic Algorithm Block Diagram 156
4.36 Memetic Algorithm Solution Landscape 157
4.37 Intermediate Memetic Algorithm (IMA) 158
4.38 Effect of Number of Random Individuals on Time (EMA) 159
4.39 Effect of Number of Random Individuals on Best Objective Value (EMA) 159
4.40 Effect of Number of Random Individuals on Time (IMA) 160
4.41 Effect of Number of Random Individuals on Best Objective Value (IMA) 161

4.42 Effect of Number of Local Search Iterations on Time (IMA) 161
4.43 Effect of Number of Local Search Iterations on Best Objective Value (IMA) 162
4.44 Effect of Number of GA Generations on Time (IMA) 162
4.45 Effect of Number of GA Generations on Best Objective Value (IMA) 163
4.46 Final Performance results of Algorithms 164

5.1 One Point Crossover Bit-mask 171
5.2 Parallel Fitness Calculation 172
5.3 Pipelined Fitness Calculation 172

Chapter 1

Introduction

During the last decade, the complexity and size of circuits have been rapidly increasing, placing a stressing demand on industry for faster and more efficient techniques for VLSI physical design automation. As the number of transistors increases in today's circuits beyond 100 million, hardware designers are becoming more and more dependent on computer aided design (CAD) tools to assist them in their designs. To aid in the placement and routing problem, heuristic algorithms are used in an attempt to find good solution in reasonable time. Even with the use of heuristic techniques and the speed of today's conventional computers, the ability to calculate an acceptable placement and routing solution is extremely time consuming. Using Moore's Law [Moor65], it is estimated that in 2008 the density of a chip will reach 3 billion transistors [Kang03]. With circuits of this size it will be relatively impossible for even the fastest and most efficient tools to solve effectively these circuit layout problems within an acceptable time frame. Therefore, it is necessary to develop faster strategies to aid hardware designers in this layout process.

One possible technique for achieving the necessary speed for these CAD tools is by creating the algorithms in Application Specific Integrated Circuits (ASIC). These circuits are optimized for a specific function with no unwanted overhead. ASIC's involve traditional logic gates and are manufactured at high costs with little flexibility [Comp99]. This high cost is due to the testing and development phases of the circuit. In addition, once the device is fabricated, any modifications to the circuit involves repeating the complete design process.

In the mid 1980's, a new technology emerged which has made it easier to develop application specific digital circuits. Reconfigurable computing (FPGAs) platforms combine the advantages of both traditional hardware and software design techniques. FPGAs have the ability to deliver the necessary speed and parallelism of hardware while maintaining the reconfigurability and flexibility of software. This allows for a single platform to be used for developing a wide variety of different hardware applications. The platform also allows for a fast and inexpensive method of designing and testing hardware. Traditionally, FPGA's have been constrained by their size and speed, restricting their use and application. As technology improved, FPGA devices have become larger and faster, thus allowing for the implementation of more complex designs.

Since FPGAs have been introduced, a new methodology of producing high performance digital circuits has arisen. A single hardware designer can create, test and implement a single algorithm in a fraction of the time and resources needed by traditional approaches while often achieving the necessary speedups. In using hardware parallelism and pipelining, FPGAs can achieve speedups of 10 to 100 times that of software implementations; however, they are still considerably slower than traditional ASIC designs [Chan97, DeHo99, Grah96]. It should be noted that although Reconfigurable Computing is much more flexible than traditional ASIC designs, it has nowhere near the flexibility of software implementations, and will never replace either traditional ASIC or software implementations. It can be thought of as a compromise, "filling the gap between [performance of] hardware and [the flexibility of] software" [Comp00a].

This thesis attempts to investigate the feasibility of using FPGA devices to improve performance of CAD algorithms by implementing a Memetic based algorithm for the VLSI circuit partition problem.

1.1 Motivation

As technology continues to increase in size and complexity, there is a need for faster search algorithms. With the introduction of FPGA's, there is a new opportunity to speed up these techniques by converting software algorithms into hardware implementations. In implementing a heuristic algorithm into hardware, through parallelism and pipelining, time consuming and repetitive loops can be efficiently implemented, decreasing the amount of time needed. In addition, once a successful implementation has been made for the VLSI circuit partitioning, these algorithms can be modified to other combinatorial optimization problems.

The motivation behind this research is to improve performance of a Genetic Algorithm for VLSI Circuit Partitioning by introducing a hardware design that exploits the inherent parallelism of the algorithm. The final goal is to develop a hardware implementation of a Memetic algorithm by incorporating a Local Search into the design to produce better results than a stand alone Genetic Algorithm. Finally, the tradeoffs of designing an algorithm using a high-level language, Celoxica Handel-C, vs a low-level language, Very High Speed Integrated Circuit Hardware Description Language (VHDL) will be investigated.

1.2 Approach

The proposed design is first programmed in ISO-C and is then converted to Handel-C. The final hardware design is implemented on the RC1000 development platform. The design process used can be seen in Figure 1.1.

1.3 Contributions

The main contributions of this research are as follows:

- Design and development of a Celoxica Handel-C implementation of a Memetic Algorithm that incorporates a novel local search methodology for circuit partitioning.

- Development of a VHDL and Handel-C implementation of a Local Search algorithm for circuit partitioning and highlighting the advantages/disadvantages of both approaches.

- Investigate the performance of Celoxica Handel-C architectures over traditional software implementations.

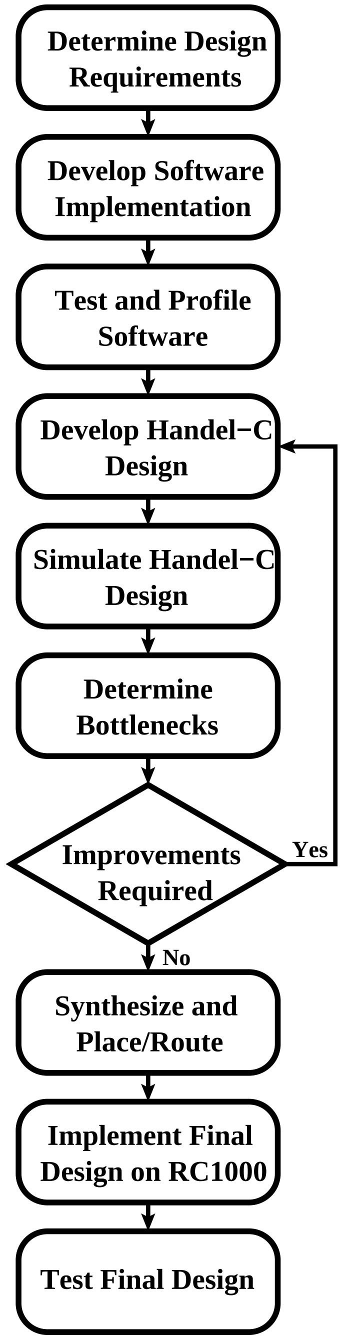


Figure 1.1: Overall Design Approach

1.4 Thesis Outline

This thesis is organized as following:

Chapter 2 - Background/Literature Review : This chapter will introduce the reader to the necessary background information for the thesis. It will also review past literature on hardware implemented CAD tools and previous hardware Genetic Algorithms designs.

Chapter 3 - Genetic Algorithm Architecture : This chapter describes the Genetic Algorithm architecture and experimental results from the hardware implementation.

Chapter 4 - Local Search and Memetic Architecture : This chapter describes the Local Search architecture and experimental results from the hardware implementation. It discusses the drawbacks of using a High-Level Language (handel-C) vs a low-level language (VHDL) in developing the Local Search design. Finally, the chapter describes the solution improvements of using the Memetic algorithm over the stand-alone Genetic Algorithm.

Chapter 5 - Conclusion and Future Directions : This chapter presents the conclusions generated from the research and possible future work.

Chapter 2

Background/Literature Review

As the complexity of Very Large-Scale Integration (VLSI) Computer Aided Design (CAD) algorithms increases, there is an increasing desire for better performance. One solution is to use Hardware Accelerators [Plat98] to increase the algorithm's performance. Hardware accelerators have become increasingly more popular over the past few years. These advances can be attributed to the increase in technology with respect to the size and speed of circuitry. In order to understand the capabilities of Hardware accelerators for CAD there is a need to review some key areas necessary for implementing such a design. This chapter discusses the following topics in detail: reconfigurable computing, VLSI CAD tools and past hardware implementations of Genetic Algorithms.

2.1 Field Programmable Gate Array (FPGA)

FPGAs [Comp00b] are devices which allow their logical blocks to be reconfigured in order to perform different tasks. These devices have created new avenues for semi-custom design, illustrated in Figure 2.1, allowing hardware engineers to implement their designs without undergoing the expensive Application Specific Integrated Circuits (ASIC) fabrication process.

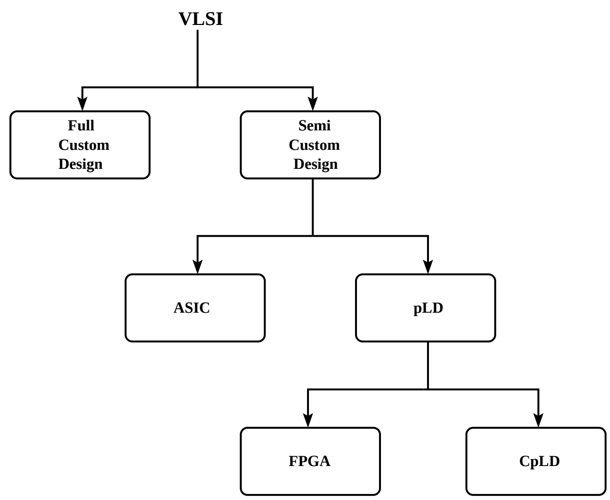


Figure 2.1: IC Technology

Although FPGAs have been around since 1985 [Xili03], their popularity has grown greatly within the last few years due to the rapid increase in speed and gate counts. The latter allows designers to implement larger complex designs onto single devices. With the ability to exploit pipelining and parallelization that have made traditional hardware so appealing, designers can now use FPGAS for rapid prototypes of their hardware designs. FPGAs also appeal to industry, allowing for fast development turnaround while saving expensive design and testing costs.

2.1.1 FPGA Internal Design

The internals of FPGAs consist of 3 elements as seen in Figure 2.2:

1. Configurable Logic Blocks (CLBs)

2. Input Output Blocks (IOBS)

3. Programmable Interconnects (Switching matrix)

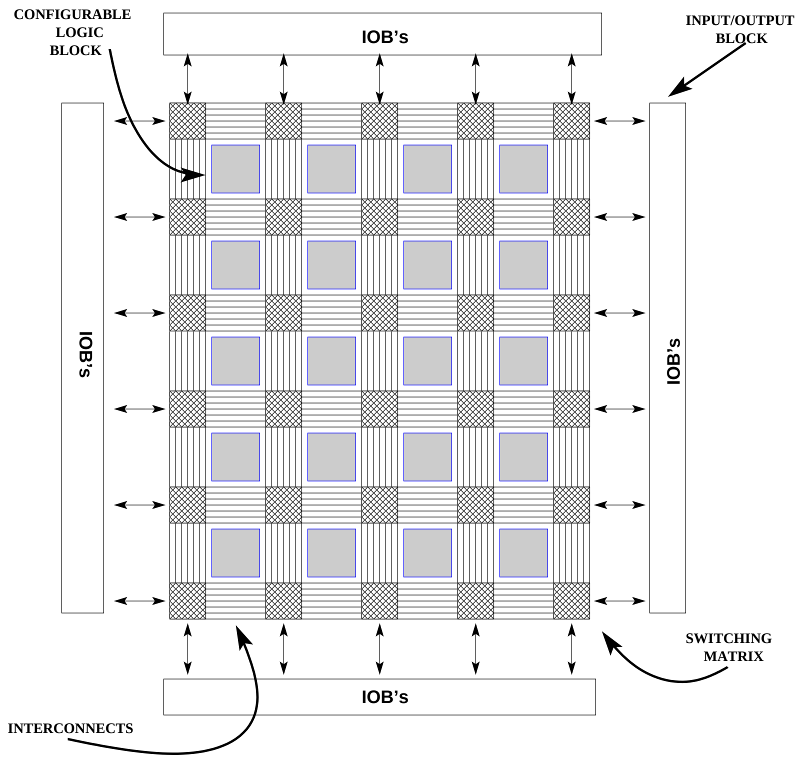


Figure 2.2: FPGA Structure

Configurable Logic Blocks (CLB)

Numerous CLBs are often grouped together to form a slice, which perform the logical function of a FPGA. Depending on the manufacturer and the generation, the structure of a CLB may vary. These CLBs are often comprised of Control, Multiplexors, Lookup tables and flip-flops which are used to generate the expected logics as shown in Figure 2.3.

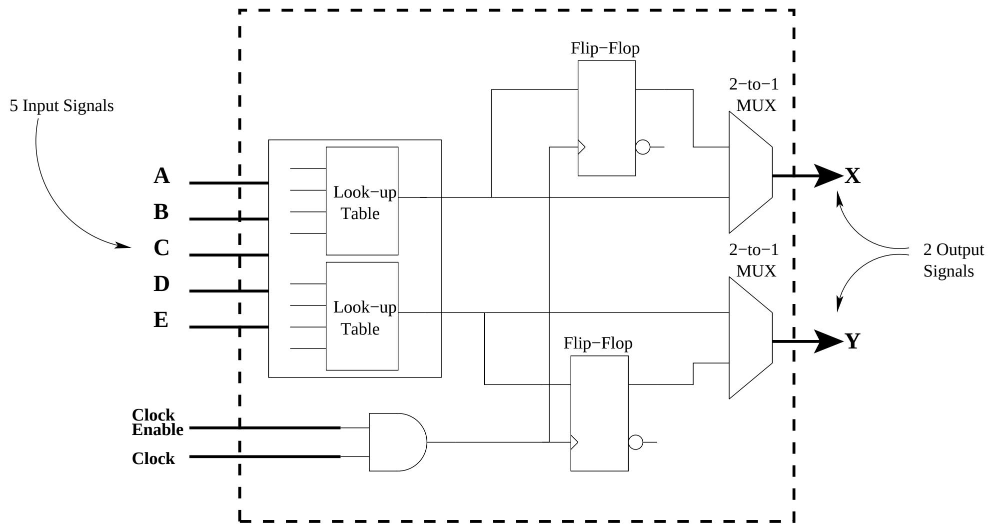


Figure 2.3: Simple CLB

Input Output Blocks

IOBs are the interface between the interior logical design and the outside world (I/O pins). The IOBs are often programmable so that each pin can handle either input or output signals. Figure 2.4(a) shows the structure of an IOB.

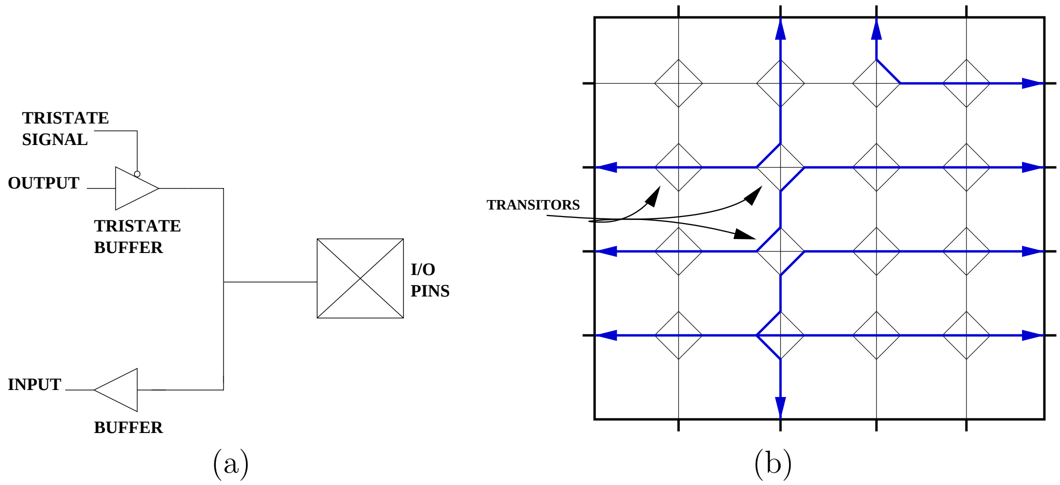


Figure 2.4: FPGA Internals

Programmable Interconnects

Programmable interconnects handle the data transfer between slices. These interconnects consist of a matrix of transistors which determines the path between the source of data and the final destination. Figure 2.4(b) is a simplified example of a switching matrix.

2.2 Hardware Development Languages

In programming hardware, there are a couple of techniques that can be used. The more common techniques are Verilog HDL and VHDL (Very high speed integrated circuit Hardware Description Language) [Comp99]. These languages use the idea of behavioral synthesis which describes how the algorithm functions in terms of inputs and outputs. Although these languages aid in the hardware design process, they require extensive time for designing and testing of the algorithms.

In the past few years, a new high level language has surfaced called Handel-C [Celo03b], which is based on the conventional ISO-C language format and is developed to assist software engineers in designing hardware. Handel-C allows designers to focus on the algorithm that is being implemented as opposed to the circuit that is being built.

Hardware Description Languages and Handel-C can be viewed in relationship to conventional software programming languages. VHDL and Verilog HDL are viewed as low level programming languages, similar to assembly language programming, whereas Handel-C is viewed as a high level language, like ISO-C. On compiling Handel-C code, the output code can be either VHDL to be ported to other VHDL code or a Electronic Design Interchange Format (EDIF) [Celo03a] file to be implemented directly into hardware. Handel-C follows a sequential programming structure, unlike most hardware description languages which are parallel by default [Loo02]. Although it maintains many of the functional properties of conventional C, there are extra features which enable the exploitation of parallelism. Some commands are shown in Table 2.1.

Function call	Functionality
Par...	Parallel Execution
Seq...	Sequential Exectuion
Par(Init; Test; Iter)	Parallel replication
Sew(Init; Test; Iter)	Sequential replication
chanin	Input Parallel communication
chanout	Output Parallel Communication

Table 2.1: Parallel Commands for Handel-C

There are many restrictions imposed on the Handel-C language over conventional-

C programming language. There is no available stack, making recursive functions difficult to implement. In addition, there is a limited amount of internal memory (arrays, variables, etc) which leads to a limiting factor on the size of the design. External memory can be used as a replacement for internal memory but often leads to slow executing clock frequency. Handel-C is developed so that each memory access, internal or external, occurs during one clock cycle. When utilizing external memory, Handel-C operates at a fraction of the operating frequency, allowing for the accessing signals to occur during a single clock cycle as shown in Figure 2.5. As in most high level languages, it is expected that in implementing designs in

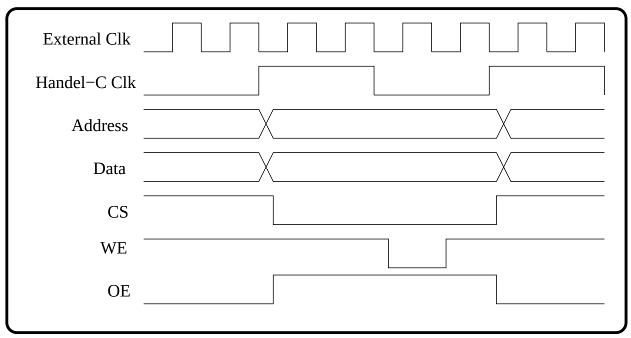


Figure 2.5: Handel-C Read Memory Access Signals [Celo03a]

Handel-C there would be a loss in the efficiency of the design, with a slight increase in both resources and delay times. Loo et al. [Loo02] found that in implementing a Data Encryption Standard (DES) and a Discrete Cosine Transform (DCT) the speeds of both Handel-C and VHDL were relatively comparable. The DES implementation executed 1.3 times faster under the Handel-C implementation compared to that based on VHDL. The DCT Handel-C implementation, however, executed at 0.75 that of the VHDL implementation. In comparing the size of the two implementations, the Handel-C design was 2-5 times larger. This large difference in area was assumed to be caused by the implementation of extra library functions not used in the algorithm. In looking at information from the Celoxica Website, the following comparison between VHDL and Handel-C implementation of a IPv6 header compression function on a Virtex 2000E-8 was found [Celo03b]:

	Handel-C/DK1	Verilog/Leonardo
Design Time	4 Man-months	12-16 Man-months
Program Size	40 pages	200 pages
Compile Time	3 minutes	1.5 hours
Size	17% Logic, 15% Memory*	All Logic, All Memory **
Speed	44 MHz	49 MHz

∗ Distributed memory. No block memory used

∗∗ A conscious choice. Used all logic to increase speed. All block memory used.

Table 2.2: Handel-C vs VHDL Resources

It should be noted that the development time needed for the Handel-C implementation is 1/3 to 1/4 that of the Verilog implementation.

2.2.1 Random Number Generators (RNG)

In many computationally intensive algorithms there is a need for good uniform random number generators. Such algorithms include back-propagation Neural Networks [Hayk99], which generate random initial weights, Genetic Algorithm [Mich94], which generate initial population and random crossover points, and Simulated Annealing [Kirk83], which generate initial starting solutions and random neighbourhood moves. These generators play a huge role in the success of these algorithm. In ISO-C programs, the random numbers that are generated are based on the following equation

$$I_{j+1} = ((aI_j + c)\ mod\ m)\ \text{[Pres92]}$$

This equation involves implementing multiplication, addition and modulus into hardware, which involves several resources as well as long delay times. A modification to this equation was proposed in [Pres92]. In this version of the RNG, called "an Even Quicker Generator" (EQG), if random numbers with $m = 2^{32}$ are needed and the size of the register holding the value is 32 bits wide, there is no need for the *mod m*.

A more common Random Number Generator implemented in hardware is the Linear Feedback Shift Register (LFSR) [Grah96, Gurw03, Mart01]. The advantage of this type of implementation is based on the relatively small amount of resources needed to realize this algorithm with negligible delays. Although this algorithm can produce uniform numbers, the algorithm introduces several problems. One of the main drawbacks is that there is a 50% probability of predicting the next random number [Mart02b]. The next value can be predicted to be either $v/2$ or $v/2+2^{n-1}$. One possible approach to solve the problem of predictability of the LFSR is to implement multiple LFSR with different initial seeds and taking one bit from each result to form the random number [Mart02b]. Martin's results showed that in implementing this method the system produced better results. Although this method may produce better random numbers, it is extremely difficult to prove the uniformity of the numbers and, therefore, it is not a good RNG for GA use. Another possible RNG is a Cellular Automata (CA) RNG [Mart02b]. This RNG consists of a circular array usually 32 bits wide. The next random number is generated with the following formula. For every bit c_t at time t, $c_{t+1} = ((west_t + c_t) \oplus east_t)$. Similar to the LFSR, there is a distinct pattern of numbers but is less predictable.

2.3 Reconfigurable Computing

Reconfigurable computing is a relatively new area of computing and is considered by many to be the future of conventional computing. This is attributed to its ability to deliver the flexibility of software while keeping the advantages of hardware. These systems can be considered as being a combination of both software and hardware. Their aim is to *fill the gap between hardware and software* [Comp00c, Comp00a] computing paradigms.

Reconfigurable devices allow designers to perform any logical hardware designs while allowing the hardware to be continuously modified and are, therefore, not restricted to a single implementation. These qualities contribute to the success of this relatively new technology, which survives on the belief that specific hardware designed algorithms should outperform general-purpose computers [Bish98].

There are several reasons for this assumption:

- General-purpose computers will always involve unwanted overhead which cause the use of unneeded clock cycles

- Hardware implementation can exploit parallelism and pipelining.

- Hardware implementations are designed specifically to accomplish one task, and are, therefore, optimized for such a task.

2.3.1 Hardware/Software Co-design

There are different ways of designing algorithms in hardware. One method is to implement a portion of the algorithms into hardware and the rest in software. The aim is to implement sections of code that involve large computation time into hardware to increase the execution performance. Although the hardware delivers better performance, it often results in less flexibility of the algorithm. By implementing algorithms in hardware, the designs are tailored to accomplish specific tasks. This limits the ability of the design to be modified for other tasks. There is a trade off between flexibility and performance as shown in Figure 2.6 and therefore both general-purpose processors and reconfigurable devices are used simultaneously. General-purpose processors are used to implement portions of code that require high flexibility, while reconfigurable devices aim to increase the execution of bottlenecks within the system. In this approach it is important to identify the main bottlenecks of the system and if it would be beneficial to implement these sections of code into hardware. The bottlenecks can be determined through profiling a software implementation of the algorithm, which identifies the portions of code that demand the majority of processing time.

The use of Amdahl's law, equation 2.1, aids in determining whether the overall algorithm will achieve beneficial performance improvement by implementing the bottleneck in hardware. This equation determines the effect on the overall system by optimizing a small section of code and can be used to justify the hardware/software co-design approach. Although many portions of code can be optimized for better performance, if this performance gained has little significant affect

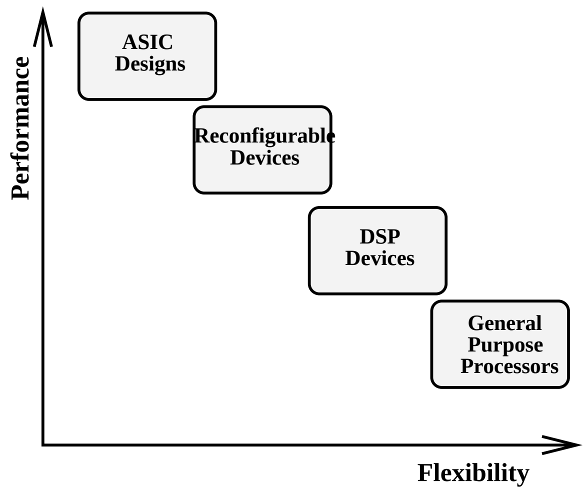


Figure 2.6: Performance vs Flexibility of today's hardware

on the overall algorithm, then implementing that portion of the design in hardware cannot be justified.

$$Speedup_{overall} = \frac{T_{old}}{T_{New}} = \frac{1}{(1-q)+\frac{q}{p}} \tag{2.1}$$

In calculating the algorithms speedup, "q" can be considered as the fraction of the algorithm that is implemented in hardware and "p" as the increase in performance of the hardware over the software. This equation assumes a linear speedup, meaning that with n processers it would take $\frac{1}{n}$ the amount of time needed by a single processor [Nich03].

2.3.2 Reconfigurable Algorithms

An alternative approach is based on implementing the entire algorithm into hardware. This method involves a complex circuit design of the algorithm which will often eliminate the flexibility of the algorithm. Although this method will most often generate a better performance system, in some cases it is not possible for the design to be implemented entirely in hardware. This could be a result of lack of space on hardware or memory management issues such as linked lists. Another issue that could arise is interconnect delay times. Often in implementing large algorithms into hardware the length of the interconnects are increased which places a limitation on the maximum frequency.

Determining the best design methodology requires understanding the requirements of the design. Usually a better performance will be achieved by implementing a complete algorithm onto one reconfigurable device. This is a result of the lack of overhead, having the hardware optimized for a specific task, and hardware's ability to exploit parallelism and pipelining. However, if there is a need for flexibility or it is not possible to implement the whole algorithm due to its complexity, then designing a Hardware/Software co-design system will result in minor performance increase.

2.4 VLSI CAD Tools

As technology advances, enabling the integration of billions of transistors onto a single die, the process of designing these circuits becomes much more complex. This complexity cannot be handled easily and therefore it is relatively impossible for hardware designers to design large circuits without the aid of advanced computer algorithms. One of the most important factors in VLSI design is to limit the delay within a circuit, allowing for higher clock frequencies. As shown in Figure 2.7, as transistors shrink in size ($< 1\mu m$) the interconnect delay (the connection between transistors) becomes a dominate factor over the gate delay. As the number of transistors increase, efforts to minimize the amount of interconnect become an impossible challenge for designers without the aid of Computer Aided Design (CAD) tools. There are numerous CAD tools developed to aid designers in implementing many of the complex tasks of the circuit layout process. These include:

1. Circuit Partitioning
2. Circuit Placement
3. Floor-planning and Macro-cell placement
4. Circuit Routing
5. Array-Based Layouts
6. FPGA Routing

Even with the aid of high performance computers, it is almost impossible to solve these tasks due to their high complexity. Accordingly, heuristic techniques are used in an attempt to generate good solutions in reasonable time. In the past, several heuristics were used to solve CAD problems, including Genetic Algorithms [Coho03], Simulated Annealing [Mall88, Baza99], Tabu Search [Arei93, Arei94] and Local Search [Fidu88, Kern70]. Each technique has a different flavor and characteristics. In order to improve upon solution quality, meta-heuristic techniques are

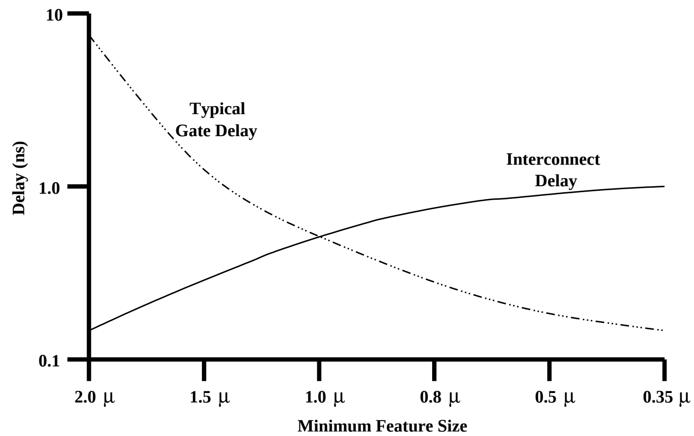


Figure 2.7: Interconnect delay vs Gate delay [Kang03]

often developed to benefit from the qualities of multiple search techniques. One of these meta-heuristics is Memetic Algorithms that are based on a combination of Genetic Algorithm and Local Search techniques. The Genetic Algorithm plays the role of effectively exploring the solution space while the Local Search algorithm is used to fine-tune the solution to its optimum/sub-optimum solution.

The aim of this research is to investigate the performance advantages of implementing a Memetic algorithm onto an FPGA platform aimed at solving the Circuit Partitioning problem.

2.4.1 Circuit Partitioning

Circuit partitioning (CP) is an important task in VLSI design, ensuring that there is minimum amount of interaction between partitions (blocks) of a circuit. In today's technology, the size of interconnect delay is associated with the length and number of interconnection wires. Minimizing the inter-partition communication will reduce the number of wires between partitions and, in effect, reduce delay times.

The main objective of circuit partitioning is to divide a circuit into two or more partitions while attempting to minimize the number of cut nets and still maintain a balance in the number of modules in each partition. Figure 2.8 illustrates how swapping of modules between the partitions can decrease the number of cut nets.

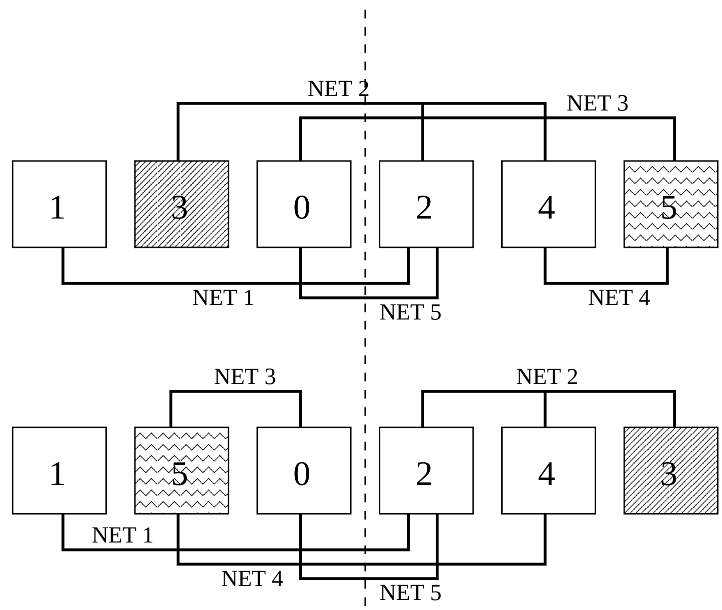


Figure 2.8: Example of Circuit Partitioning

Mathematical Formulation

The following is the standard mathematical formulation for a two block circuit partitioning problem [Arei00]:

We define:

$$x_{ik} = \begin{cases} 1 & \text{if module } i \text{ is placed in block } k \\ 0 & \text{otherwise} \end{cases}$$

$$y_{jk} = \begin{cases} 1 & \text{if net } j \text{ is placed in block } k \\ 0 & \text{otherwise} \end{cases}$$

$$m = \text{number of modules}$$

$$n = \text{number of nets}$$

$$q = \tfrac{1}{2} \text{ of allowable difference in modules}$$

The circuit partitioning problem can, therefore, be formulated as following:

$$Max \quad \sum_{j=1}^{n} \sum_{k=1}^{2} y_{jk} \tag{2.2}$$

(Maximize uncut nets)

Subject to

(i) Module placement constraints (A module belongs to a single block):

$$\sum_{k=1}^{2} x_{ik} = 1 \ \ \forall_i = 1, 2, \ldots, m$$

(ii) Block size constraints (Difference between each block less than q):

$$\frac{m-q}{2} \leq \sum_{i=1}^{m} x_{ik} \leq \frac{m+q}{2}, \ \ \forall k \in \{1, 2\}$$

(iii) Netlist constraints:

$$y_{jk} \leq x_{ik},\ where \quad \begin{array}{l} 1 \leq j \leq n \\ k \in \{1,2\} \\ i \in Net\ j \end{array}$$

(iv) 0-1 constraints:

$$\begin{array}{l} x_{ik} \in \{0,1\},\ \ 1 \leq i \leq m \\ y_{jk} \in \{0,1\},\ \ 1 \leq j \leq n \\ for\ k \in \{1,2\} \end{array}$$

These constraints are placed on the optimization algorithms to ensure feasible solutions.

2.4.2 Benchmarks

For this work, eight benchmarks of variable sizes were chosen and used to validate the architectures proposed in chapter 3 and 4. The benchmarks range in size from 24 to 3014 cells and 32 to 3029 nets. Statistics of these benchmarks are presented in Table 2.3. The Cell Degree indicates the number of nets that are connected to a single cell. The Net Size is the number of cells connected to a single net. The Chip1 and Chip4 benchmarks are from work presented in [Fidu82]. The remaining benchmarks can be found in the "*1990 MCNC LAYOUT BENCHMARK SET*" [MCNC90]. The connectivity of the cells and nets in each benchmarks are presented in Table 2.4.

Benchmark	Cells	Nets	Cell Degree			Net Size		
			MAX	μ	σ	MAX	μ	σ
pcb1.dat	24	32	7	3.50	1.35	8	2.63	1.19
frac.dat	149	147	7	3.10	1.65	17	3.14	2.26
chip4.dat	224	221	5	2.34	1.13	6	2.58	0.99
chip1.dat	300	294	6	2.82	1.15	14	2.87	1.39
prim1.dat	833	902	9	3.49	1.29	18	3.22	2.58
struct.dat	1952	1920	4	2.8	0.67	17	2.85	1.90
ind1.dat	2271	2192	10	3.41	1.14	318	3.53	9.00
prim2.dat	3014	3029	9	3.72	1.55	37	3.70	3.82

Table 2.3: Benchmark Statistics

Benchmark	Nets connected to a Cell (%)						Cells connected to a Net (%)				
	1	2	3	4	5	>5	2	3	4	5	>5
pcb1.dat	0.00	29.17	25.00	25.00	12.50	8.34	62.50	28.12	3.12	3.12	3.12
frac.dat	16.11	27.52	24.16	8.05	16.11	8.05	47.62	29.93	9.52	8.84	4.08
chip4.dat	22.95	46.72	6.97	20.08	3.28	0.00	64.25	23.98	4.52	3.62	3.62
chip1.dat	11.33	36.67	16.67	30.00	5.00	0.33	55.10	24.15	8.50	8.84	3.40
prim1.dat	5.76	17.41	24.61	32.77	16.09	3.36	54.77	26.16	6.87	2.88	9.32
struct.dat	3.28	24.59	24.59	11.42	0.00	0.00	38.39	59.95	0	0	1.67
ind1.dat	1.45	21.27	35.31	20.48	20.96	0.52	65.01	15.78	5.47	2.97	10.77
prim2.dat	1.43	15.03	42.00	17.22	13.34	10.98	60.58	12.05	6.70	6.34	14.33

Table 2.4: Statistical Connectivity of Benchmarks

2.4.3 Genetic Algorithms

A Genetic Algorithm (GA) is a population based heuristic technique that mimics the biological reproductive system and used to solve search and optimization problems [Glov95, Beas93b, Whit94]. The goal of a Genetic Algorithm is to maximize/minimize a specific objective function, also known as fitness. As in human evolution, the population of the Genetic Algorithm evolves over numerous generations. It operates on the theory of survival of the fittest where the fittest survive and the weakest die off [Beas93b].

In mimicking the reproductive system, GA makes a new generation by selecting two mating parents from the population to generate offspring. These parents often compete based on their performance with other parents for the chance to reproduce. In traditional Genetic Algorithms, the number of chromosomes in the population remains static. This often means that in the reproductive process, parents with low fitness (low performance) will produce very few or no children, whereas parents with high fitness will produce several offspring. Eventually, over a number of generations, the parents with low performing genes sequences will die off, leaving only the stronger parents and their children to survive.

In trying to mimic the reproductive system, GA follows the steps shown in Figure 2.9 [Mitc96, Mich94].

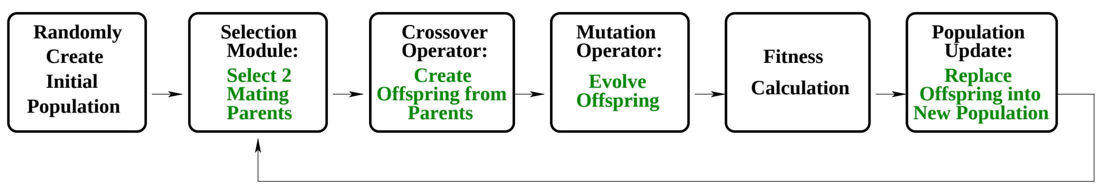


Figure 2.9: Genetic Algorithm mating process

Selection Module

The selection process attempts to select two different parents with high fitness to mate. The purpose of the selection process is to obtain fit individuals from the population, in hope that the offspring produced will have better fitness values than their parents. The more popular selection techniques are Roulette Wheel and Tournament Selection.

In Roulette Wheel selection strategy, each individual in the population is assigned a slice of the wheel that is proportional to the individuals fitness, $p_i =$

$\frac{fitness_i}{\sum_j Fitness_j}$ [Glov95]. Individuals with higher fitness get a larger slice of the wheel whereas individuals with lower fitness get a smaller share. In spinning the wheel, probability states that the wheel should land more frequently on good fitness individuals than individuals with less fitness. This implies that, over time, less fit individuals should die off leading to a highly fit population.

Tournament Selection performs a competition among individuals to become mating partners. Two individuals are randomly selected from the population and allowed to compete. The individual with the higher fitness value of the two competitors wins the right to mate with another individual from the population. This selection strategy allows for diversity by enabling individuals with low fitness values to enter into the competition.

Crossover Operator

Crossover is a natural process where two parents create offspring by combining some genes from each parent. As in human evolution, the crossover technique attempts to combine good gene sequences from each parent in hope that the resulting chromosome will have a higher fitness than its parents. This process of gene exchanging is essential and characterizes Genetic Algorithms [Glov95]. The crossover process often occurs with a high probability [Mazu99].

There are numerous crossover techniques for binary encoding. The three most common techniques are one-point crossover, two-point crossover and uniform crossover.

- One-point crossover is the traditional method for performing the crossover operation [Beas93b]. This technique involves randomly selecting a point within the chromosome to generate a "Head" and a "Tail" of the chromosome. The children are created by combining the "Head" from the one parent and the "Tail" from the other. Figure 2.10 demonstrates a one-point crossover operator.

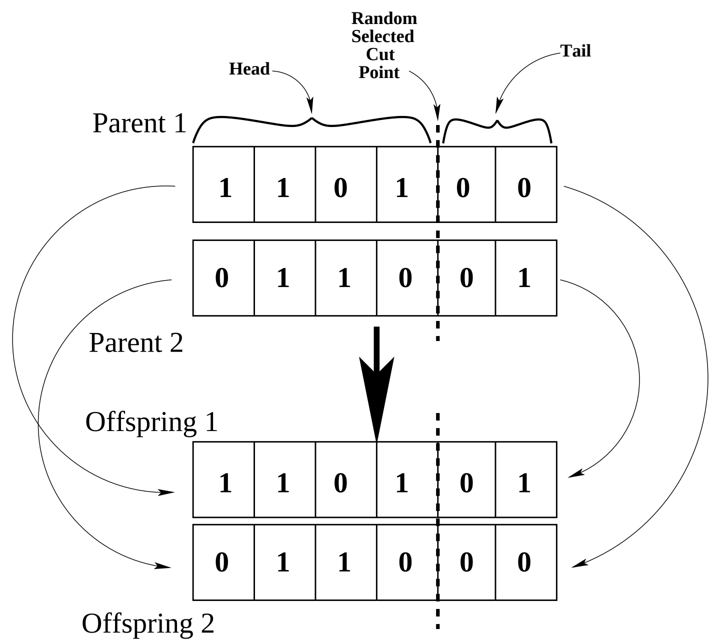


Figure 2.10: One-Point Crossover

- Two-point crossover is similar to the one-point crossover operator. Two points in the chromosome are selected. In this process, the data exchanged is the gene sequence between the two selected points.

- Uniform Crossover follows a different philosophy for generating children. In this process, each gene in the chromosome is randomly selected from one of two parents to form children. Figure 2.11 demonstrates a uniform crossover technique which uses a random mask to select genes from each parent. A '1' value in the mask will cause that gene to be selected from $parent_1$ and a '0'

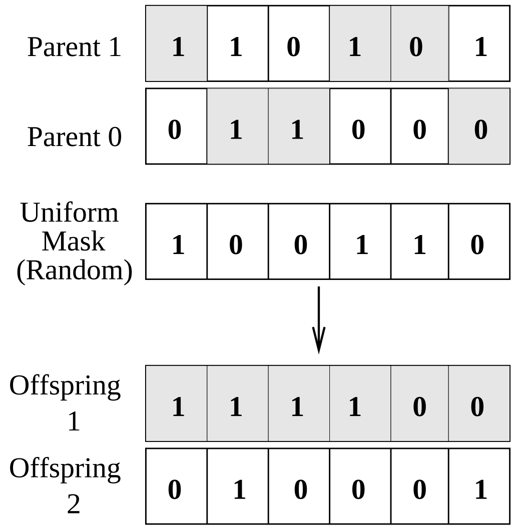


Figure 2.11: Uniform Crossover

value will cause the gene to be selected from $parent_0$. The second child can be created by selecting the alternate genes from the first child.

Mutation operator

Mutation is the process of introducing a random element that creates new individuals by a small change in gene sequence. It allows for diversity between the different individuals within the population and allows for the exploring of the solution space in attempt to avoid stagnation. In Mutation, each gene in the chromosome is tested with a low probability to see if that gene should be altered. Having a high Mutation rate may result in a “Random Search”, where the offspring have little relationship with the parents since a good portion of the genes have been altered [Beas93b].

Fitness Calculation

This process is problem dependent and is the only process that needs to be modified in order to handle different optimization problems. Fitness Calculation is the process of evaluating individuals based on their objective function value. Each individual in the population is given a numerical measure of merit which demonstrates its superiority to other individuals. This fitness value can be considered the same as strength or intelligence in humans beings.

For the circuit partitioning problem, if the circuit is represented by 300 nets, of which 100 nets are cut for a given chromosome, the fitness can be calculated as:

$$Total\ nets - Cut\ nets = 200$$

Replacement Strategy

This is the process of replacing the old population with the newly generated population in an attempt to move the higher fitness individuals into the new population while also maintaining diversity.

There are several techniques proposed by Smith et al. [Smit98] for population update. Many of these replacement techniques are computationally intensive, searching the fitness values to replace low fitness individuals. From these techniques, there are four simple and efficient population update methods.

1. *Generational GA* - all the parents are replaced by the children, and it is the responsibility of the crossover and mutation to preserve good solutions [Smit98].

2. *Tournament replacement*- this technique is applied on the parents and offspring, selecting two individuals to join the new population.

3. *Best Child and Parent* - the best parent and the best offspring are placed in the population.

4. *Best Survive* - the two best individuals from the parents and/or the offsprings are replaced into the population.

Parameter Tuning

In Genetic Algorithms, several parameters need to be tuned to obtain good solutions [Sitk95]. These parameters are crossover rate, mutation rate and population size. Each parameter affects the GA differently. The population size determines the size of the search space. If the population size is limited in size this in turn limits the exploration capability. A large crossover rate increases the creation of new offsprings, as well as causing disruption of strings. Mutation rate assists in escaping local minimums but can be considered as a random walk if the value is too large. A successful GA results often comes from finding a good balance of these parameters.

2.4.4 Local Search

Local search methods are iterative algorithms that seek to enhance the solution by stepwise improvements. These heuristic techniques, although attempting to improve solution quality, often result in suboptimal solutions by getting trapped in local minima. The simplest form of local search attempts to swap elements in combinatorial optimization problems.

In Figure 2.12, Abramson et.al. proposed a generic template for local search [Abra97]. The following units are defined in the structure:

- a unit for storing the current solution
- a unit for storing the new solution
- an update unit
- a change-in-cost generator
- a neighborhood generator
- a unit for applying a move

The goal of this template is to define a generic structure that can be applied to different problems.

Neighbourhood move for VLSI Circuit Partitioning

Local Search is a simple technique that follows a basic search template presented in Figure 2.12. Since Local Search attempts to make gradual improvements to the objective function, for VLSI Circuit partitioning the aim is maximize the number of uncut nets. For this reason, a net representation is developed to move entire nets in each block. To search all neighbors of a solution, an attempt to move each net completely within a block is performed. The move with the highest object value is, therefore, chosen as a candiadate.

Within this local search algorithm, the most crucial and complex issue is the determination of other nets affected by a certain neighbourhood move. Once a net

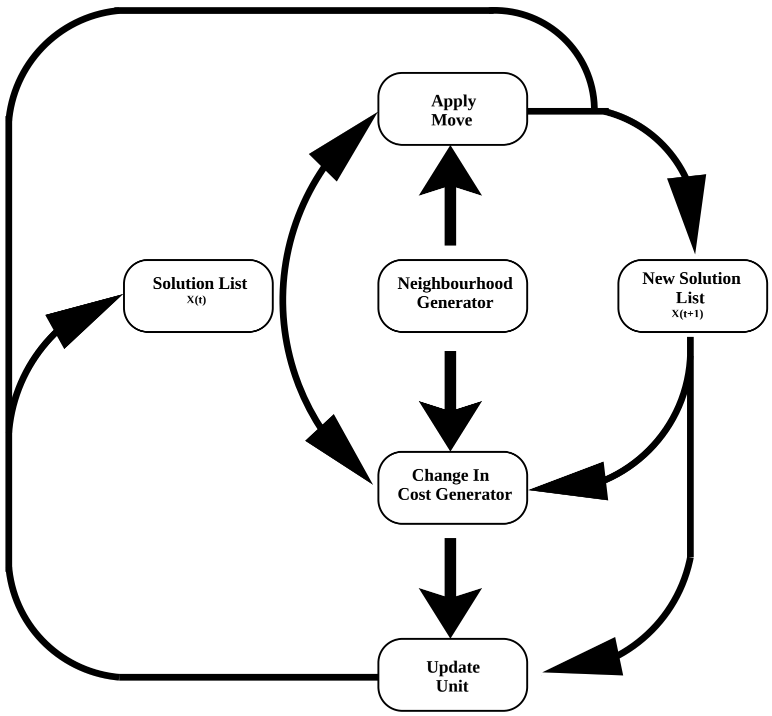


Figure 2.12: An architecture for local search proposed by Abramson et.al. [Abra97]

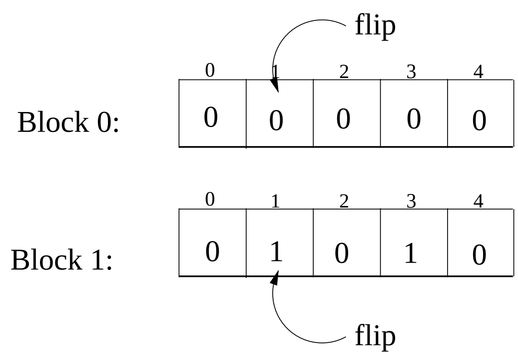


Figure 2.13: Search neighbors by flipping all nets one by one

has been moved exclusively into a block, the resulting solution might be infeasible due to balance constraints. Therefore, it is necessary to re-check all nets that are connected to any modules which have been moved. This process consists of 3 nested loops. The first is used to determine the modules affected by moving the original net. The second loop determines the nets affected by moving a certain module. The third loop is performed to determine if a net has moved entirely in/out of a block.

If we wish to move net x_i into a specific block, the following process can be used to verify the feasibility of the data:

```
Check x for 1 to m modules
  if '1' exists
    Check i for 1 to n nets
      if one exists
        Check j for 1 to m modules
```

Since there are three layers of loops for every net being modified, the complexity of the feasibility check is $O(m^2n)$, where m is total number of modules, and n is the total number of nets. The total complexity of the process of determining the best neighbor is $O(m^2n^2)$.

2.5 Hardware Accelerators for CAD

Hardware accelerators have come a long way over the past few decades. In the 1980's, the majority of hardware accelerators were developed for fixed CAD algorithms often dealing with Logic simulation of circuits consisting of less than 100,000 gates [CD88, Ambl89]. As circuits increased in complexity there was a greater need for high performance CAD tools. In the past, hardware accelerators were developed as ASIC devices with little flexibility. With the introduction of reconfigurable computing, a new option has arisen for hardware accelerators. These new devices allow designers to use parallelism and pipelining as well as logical operations to enhance the performance of the algorithms, while still maintaining flexibility in their designs. Since the mid 90's, there have been a few hardware implementations of CAD tools on reconfigurable platforms.

One of the main hardware implementation of CAD tools is the Boolean Satisfiability problem (SAT). The SAT Problem is an NP-complete problem [De J89, Plat98], and is computationally intensive for general purpose processors due to its huge search area and need to perform extensive logic operations. The aim of the SAT problem is to find a boolean solution that satisfies a given logical expression in Product of Sum format.

$$(C_1 \vee C_2 \vee C_3) \wedge (......) \wedge (C_{n-2} \vee C_{n-1} \vee C_n)$$

The Boolean Satisfiability Problem is used in a couple of different areas for CAD tools, such as Test Generation, Logical Verification and Timing Analysis [Zhon98a]. Similar to other NP-complete problems, heuristic techniques must be used to generate acceptable solutions in reasonable time. Although heuristics can generate fast solutions, these solutions may fail to prove satisfiability [Plat98].

With the ability to generate logical functions and exploit parallelism, reconfigurable computing has the ability to dramatically increase the performance of the SAT problem over software implementations. This is due to easy implementation of the logical function, often resulting in one clock cycle per calculation. Most hardware accelerators for the SAT problem in the past [Plat98, Zhon98a, Zhon98b, Hama97] achieved extensive speedups over software implementations.

Other hardware implementation of CAD tools include:

- Chan et al. proposed hardware assisted designs that use fine-grained parallelism to aid in increasing the performance of a PathFinder router algorithm [Chan97]
- Wrighton et al. proposed a hardware assisted systolic approach to Simulated Annealing for FPGA placements [Wrig03]
- Luo et al. implemented a scanline algorithm for Design Rule Checking (DRC) [Luo99]

2.6 Hardware Based Genetic Algorithms

A hardware based GA is an appealing option to solve many optimization problems due to its speed and efficiency. With the introduction of FPGAs, GA algorithms can be easily modified to handle many different problems. In the past decade, there has been significant activity in the development of Genetic Algorithms by hardware implementations which has contributed to the advance in FPGA technology.

2.6.1 Hardware/Software Co-Design Approaches

One of the earliest hardware Genetic Algorithm implementations was by Stikoff et al. in 1995 [Sitk95]. A Hardware/Software co-design system was introduced for minimizing communication between numerous FPGA chips. The aim was to develop a system that would overcome the bottlenecks of the algorithm by implementing several functionalities into hardware. Once the software implementation of the algorithm was developed, it was determined through profiling that 84% of the execution time was spent calculating the internal-net fitness values which caused them to implement this portion of code into hardware. In comparing the software and co-design approaches, Sitkoff found that by implementing the fitness calculation into hardware, there was an improvement in processing time of a factor of three over the software design executing on a SUN SPARCstation 20 running at 60 MHz.

Koza et al. [Koza97] also found that the burden placed on the algorithm was implemented in the fitness calculation function. The system consisted of a co-design approach which incorporated the fitness calculation in a Xinlix XV6216 FPGA. The idea of the design is to use a host computer to do all evolutionary computations and send the population to the FPGA for evaluation. There was no performance analysis for this design.

2.6.2 Pure Hardware Genetic Algorithm Implementations

In 1995, Scott et al. [Scot95] developed a complete Genetic algorithm in hardware for simple linear equations using VHDL. The proposed architecture was spanned across multiple FPGAs operating at a maximum clock frequency of 8MHz. Scott et al. average speedup for the linear equations was 17 times that of the software implementation running on a Silicon Graphics 4D/440 with four MIPS R3000 CPUs each running at 33MHz. The bottleneck of the system was found to occur in the population sequencer/Selection module and the fitness module. In implementing two Selection routines in parallel, the algorithm had a slight increase in speed, but was still limited by the fitness function. A few improvements were suggested to the algorithm:

1. Increase parallelization of the selection modules
2. Use memory configurations which support read and write in one clock cycle
3. Merge the population sequencer with the memory interface module
4. Parallelize and pipeline the selection-crossover-fitness modules

A modified Genetic Algorithm was also introduced by Aporntewan et al. [Apor01]. This algorithm is a compact version of a Genetic Algorithm and does not follow the normal convention of a traditional Genetic Algorithm [Beas93a]. [Apor01] claim that in using this method, they can achieve 1000x speedups over software versions, implemented on an Ultra Sparc 2 operating at 200 MHz. These speedups are achieved through the simplicity of this design: using only adders, subtractor and comparators. Although the Compact GA is efficient, it only simulates the tournament selection and uniform crossover and therefore cannot replace Simple GA's [Apor01].

In 1995, an architecture implementing a Genetic Algorithm was introduced for the Travelling Salesman Problem (TSP) using reconfigurable hardware [Grah95]. Graham et al. developed their architecture on a two-board Splash 2 system, consisting of 34 Xilinx 4010s FPGAs and having a maximum clock frequency of 11 MHz. The algorithm was pipelined between 4 FPGA modules to achieve its performance of 7-10 times that of the software implementation running on a HP PA-RISC workstation running at 125 MHz [Grah96]. In their analysis of the system, they highlight the following factors that contributed to the success of the design:

1. Fine-Grain Parallelism:

   High parallelism within the selection routine is estimated to generate a speedup of 38 times that of the selection routine in software.

2. Address, Branch, and Function Call Overhead:

   In analyzing the assembly code generated from the software implementation, it was found that two-thirds of the instructions were overhead (branching, address lookups, etc.).

3. Coarse-Grain Parallelism:

   The parallel execution of four FPGAs attributed to a factor of 1.5 to 2 times the systems speedup.

4. Random Number Generator (RNG):

   In analyzing the crossover and mutation routines, generating random numbers attributed to 80% of the instructions. Since only a small portion of the time is spent in the selection routine, implementing a more efficient RNG in hardware can aid in only about 10% increase in performance.

In 2001, Shacklefor et al. [Shac01] introduced a steady state genetic algorithm for implementation on a FPGA for the set covering problem and protein folding problem. The proposed architecture involved a 6 stage pipeline with slight modifications to the standard GA process. The hardware implementation outperformed a C program running on a 366 MHz pentium CPU by 320 times. It was determined that the limiting factor of the performance of the algorithm is the throughput of the cost modules. Increasing the number of cost modules running in parallel would dramatically increase the processing performance, as seen in Figure 2.14.

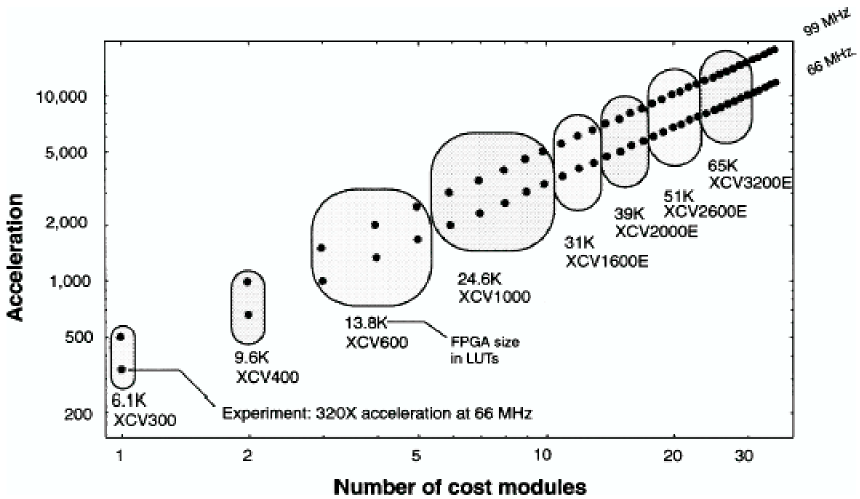


Figure 2.14: Protein folding problem: performance scaling as a function of FPGA size [Shac01]

In 2003, yet another complete VHDL implementation of a Genetic algorithm was developed [Gurw03]. The design identified the same findings of Stikoff et al., described in section 2.6.1, where the majority of the execution time was found in calculating the fitness function. The design was implemented on a Rapid Prototyping Platform containing a Virtex-E XCV2000e FPGA device with a maximum clock frequency of 40 MHz which led to over 40 times speedup than the software algorithm running on a SUN ULTRA10 at 440 MHz. This speed-up was attributed to pipelining and parallelization. The main limitation in this design was the size of the available memory. The proposed design used dual-input/output block rams, limiting the architecture to benchmarks of size 32 nets.

2.6.3 Synthesized Hardware Genetic Algorithms

Megson et al. [Megs98] proposed implementing Genetic Algorithms using Systolic Arrays. All the proposed ideas were developed in C and were not tested or simulated for performance. [Megs98] claim that in using the systolic approach the design would be easily implemented, would be modular, and easily expandable to any problem size and allows for massively parallel architecture.

Perkins et al. [Perk00] successfully designed and synthesized a Genetic Algorithm for non-trivial 1-D signal reconstruction. Although the design was not implemented on hardware but through simulation and synthesis the hardware design achieved 1000 times speedup over a C implementation for a small problem[1]. The increase in performance was contributed to the following reasons:

1. Efficient hardware pipelined fitness evaluation
2. Evaluation of an entire population of individuals in parallel
3. Elimination of slow off-chip communication (off chip memory).

Ramamurthy et al. [Rama] described a framework for a VLSI architecture that incorporates a microprocessor to perform the fitness calculations. The framework defines the basic functionalities of a Genetic Algorithm but is restricted to a population size of 16 two byte members and solving single variables equations.

[1]Authors fail to mention the processor speed of the computer used

2.6.4 High-Level Hardware Implementation

Martin introduced a Genetic Programming architecture design based on Handel-C [Mart01]. In the design, two simple problems were used: a regression problem ($x = a + 2b$) and the 2-bit XOR boolean logic problem. The design was broken down into three levels of parallelization to gain performance:

- *Intrinsic Parallelism* - Which exploits the parallelism of simple statements throughout the entire algorithm.
- *Geometric Parallelism* - Involves partitioning a task into smaller units to be copied many times to increase performance. In this design a master and numerous slaves operated in parallel. For this algorithm, the master stored the population and the slaves evaluated the fitness values of individuals.
- *Asynchronous Parallelism* - Involves two or more processes that operate independent of each other with little communication. The parallelism occurred with the random number generator, which continuously generated random numbers used as needed.

Tables 2.5 and 2.6 show the results of implementing two algorithms on an RC1000 development board consisting of a Xilinx Virtex-E XVB2000e-6 in comparison to a Power-PC running at 200 MHz, with a population size of 8.

Martin [Mart02a] further improved the proposed design to handle larger problem sets (Artificial Ant Problem) and incorporated a pipelined architecture. Unlike the previous architecture, this design utilized off-chip memory to store the population and fitness values. In comparing the results of the XOR problem, the pipelined

Measurements	Power-PC simulation	Handel-C Single Fitness	Handel-C 4 Parallel Fitness
Cycles	16,612,624	351,178	188,857
Clock Frequency	200 MHz	25 MHz	19 MHz
Speedup (Cycles)	1	47	88
Speedup (Time)	1	6	8

Table 2.5: Results of running the regression Problem[Mart01]

Measurements	Power-PC simulation	Handel-C Single Fitness	Handel-C 4 Parallel Fitness
Cycles	27,785,750	715,506	384,862
Clock Frequency	200 MHz	22 MHz	18 MHz
Speedup (Cycles)	1	38	72
Speedup (Time)	1	4	6

Table 2.6: Results of running the XOR Problem[Mart01]

architecture produced 8 times faster results than the original design while operating at twice the frequency. For the Artificial Ant problem using 32 parallel fitness evaluations and operating at 37 MHz, the new architecture achieved speedups of nearly 100 times that of software running on a PowerPC at 200MHz. Due to the number of parallel Fitness evaluations, the design required nearly 80% of the FPGA resources and 4 hours to compile the design using a 1.4 GHz Athlon computer.

From this improved design [Mart02a] concluded that:

- The parallel fitness evaluations were only effective when the problems were large enough that the fitness became the bottleneck.
- For a fixed problem required to be executed many times, a hardware architecture with parallel fitness evaluations can reduce time by two orders of

magnitude. For problems that are not fixed, a large investment in time is required to modify and compile the design.

- Although Handel-C is beneficial to software engineers with limited hardware experience; knowledge of how hardware works is still required to achieve acceptable speedup from the design.

2.7 Summary

This chapter presented an overview of reconfigurable computing, VLSI CAD tools and hardware implementations of Genetic Algorithms. Literature review indicates that the use of reconfigurable computing technology for hardware accelerators has increased software algorithms performance by orders of magnitude and is desirable for many applications. It is shown that there has been extensive work on hardware Genetic Algorithm accelerators with few applied to VLSI CAD tools. Although Genetic Algorithms are known as effective methods for exploring the solution space, they are inefficient at fine tuning the search without the aid of local search algorithms. The ability of Memetic algorithms to effectively explore and exploit the solution space qualify them as good candidates to solve NP-complete problems. Therefore, this thesis will attempt to implement a Memetic algorithm in hardware for solving the circuit partitioning problem.

Chapter 3

A Genetic Algorithm Processer

Genetic Algorithms are search techniques based on the biological reproductive process, following the theory of natural selection[Reev02]. The aim is that through reproduction and mutation, good gene sequences will evolve and become stronger while weak genes will die off and get eliminated from the population. They are considered robust algorithms with the ability to solve many complex NP-Hard problems. They tend to explore the solution space through the use of a population of various unique solutions, while placing little emphasis on fine tuning its results.

3.1 Hardware Design

In designing a Genetic Algorithm processor for the VLSI Circuit Partitioning the aim is to exploit the natural parallelism that is inherent within Genetic Algorithms. A parallel flow of reproduction process is shown in Figure 3.1 which demonstrates how the mutation, repair and fitness operations can be performed on each offspring (generated by the crossover) in parallel. This flow also allows Genetic Algorithms to be implemented as pipelined architectures, allowing each component to operate independently and in parallel. The following section gives a general overview of the architecture design and specifications.

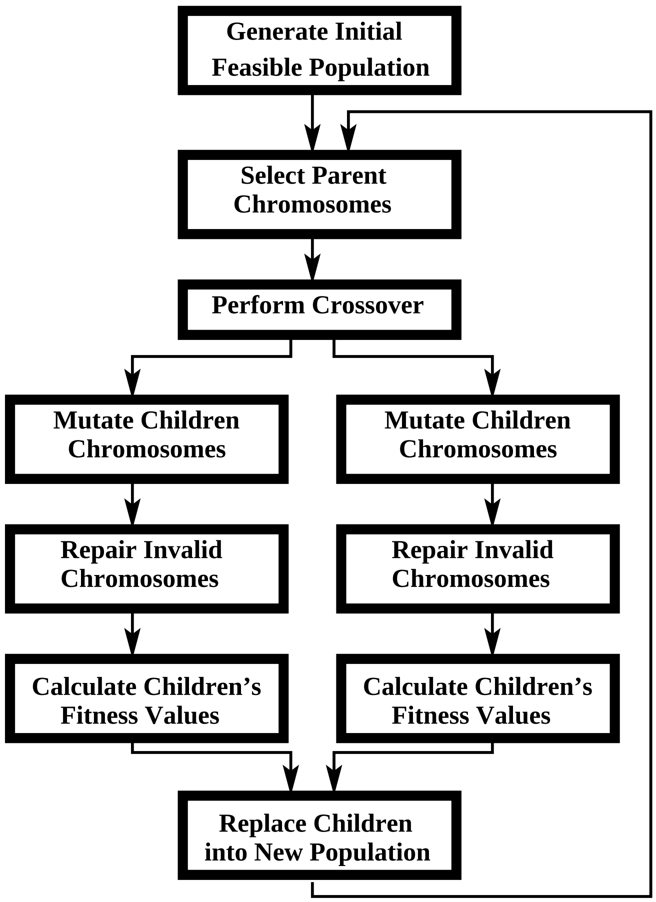


Figure 3.1: Parallel Flow Genetic Algorithm

3.1.1 Architecture Specifications and Constraints

The Genetic Algorithm Architecture should meet the following specifications and constraints:

- The architecture must be small enough to fit in common FPGA devices such as the RC1000 [Supp01] development platform.
- The architecture must be designed to allow enough flexibility to solve combinatorial optimization problems in general and not just circuit partitioning.
- The architecture should be able to handle all sizes of circuits and should be easily modified at compile time, allowing for different configurations.
- The architecture is not constrained by using internal Ram but can use external memory as well.
- The architecture must have user programable parameters (ie. Population Size, Crossover Rate, etc).

3.1.2 Genetic Algorithm Architecture Overview

The architecture is broken down into two independent parts, as shown in Figure 3.2. The first part is designed to initialize the Genetic Algorithm by developing a random initial population while the second part of the design performs reproduction. Both components are pipelined to decrease the execution time. A detailed description of execution and interface is presented in sections 3.2 and 3.3 respectively.

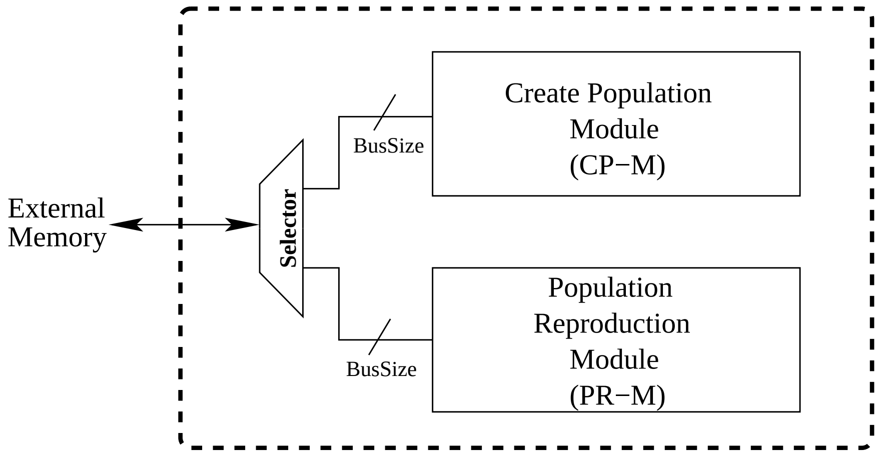


Figure 3.2: Genetic Algorithm Block Diagram

Data Representation

For this work, the data representation follows the same used by [Gurw03]. This representation uses binary values to show the connectivity of a net. A '1' indicates that a net is connected to the corresponding cell while a '0' indicates that the net is disconnected. An example of this Netlist representation is shown in Figure 3.3.

A circuit Netlist is a collection of data that is used to show the connectivity of all nets within the benchmark. In the Genetic Algorithm implementation, the Netlist representation is used to calculate the fitness value of each chromosome by determining the number of nets that are uncut (all cells attached to the nets lie within one partition). From Figure 3.3, it is clear that Net #1 is connected to $Cell_1$, $Cell_2$ and $Cell_4$.

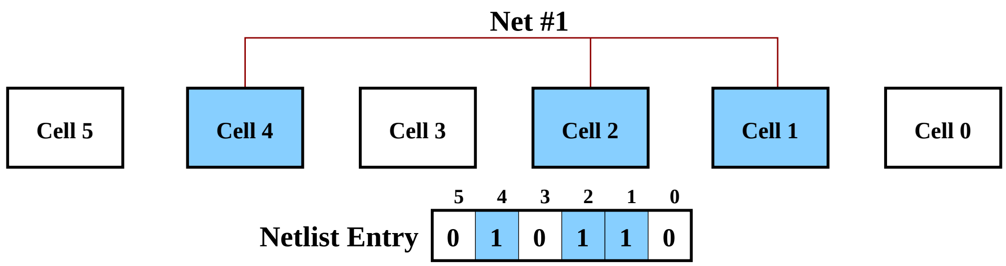


Figure 3.3: Netlist representation of a single net

Memory Organization

In storing the population and Netlist data into memory, a fixed number of data words is reserved to represent each chromosome or net. The fixed memory size must be a power of two (2^x) to allow for easy indexing of the memory. This constraint eliminates the need for multiplication offsets which significantly decreases the amount of resources needed. Figure 3.4 explains how a single memory entry consisting of 75-bits is stored within eight half-words of data. From the figure,

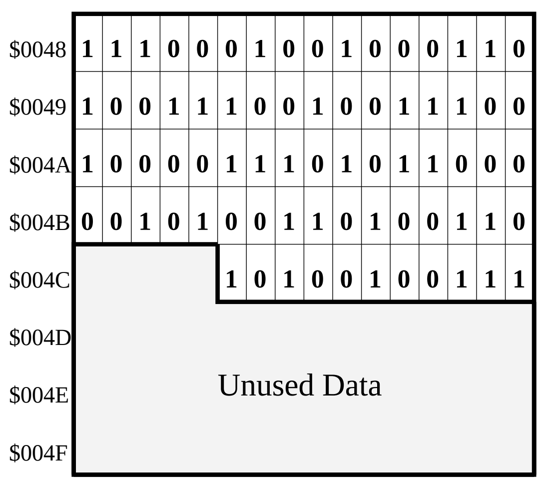


Figure 3.4: Example of Chromosome Data

it is evident that although eight half-words of information are used to store each chromosome, a little over half the allotted space is required and by using this storage format the remaining memory is wasted. The wasted space is necessary to achieve higher speeds from the system.

In indexing the Netlist or population memory the address value is divided into two parts, as illustrated in Figure 3.5. The upper address lines are used to index the starting position of the desired entry and the lower address lines indexing the desired information. The figure uses four bytes to represent each entry in the Netlist

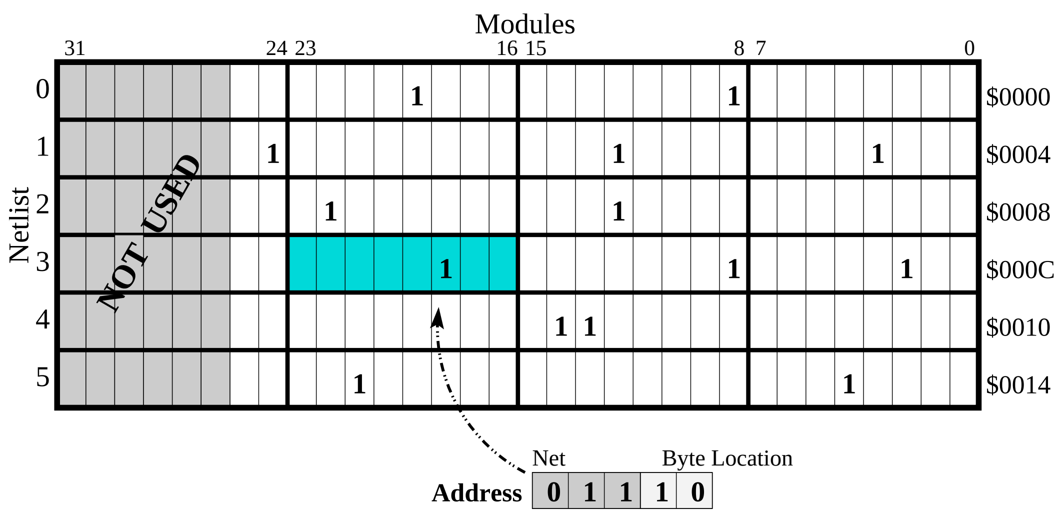


Figure 3.5: Example of Netlist Data

meaning that the lower two address lines are used to access individual bytes of data for the given net and the remaining address lines are used to index the location of the net in memory.

Constants

In order to meet the architecture specification of flexibility, system constants are used to adapt the algorithm to different FPGA configurations. These constant values are found in a header file of the code and can be changed preceding synthesis. The constants and their definitions are shown in Table 3.1 and are illustrated in Figure 3.6.

Constant Name	Description
AddrWidth	This constant holds the maximum number of address lines that can be used to retrieve data from memory
TotalAddress	This constant holds the size of memory modules. It is equivalent to $2^{AddrWidth}$
NetWidth	This constant holds the number of address memory bytes needed to represent a chromosome
NetSize	This constant holds the number of bytes reserved to store each chromosome. It is equivalent to $2^{NetWidth}$
DataWidth	This constant holds the data bus width as a power of 2
DataSize	This constant holds the size of the data bus. It is equivalent to $2^{DataWidth}$

Table 3.1: Handel-C constant definitions

Registers

To allow for user programmable parameters, registers have been introduced into the architecture to control the algorithm flexibility. These parameters are stored internally and are programmed through the memory. The definition of registers can be found in Table 3.2.

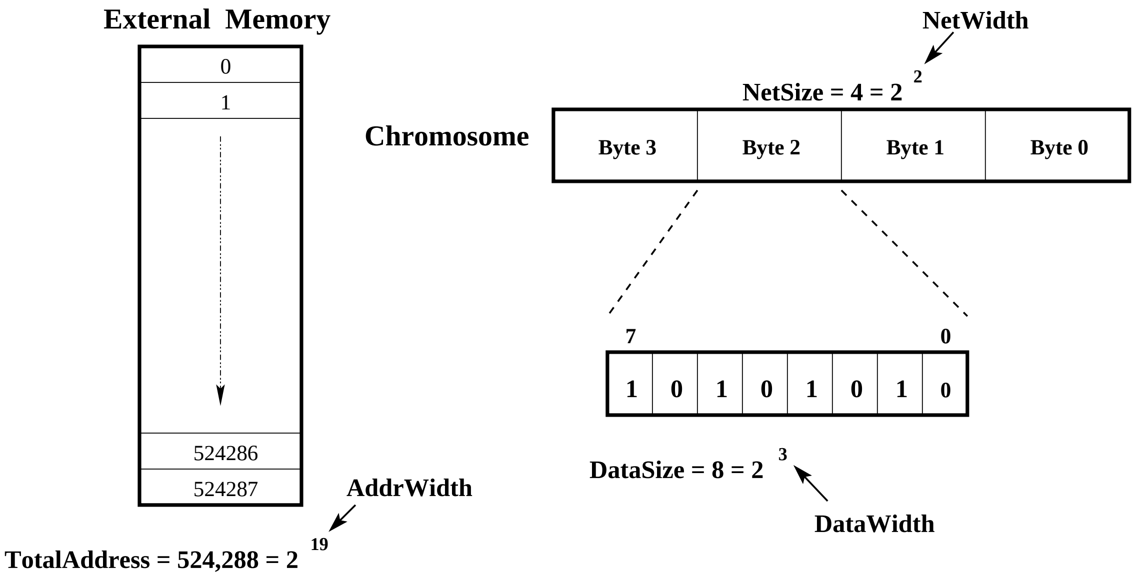


Figure 3.6: Handel-C constant definitions

RC1000 Limitations

The processor will be implemented on an Celoxica RC1000 development board [Supp01]. Even though the architecture constraints are met, the RC1000 board adds additional constraints summarized as follows:

1. Population Size : The population is limited to values of the power of 2 (2^2, 2^3, . . . , *etc*) and also limited by the size of memory available to hold the population.

$$\text{Allocated Memory} \geq \text{NetSize} \times \text{Population Size}$$

2. Netlist Memory : The limitation on the Netlist size stored in memory is:

$$\text{Available Netlist Memory} \geq \text{NetWidth} \times \text{NetNum}$$

Memory Location	Register Name	Register Size	Description
0x00	Net Number	16 bits	The number of nets within the Netlist data file. The maximum number of nets is $2^{16} - 1 = 65535$
0x01	Cell Number	16 bits	The number of cells inside each chromosome. The maximum number of cells is $2^{16} - 1 = 65535$
0x02	Allowable Block Difference	16 bits	The allowable difference between the number of cells in block 1 and the number of cells in block 0
0x03	Crossover Rate	16 bits	The probability of the two selected individuals mating and generating offspring
0x04	Mutation Rate	16 bits	The rate at which the offspring will be mutated
0x05	Population Size	16 bits	The size of the population (must be power of 2)
0x06	RNG Seed	DataWidth	The seed value that is used to initialize the Random Number Generator
0x07	Generation Size	16 bits	Number of generations that the GA will undergo

Table 3.2: Register Description

3. Population Memory : The limitation on the population memory size is:

$$\text{Available Population Memory} \geq \text{NetWidth} \times \text{PopSize}$$

4. Memory Data Bus : In order to correctly synthesize the design, the external memory data bus must be of 16-bits or greater. This is due to loading the 16-data into the registers.

RC1000 Memory Usage

In satisfying these constraints for the RC1000 board a minimum of three memory banks are needed to hold the required information. One bank is dedicated to hold the Netlist information. The other two banks are separated into new and old storage data, used to hold different working data for the system, as shown in Figure 3.7. These memory banks are divided up to hold the following information:

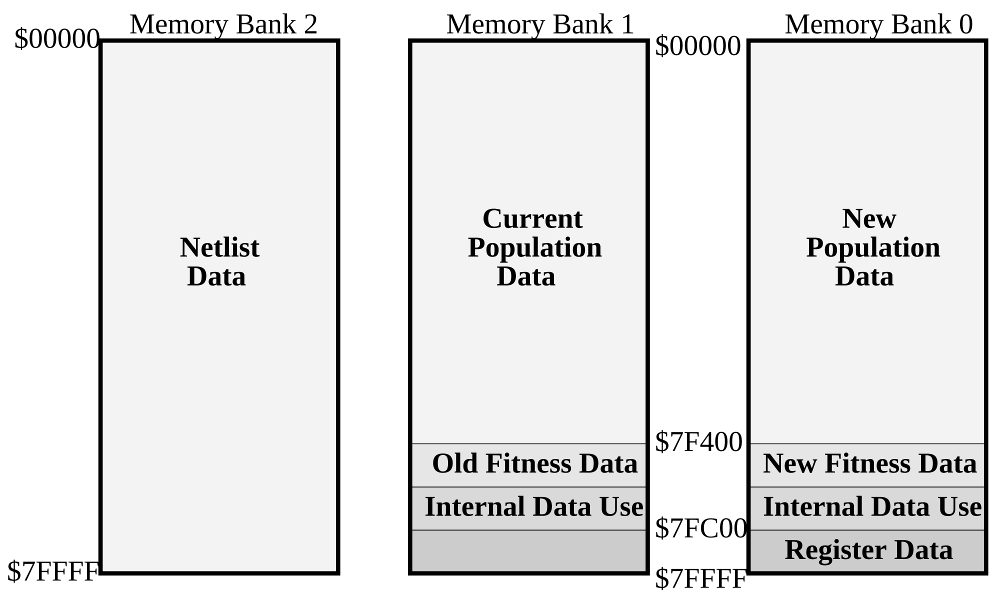


Figure 3.7: Genetic Algorithm Memory Map

1. *Population Data* : The population data holds the chromosome data of each individual in the population. This information is stored at the beginning of the memory block. Section 3.1.2 describes how the individual population is stored in memory. In implementing the design onto the RC1000 board, the population was limited to a size of 1024. This gives the maximum allowable

size of the population data to be 522,232 words (16,711,424 bits) allowing for larger benchmarks.

2. *Fitness Data* : The fitness data consist of 1024 words and are located at memory location 521,216 on both the new and old memory blocks. This holds the corresponding fitness values of each individual within the population.

3. *Internal Data Use* : Since the mating process is pipelined, there is a section of memory that is used solely for internal purposes. These data are stored immediately following the fitness data and have a maximum size of 1024 words.

4. *Register Data* : In order to transfer the register information to the GA algorithm, the register values are passed in through the memory. This information is located at the end of the memory bank 0, at memory location 523264.

A summary memory usage and starting locations can be found in Table 3.3

3.2 Create-Population-Module (CP-M)

The "Create-Population-Module" is developed to generate the initial random population for the Genetic Algorithm. To accomplish this task efficiently, the procedure is broken down into three pipelined components, as shown in Figure 3.8. The first submodule (Init Population Submodule) is responsible for the random generation of the initial population within the Genetic Algorithm process. The second component is used to repair individuals within the population that are infeasible as explained in section 2.4.1. This module randomly selects points in the chromosome

Memory	Addressable Memory Size	Description
Netlist Memory	$2^{NetWidth} \times NetNum$	The Netlist memory stores the binary chromosome information about each net
New Population Memory	$2^{NetWidth} \times PopSize$	This section of memory stores the binary information about the newly generated population
New Fitness Memory	$PopSize \times 2$	This memory holds the new fitness values of each individual in the new Population Memory
Internal Data Storage	$PopSize$	This data is used internally to determine who the children's parents are from the previous population
Old Population Memory	$2^{NetWidth} \times PopSize$	This section of memory stores the binary information about the current population
Old Fitness Memory	$PopSize$	This memory holds the fitness values of each individual in the Old Population Memory
General Purpose Memory	8	This memory holds the values that are to be loaded in the registers upon starting the program

Table 3.3: Memory Usage

and moves the cells from one partition to the other. The third module calculates the fitness values for the newly generated population. This process loops through each net in the Netlist to determine the number of nets that are uncut.

3.2.1 Init-Population-Submodule (IP-SM)

The "Init-Population-Submodule" is one of the main components of population initialization process. In this submodule, feasible/infeasible chromosomes are generated using the Random Number Generator (RNG)[Pres92] which generates a

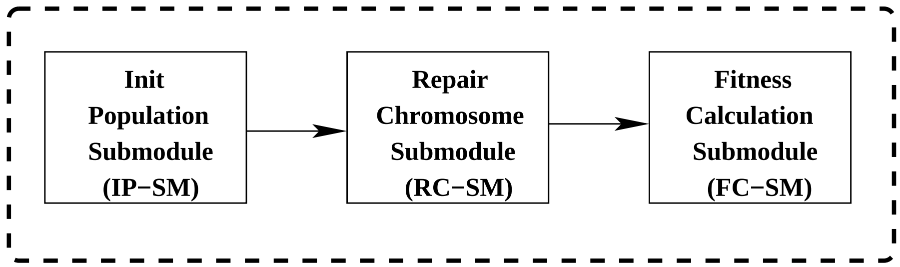


Figure 3.8: Create-Population-Module (CP-M)

sequence of random bits or genes. This process is repeated until each chromosome in the population can represent a point in the solution space.

Signal Organization

Figure 3.9 describes the signal interface between the IP-SM with other submodules within the system. A description of the signals can be found in Appendix A.1. Once a chromosome is generated and stored into memory, the Repair Channel Information is used to send the location in memory of the current chromosome to the repair function for checking its feasibility. A high on the *RepairStop* informs the Repair Chromosome Submodule that the entire population has been created and ends its process.

Functionality of IP-SM

The task of the IP-SM is to generate the initial pattern of a chromosome. The chromosome is stored in memory consisting of n data blocks of size *DataSize*, where n is $\frac{Cells}{DataSize}+1$. Once the system initiates the creation of the population, by driving the *PopInitEnb* signal high, the process shown in Figure 3.10 is initated.

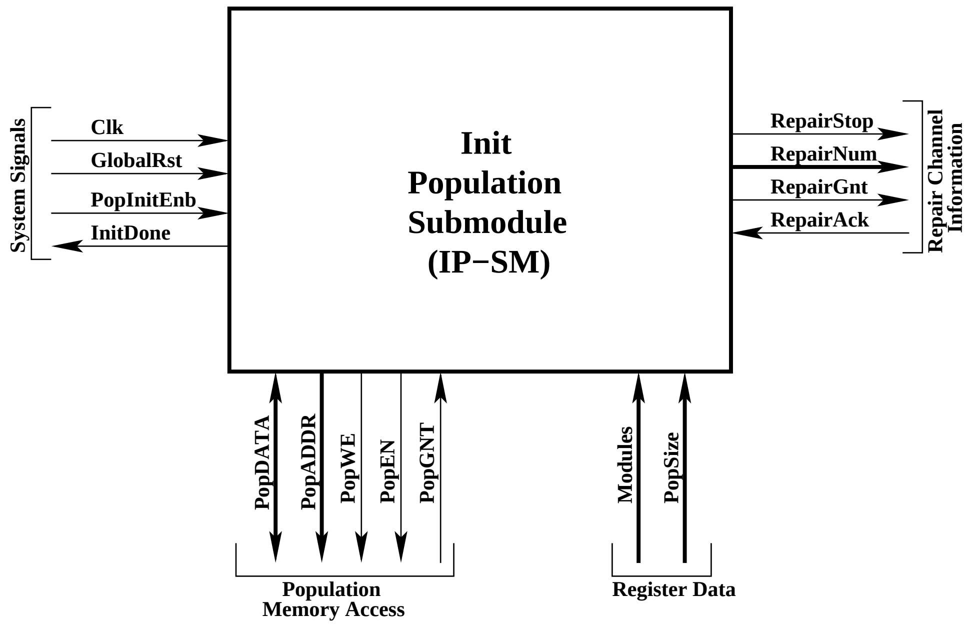


Figure 3.9: Init-Population-Submodule Signal Diagram

The process begins by generating a bit mask to mask out the upper unwanted bits on the Most Significant Byte (MSB) of the chromosome. The number of required bits of the last byte of data are calculated to be the remainder of $\frac{Cells}{DataSize}$. Once the mask is created, the submodule begins creating the population. This is achieved by generating random bit sequences using a RNG. These sequences represent the initial makeup of each chromosome within the population. This process is repeated for each data byte of the chromosome. The MSB of data consists of the remainder of the chromosome cells. In generating this data, the mask is applied to this random set of bits to set the upper $(DataSize - Remainder)$ bits to zero. Figure 3.11 illustrates a simple circuit on how chromosomes are created. Once a chromosome is completely generated, the required information is passed to the Repair Chromosome Submodule to check feasibility. After the information is passed

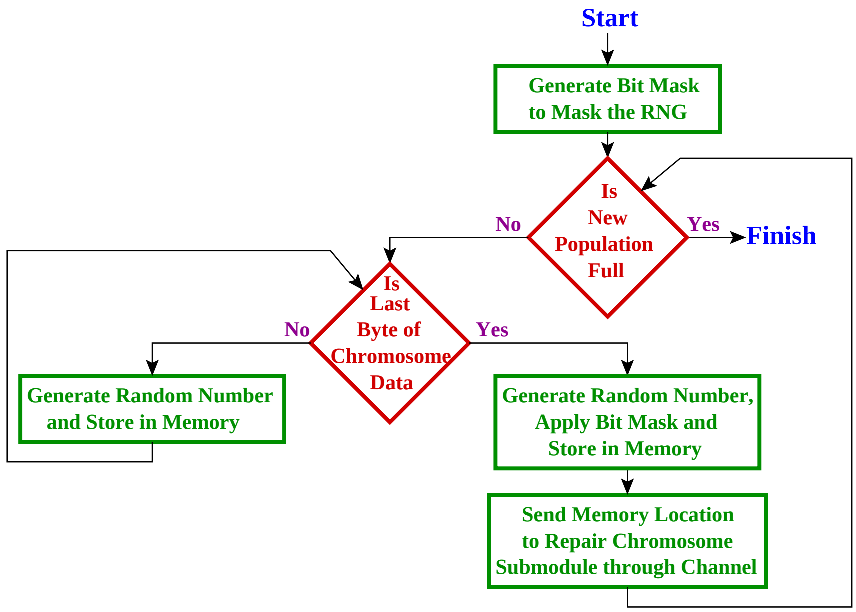


Figure 3.10: Init-Population-Submodule Block Diagram

on the channels, another chromosome is created. Upon completion of all chromosomes in the population, the system informs the Repair Chromosome Submodule that it has completed its task and places a high value on the *InitDone*. An example of the stored chromosome of size $Cells = 28$ is shown in Figure 3.12

3.2.2 Repair-Chromosome-Submodule (RC-SM)

The "Repair-Chromosome-Submodule" is used to modify infeasible chromosome generated by the IP-SM or Cross-Parent-Submodule (CP-SM). The objective is to create feasible solutions that meet the circuit partitioning balancing criteria. In fixing the chromosomes, random cells are selected within the chromosome and are moved from the partition with more number of cells to the partition with the fewer number of cells. This is repeated until the difference between the number of cells

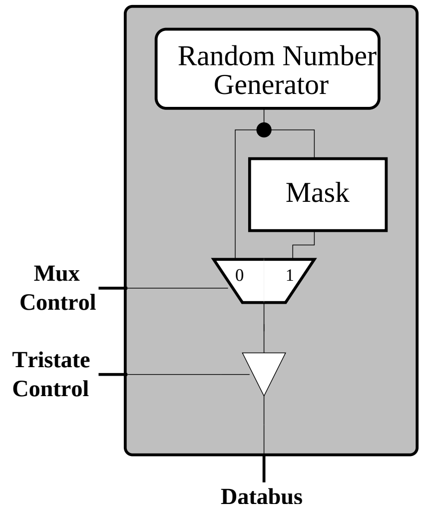


Figure 3.11: Internal design to Init-Population-Submodule

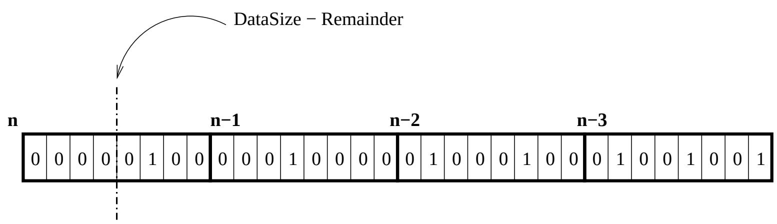


Figure 3.12: Stored chromosome data

within the partitions meets the balancing criteria.

Signal Organization

Figure 3.13 describes the signal interface between the RC-SM with other submodules within the system. A description of the signals can be found in Appendix A.2. The Repair Channel receives information from the IP-SM on the designated chromosome to be repaired. Once repaired, the same information is passed to the "Fitness-Calculation-Submodule" (FC-SM) through the Fitness Channel. If the

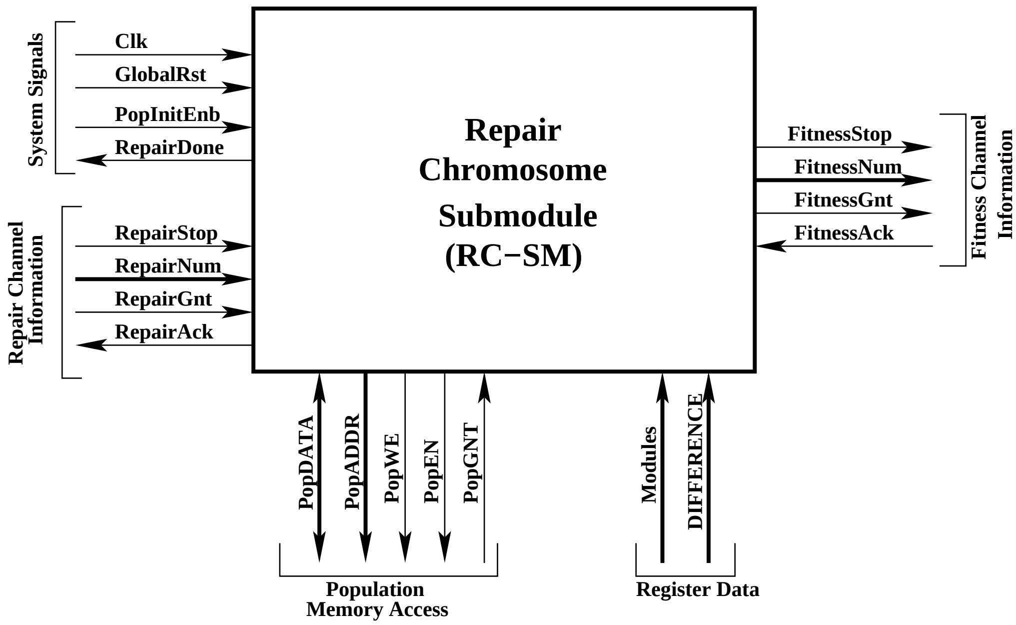


Figure 3.13: Repair-Chromosome-Submodule Signal Diagram

system receives a high signal on the *RepairStop* then all chromosomes within the population have been created and repaired. A high signal is then placed on the *FitnessStop* and the repair process halts.

Functionality of RC-SM

The task of the RCS is to determine the feasibility of a chromosome and repairing it. Once the system initiates the RC-SM, by driving the *PopInitEnb* signal high, the system enters an idle state until information is passed from the IP-SM. Once information regarding a chromosome is received the process in Figure 3.14 is initiated.

The initial task of the submodule is to determine the feasibility of a chromosome. This task is done by reading each byte of the chromosome and counting the number

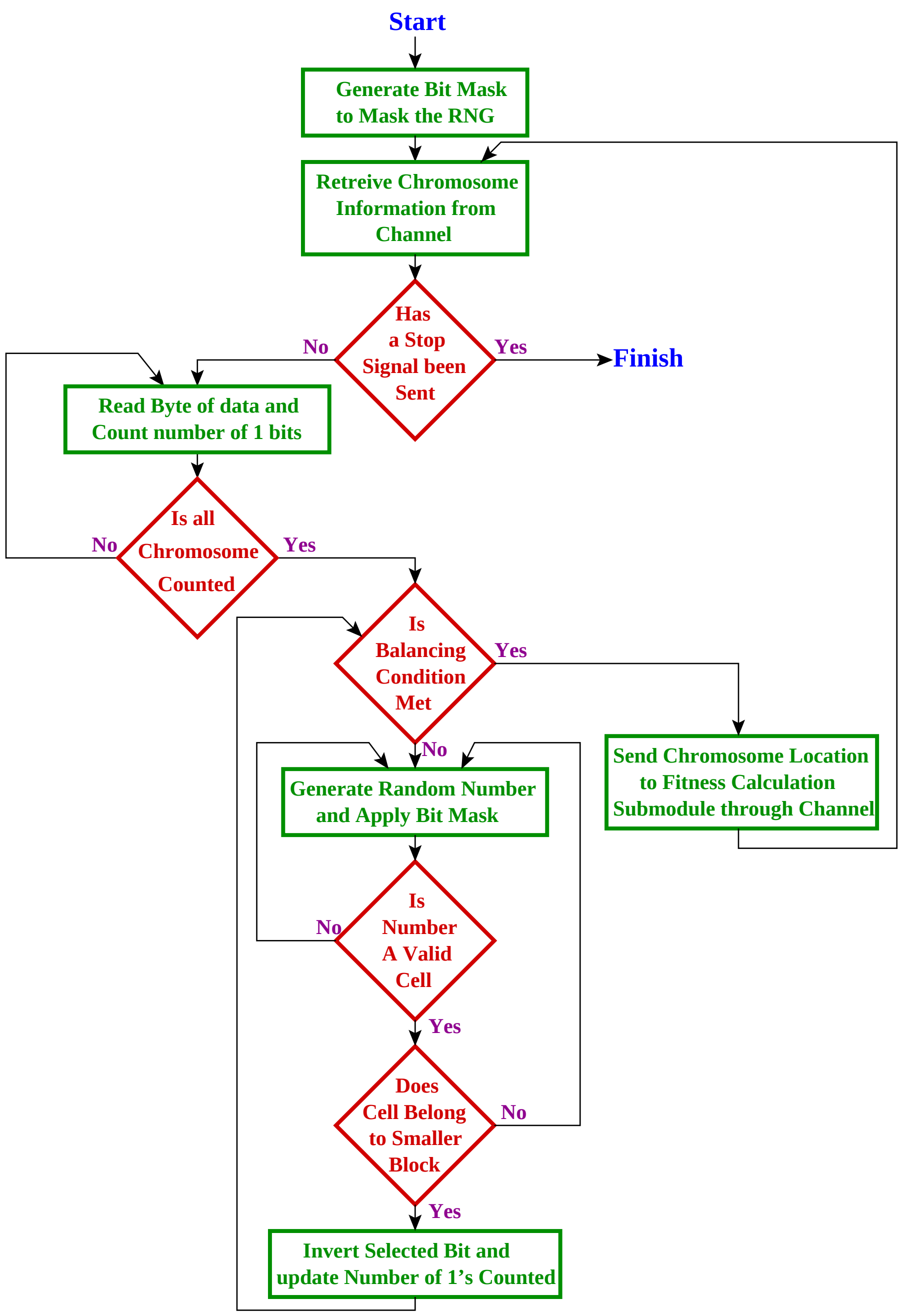


Figure 3.14: Repair-Chromosome-Submodule Block Diagram

of 1's that appear. In determining if the balancing criteria is met, both of the following equations must be satisfied.

$$BALANCE \geq Cells - 2 \times (Number\ of\ 1's)$$

and

$$BALANCE \geq 2 \times (Number\ of\ 1's) - Cells$$

If both equations are satisfied, the chromosome is considered feasible and the Repair process is complete; otherwise the chromosome is infeasible and must be repaired.

In repairing infeasible solutions, a masked random number is used to select random cells to move from one partition to the other. The masked random number generator generates numbers between 0 to $2^b \times DataSize$, where 2^b is the minimum number of bytes required to represent a chromosome. Although it is possible to generate invalid numbers (numbers larger than the number of cells within a chromosomes) the probability of selecting a feasible cell is $0.5 + \frac{1}{2^b \times DataSize}$. If the number generated is outside the boundaries of the chromosome data, then a new number is repeatedly generated until a valid number is selected. This repair process is illustrated in Figure 3.15

Once a viable cell is selected, it is moved from its current partition to the partition with the lower number of cells by inverting its value in the chromosome data. The register containing the number of ones is updated and the feasibility of the new solution is determined. This process is repeated until a feasible chromosome is generated that meets balancing criteria.

Once the chromosome is repaired, the system informs the FC-SM that the chromosome is feasible and waits for another chromosome to repair. If a stop signal

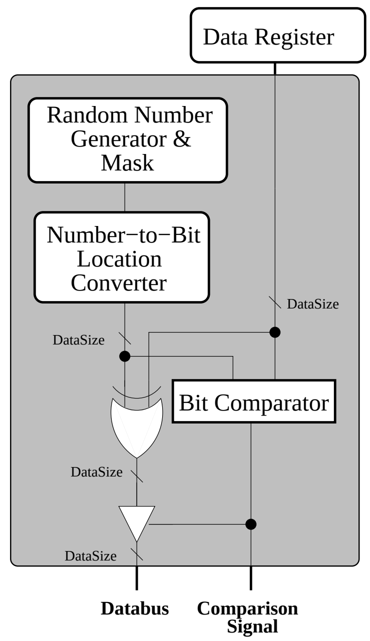


Figure 3.15: RC-SM Internal Repair Logic

occurs on the Repair Channel then a stop signal is sent to the Fitness Calculation Submodule and a high value is placed on the *RepairDone*.

3.2.3 Fitness-Calculation-Submodule (FC-SM)

The "Fitness-Calculation-Submodule" is used by both the Create-Population-Module (CP-M) and the Population-Reproduction-Module (PR-M) to calculate the fitness value of a given chromosome. In this process, each net within the Netlist is compared to the chromosome to determine the number of nets that are completely contained within a partition (uncut). This value is assigned to the chromosome to represent its fitness in comparison to other chromosomes.

Signal Organization

Figure 3.16 describes the interface between the FC-SM and other submodules within the system. A detailed description of the signals can be found in Appendix A.3. Although similar in functionality, there is one slight difference between the

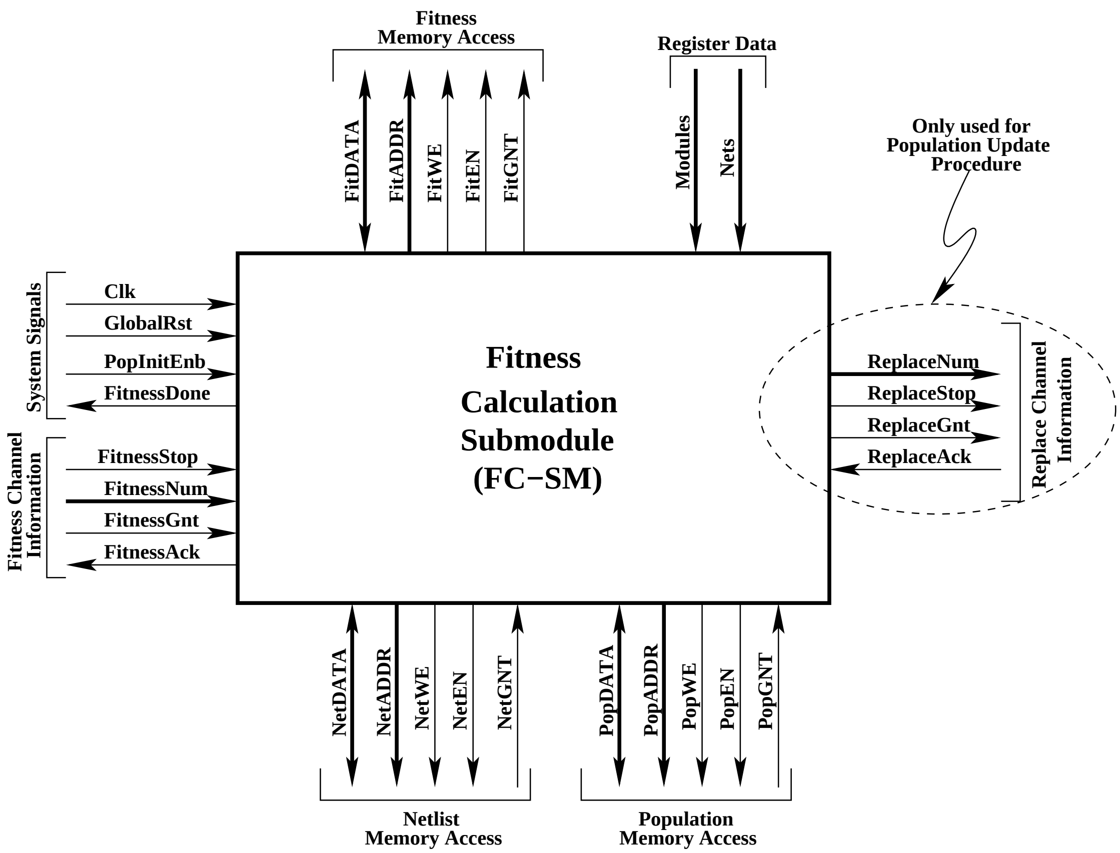


Figure 3.16: Fitness-Calculation-Submodule Signal Diagram

FC-SM used in the Initial-Population-Module and that used in the Population-Reproduction-Module (PR-M). When the submodule is used in the PR-M there is a channel communication with the Replace-Population-Submodule (RP-SM), described later on in this chapter. Once the fitness value has been calculated, the FC-

SM passes this chromosome to the Replace-Parent-Submodule (RP-SM) informing it of the newly created children. The Fitness Channel has the same functionality for both modules, receiving new members to be processed

Functionality of FC-SM

After the system initiates the fitness calculation process, by driving the *PopInitEnb* signal high, it remains in an idle state waiting for information to be passed through the fitness channel (ie. identify a member for fitness calculation). The process in Figure 3.17 is activated by receiving information of the designated individual.

The process begins by retrieving a byte of data from both the chromosome and the current net being tested. These two pieces of data are then compared to determine if the net is cut, cells connected to the net lie in both partitions, or otherwise uncut. This is accomplished using the circuitry presented in Figure 3.18, where each byte of the chromosome is compared with the corresponding byte of the net being tested. When the complete chromosome is compared and a high exists on the output of the Fitness Compare Logic, this indicates that the net is uncut. An incrementing counter is used to accumulate the number of uncut nets that exist. This process is repeated until all nets within the Netlist have been compared to the current chromosome and the number of uncut nets has been determined. Figure 3.19 gives an overview of the fitness calculation process.

Following the fitness calculation, the system returns to an idle state waiting for the next chromosome to process. If a high signal is passed on the *FitnessStop* then the submodule ends its processing and places a high signal on the *FitnessDone* to inform the system that it has completed its task.

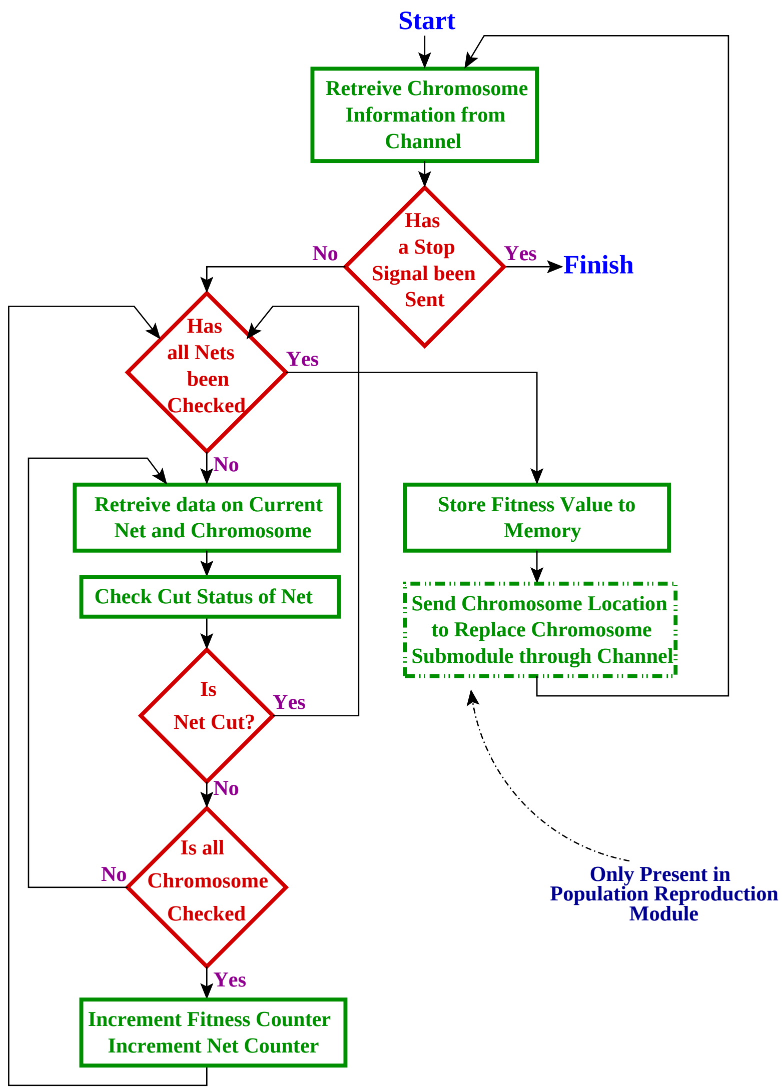


Figure 3.17: Fitness Calculation Submodule Block Diagram

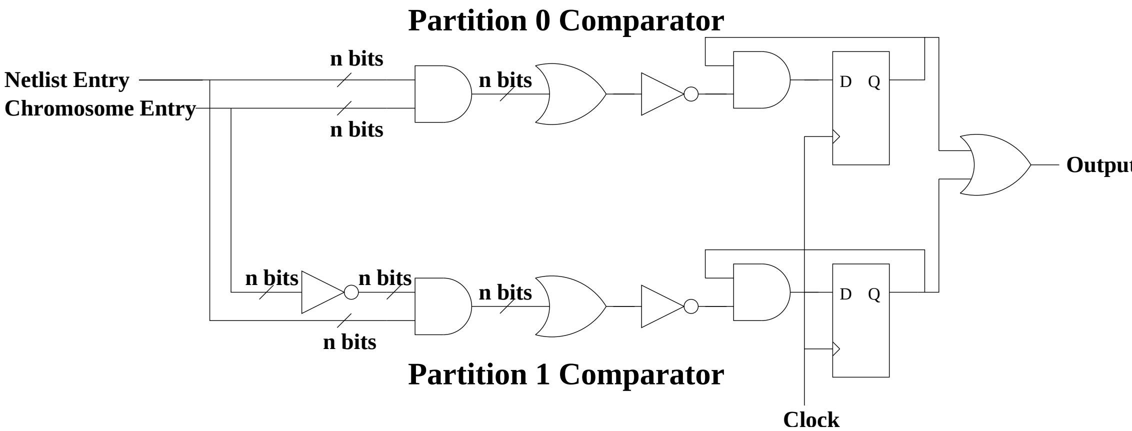


Figure 3.18: Fitness Compare Logic [Sitk95]

3.3 Population-Reproduction-Module (PR-M)

Once the initial population is created, the evolutionary mating process begins. Population mating is the procedure of creating new offspring chromosomes through a random combination of the parents' genes and mutation. The aim is to create a new and better fit population than the previous population. In the current design, this process is broken down into seven pipelined components, as shown in Figure 3.20. A detailed explanation of each component's tasks was introduced in section 2.4.3. To create a completely new population, the pipeline process must be executed $\frac{Population\ Size}{2}$ times, since each set of parents selected creates two offspring in the new population.

3.3.1 Select-Parent-Submodule (SP-SM)

The "Select-Parent-Submodule" performs the initial task of the PR-M. This submodule is used to select two fit individuals from the current population (ie. par-

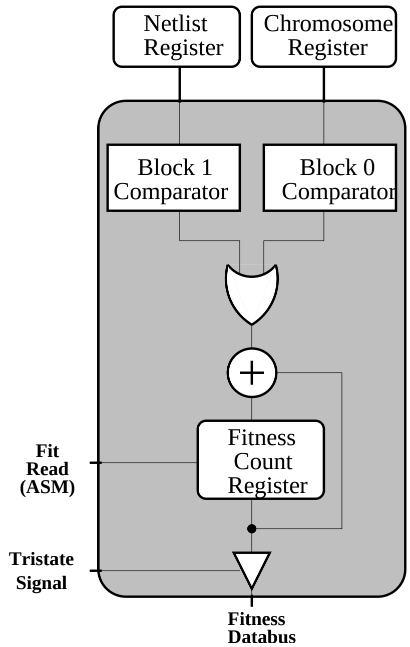


Figure 3.19: Internal Fitness Layout

ents)to create new offspring. The task is performed by tournament selection, described in section 2.4.3. After selecting the two parents from the current population, the selection routine determines, with a given probability, if these two individuals should mate to produce offspring or survive unaltered into the new population. The probability of successful mating of the two individuals is $\frac{Crossover\ Rate}{65535}$, where $Crossover\ Rate$ is a user defined variable. Following the selection of parents, the addresses of the two individuals are sent to the Cross-Parents-Submodule (CP-SM) to create offsprings. Otherwise, the addresses of the individuals are sent to the Copy-Parents-Submodule (CP-SM) where they are copied directly to the new population.

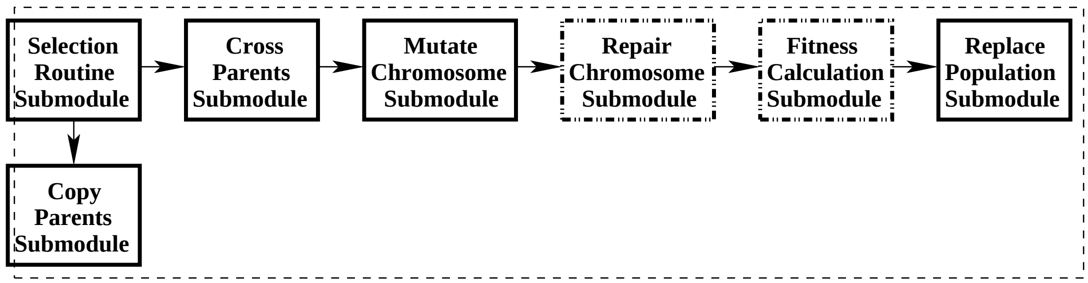


Figure 3.20: Population Reproduction Module (PR-M)

Signal Organization

Figure 3.21 describes the signal interface between the SP-SM with other submodules within the system. A detailed description of the signals can be found in Appendix A.5. The submodule has communication channels with CP-SM and the Copy-Parents-Submodule (CoP-SM). Both channels are used to send selected parents by the process as well as the location in the new population they should be stored. The *CopyStop* and *CrossStop* signals are used to inform the submodules that the new population is complete.

Functionality of SP-SM

The task of SP-SM is to select two mating parents from the current population. Once the system initiates the selection procedure, by driving the *PopRepoEnb* signal high, the process shown in Figure 3.22 is initiated. The process begins by initially selecting one individual from the population as a potential parent in the mating process. This is accomplished using the RNG and the mask modules. Since the population must be of a size equal to a power of 2 (2^x), the mask is simply calculated as $Mask = PopSize - 1$. Figure 3.23 demonstrates an example of how individuals

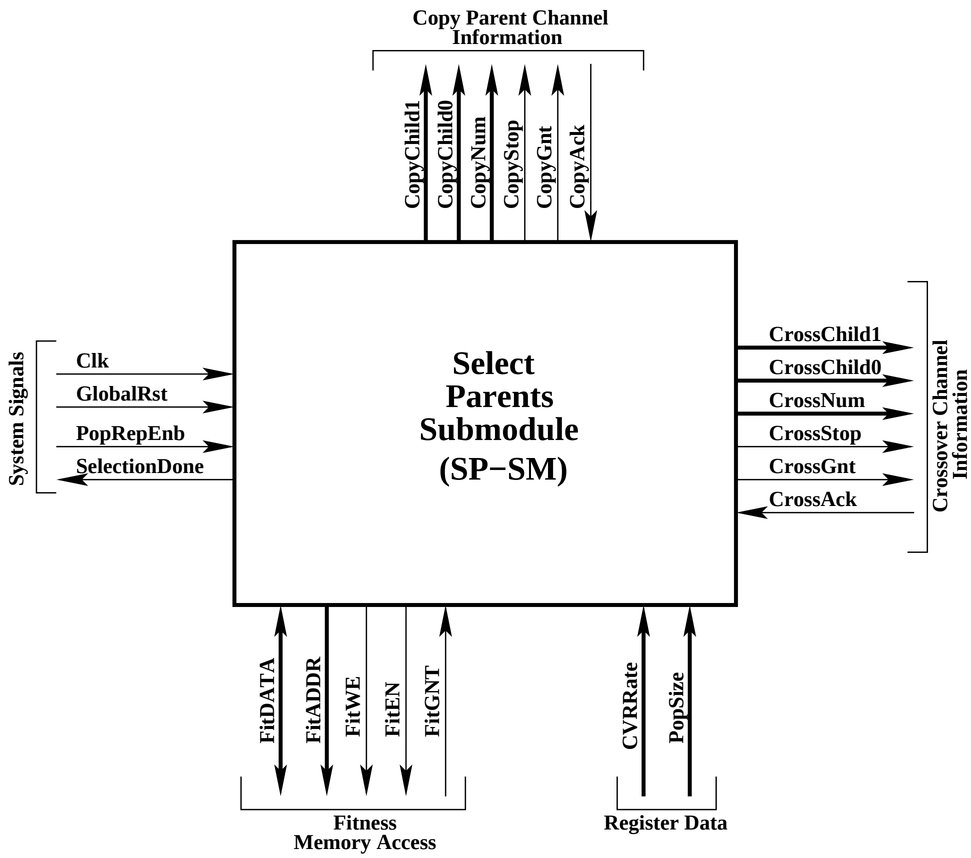


Figure 3.21: Select-Parent-Submodule Signal Diagram

are selected when the *PopSize* is 32 (2^6).

A second unique individual is selected from the population[1]. Figure 3.24 demonstrates the process of selecting unique individuals from the population. A competition takes place between the two individuals to determine which one is better fit to reproduce. The chromosome with the highest fitness value becomes the mating parent.

[1]Unfortunately these two selection processes must occur in a sequential manner to maintain the RNG Seed

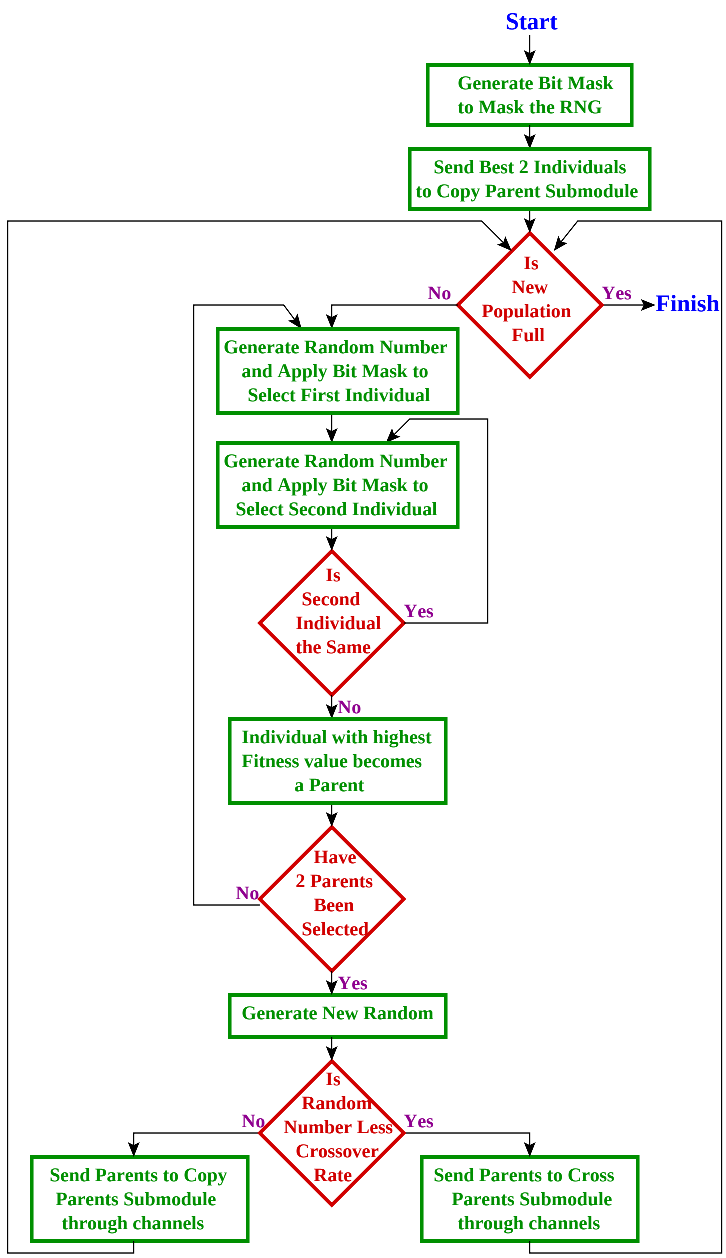


Figure 3.22: Select Parents Submodule Block Diagram

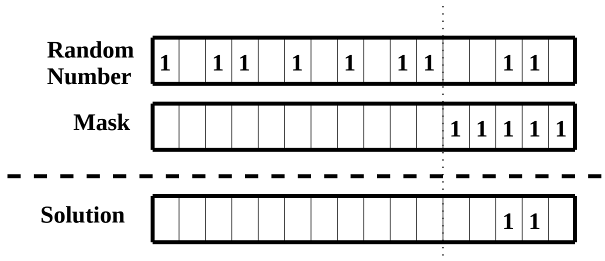


Figure 3.23: Masking Random Number

This process is again performed to select the second parent. The two parents selected will eventually generate new offspring if a random number generated is less than the crossover rate determined by the user. Otherwise an unmodified copy of the parents is passed to the next generation.

This process is repeated until enough offspring have been generated to fill the new population, $\frac{PopSize}{2}$ times. Following the generation of the new population, stop signals are sent on the two communication channels. A high signal is placed on the *SelectionDone* to inform the system that the Select Parent Submodule completed its task.

3.3.2 Cross-Parent-Submodule (CP-SM)

The "Cross-Parent-Submodule" performs the mating task of the two parents to create the initial gene sequence for the offsprings. This is done by selecting random genes from each parent to form a new gene sequence for each offspring. In determining which genes are selected the submodule uses the Uniform Crossover technique, described in section 2.4.3. Once the gene sequences for new offspring have been generated they are passed to the Mutate-Chromosome-Submodule (MC-SM) for

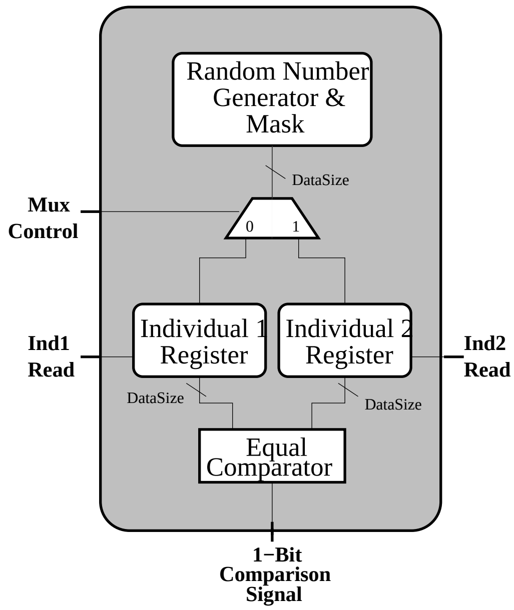


Figure 3.24: Selection of Unique Individuals

mutation.

Signal Organization

Figure 3.25 describes the signal interface between the CP-SM with other submodules within the system. A description of the signals can be found in Appendix A.6. The Crossover Channel receives information from the Select Parent Submodule as to who the parents are and where in the new population memory the offspring are to be stored. When the crossover process is complete the memory locations of the offspring are individually sent to the MC-SM for mutation. If the system receives a high from the *CrossStop* then all chromosomes for the given generation have been created. A high is then placed on the *MutationStop* and the crossover process halts.

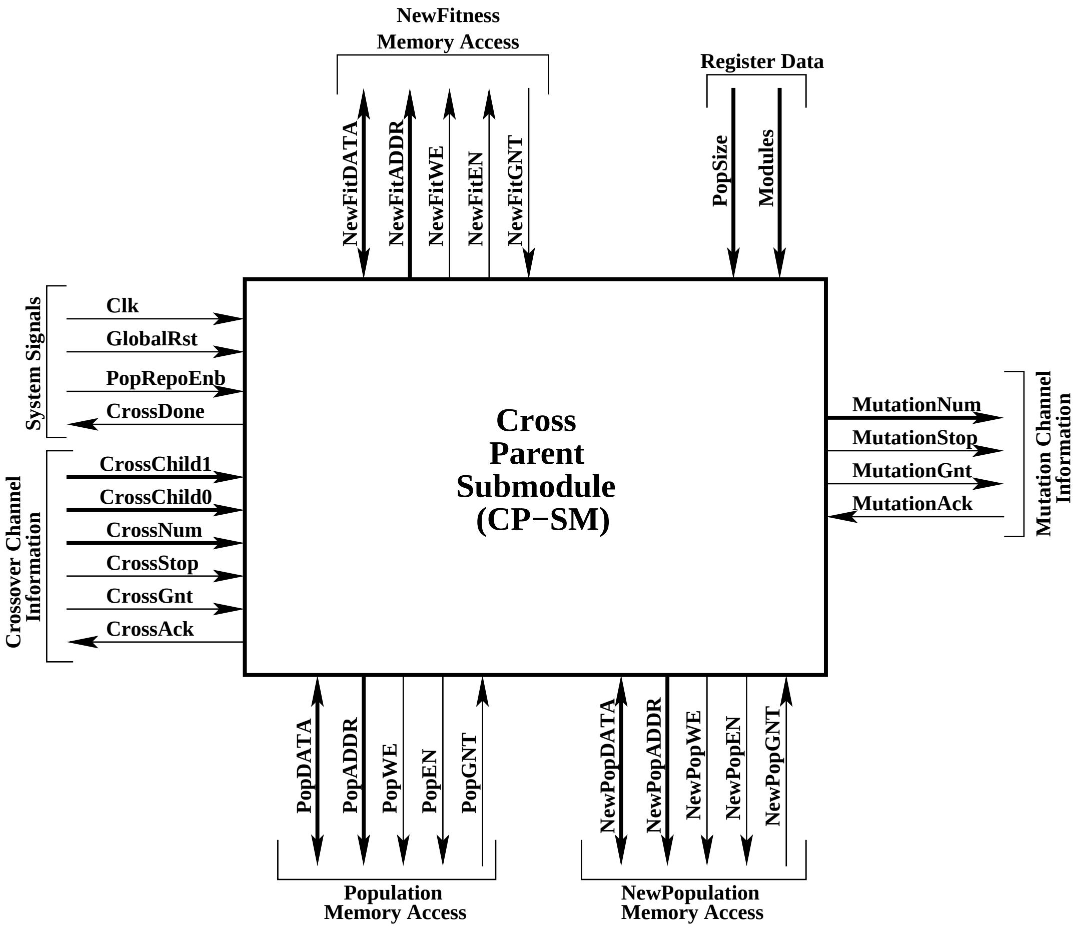


Figure 3.25: Crossover-Process-Submodule Signal Diagram

Functionality of CP-SM

The task of the CP-SM is to mate the two parents to generate two new offspring. Once the system initiates the crossover procedure, by driving the *PopRepoEnb* high, the process shown in Figure 3.26 is initiated.

The process begins by remaining in an idle state until information is received from the Select-Parent-Submodule. After it is determined which parents are to mate, a random combination of gene sequences from each of the parents is stored

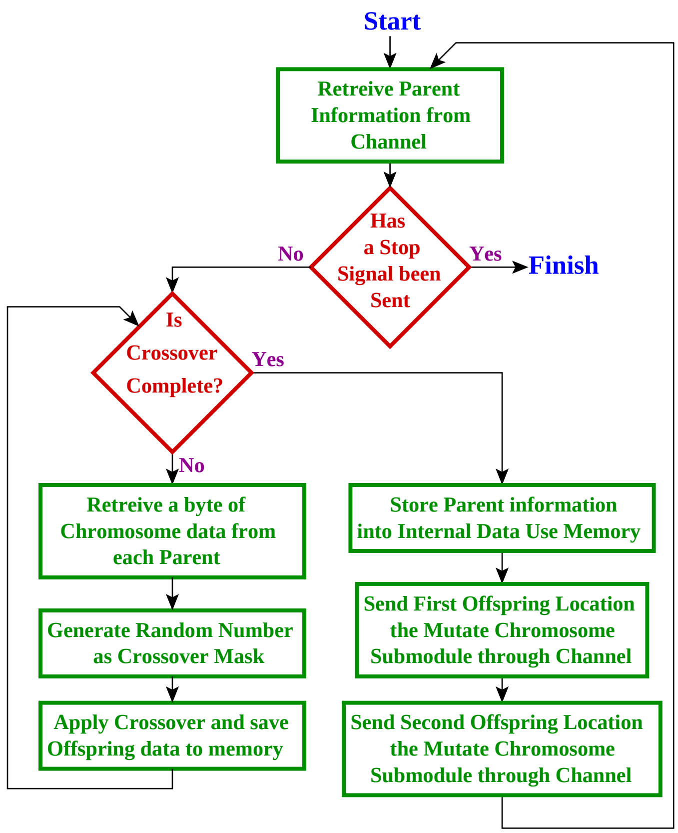


Figure 3.26: Cross-Parent-Submodule Signal Diagram

in the offspring memory location, following the uniform crossover technique. This process is done by obtaining a random number to mask which genes come from each of the parents. Figure 3.27 shows the logical design of the Uniform crossover function. This process is repeated until new offspring have been created and stored in the new population. The location of the offspring in the new population memory is passed to the MC-SM one at a time and the location of the parents in the current population is then stored in *Internal Data Use* memory. This is so that it is possible to determine who the offspring's parents are at a later date.

Once the information has been passed on the channels, the system returns to an

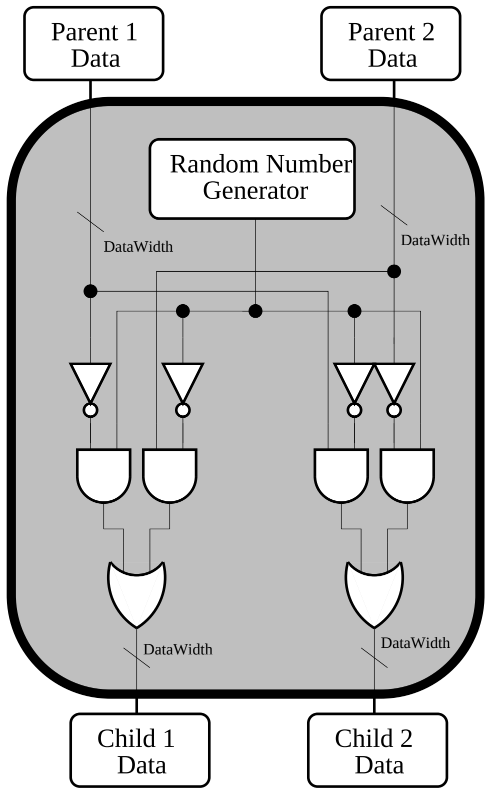


Figure 3.27: Crossover Combinational Logic

idle state waiting for the next parents to process. If a high signal is passed on the *CrossStop* then the submodule ends its processing and places a high signal on the *MutationStop* to inform the MC-SM that the new population has been created. A high is then placed on the *CrossDone* to inform the system that it has completed its task.

3.3.3 Mutate-Chromosome-Submodule (MC-SM)

The "Mutate-Chromosome-Submodule" performs an evolutionary mutation on the offspring and causes slight changes to the genes within the chromosomes. In the mutation process, each gene of the chromosome has a given probability, $\frac{Mutation\ Rate}{65535}$, of mutating. If a cell within the chromosome is to be mutated its value is inverted causing the given cell to be moved into the opposite partition. Upon completing the mutation process, the location of the current chromosome in memory is passed to the RC-SM.

Signal Organization

Figure 3.28 describes the signal interface between the MC-SM with other submodules within the system. A description of the signals can be found in Appendix A.8.

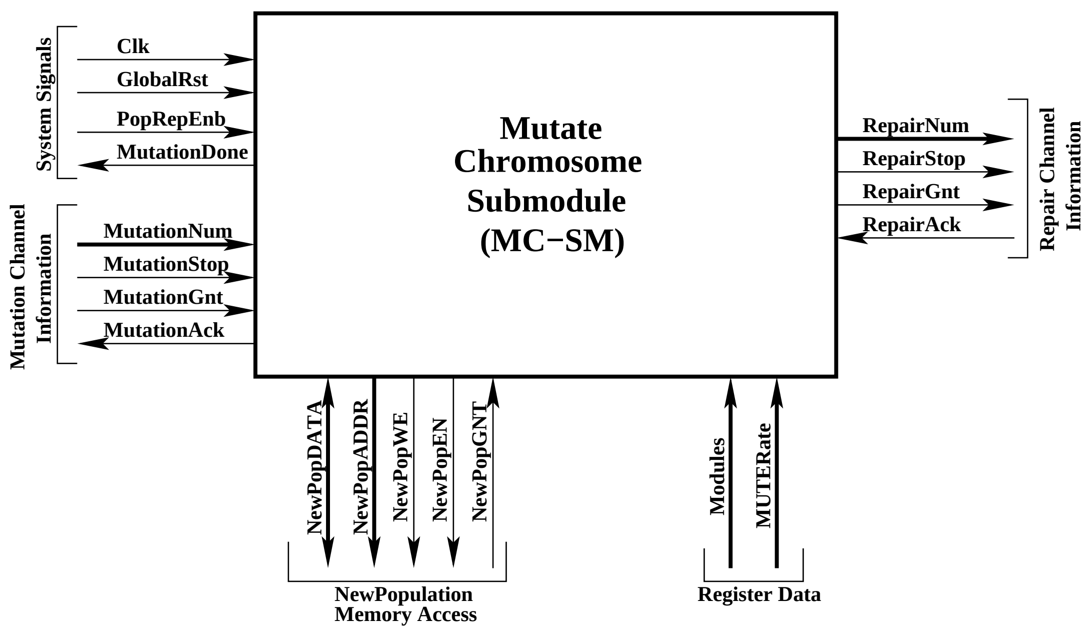


Figure 3.28: Mutation-Chromosome-Submodule Signal Diagram

The submodule has one incoming and one outgoing channel communication. The Mutation Channel is used to receive information about which of the chromosomes within the new population needs to be mutated. This channel will also inform the system as to when the new population is finished by sending a high signal on the *MutationStop*. The Repair Channel is used to send the same information to the RCS once the mutation process is finished.

Functionality of MC-SM

The task of the MC-SM is to perform a mutation on newly generated offsprings. Once the system initiates the mutation procedure, by driving the *PopRepoEnb* signal high, the process shown in Figure 3.29 is initiated.

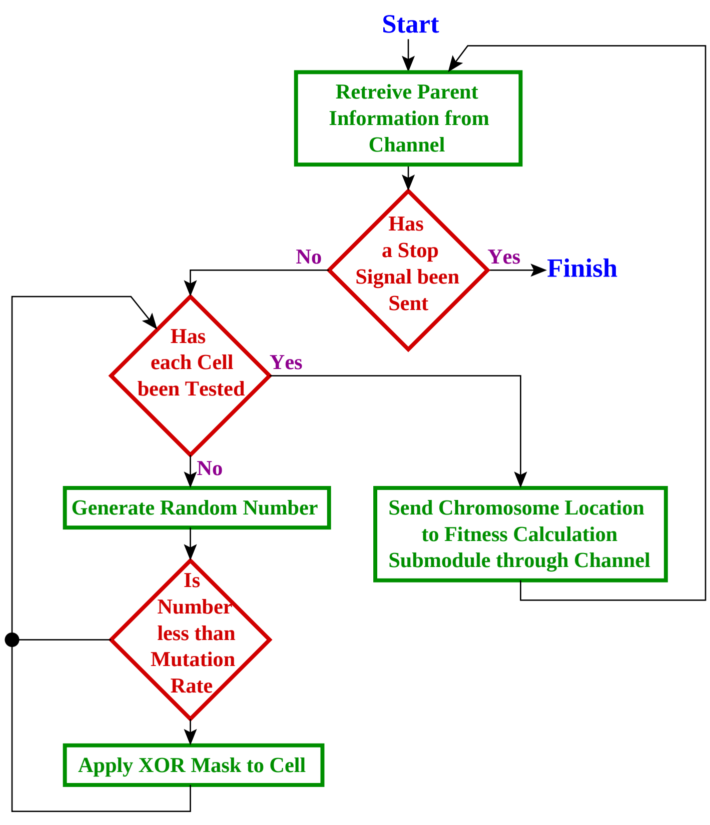


Figure 3.29: Mutate-Chromosome-Submodule Block Diagram

The process begins by remaining in an idle state until information is received on which chromosome in the new population is to be mutated. Once this information is received, the mutation process can begin. The mutation process generates a random number for each cell in the chromosome. If the number is smaller than the value in the *MUTERate* register, defined by the user, then mutation of this cell will occur.

The gene or cell is mutated by inverting the current value of the bit and storing it back into memory. The mutation is accomplished by applying a XOR with a mask, as shown in Figure 3.30.

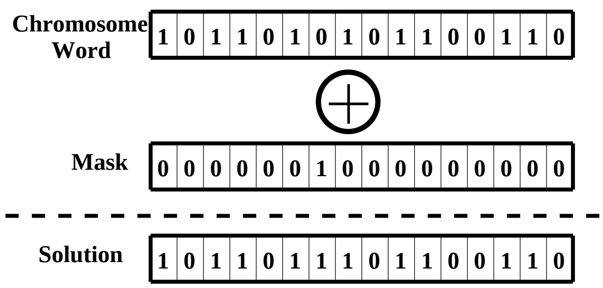


Figure 3.30: Bit Mutation

When the mutation process is complete and each cell has been checked, the number of the chromosome in the new population is passed to the repair function through the Repair Channel.

Once the information has been passed on the channels, the system returns to an idle state waiting for the next chromosome to process. If a high signal is passed on the *MutationStop* then the submodule ends its processing and places a high signal on the *RepairStop* to inform the RCS that the new population has been created. A high is then placed on the *MutationDone* to inform the system that the submodule has completed its task.

3.3.4 Replace-Population-Submodule (RP-SM)

The "Replace-Population-Submodule" performs the task of selecting which of the parents and children should be placed into the new population. There are many different techniques used to generate a new population, as described in section 2.4.3. In selecting a replacement technique the limitations placed on the system must be considered. In generating the new population, the old population cannot be modified until the entire new population is generated, so that the original information is not modified while the pipeline is in use. Due to this limitation, the replacement routine selected is the 'Best Child and Parent' technique, replacing the memory of the least fit child chromosome with the chromosome data of the best fit parent.

Signal Organization

Figure 3.31 describes the signal interface between the RP-SM with other submodules within the system. A description of the signals can be found in appendix A.9. The submodule has one incoming channel communication, Replace Channel, to receive information about which of the chromosomes within the new population have just been generated. This channel will also inform the system as to when the new population is finished by sending a high signal on the *ReplaceStop*.

Functionality of RP-SM

The task of the RP-SM is to select which of the parents and children should be placed into the new population. Once the system initiates the replacement proce-

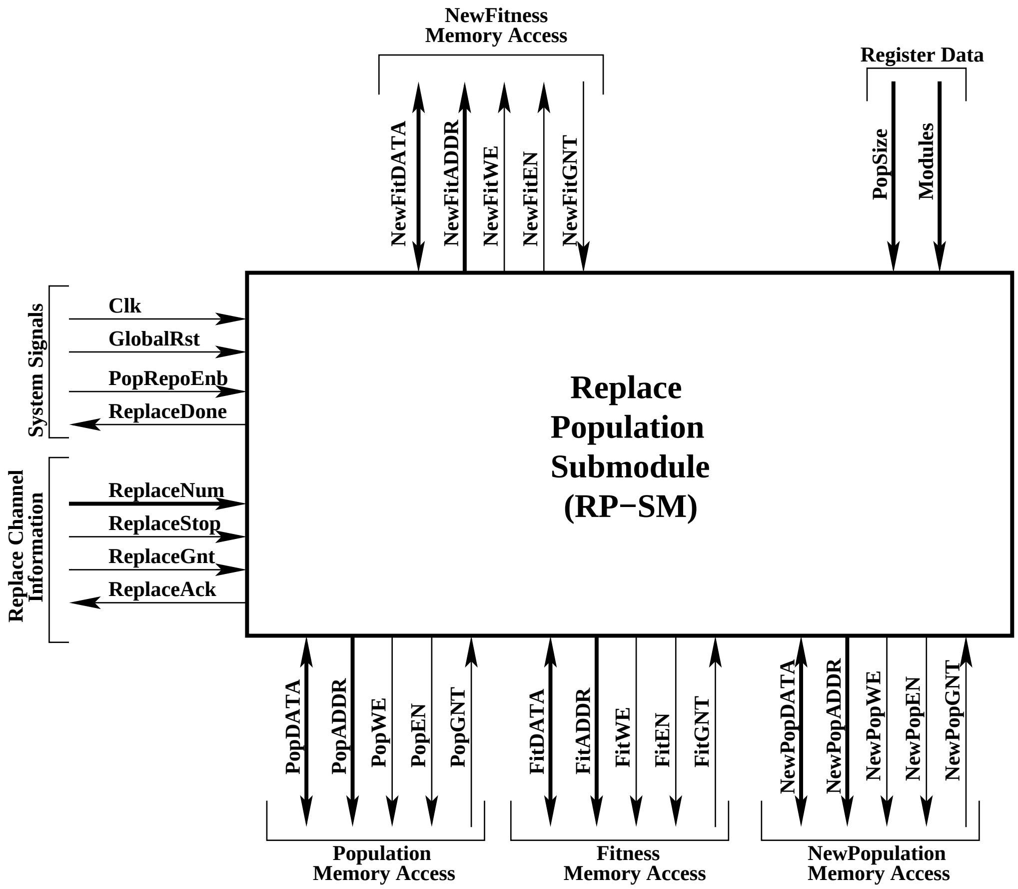


Figure 3.31: Replace-Population-Submodule Signal Diagram

dure, by driving the *PopRepoEnb* signal high, the process shown in Figure 3.32 is initiated.

The process begins by remaining in an idle state until information is received on the location of the new chromosome by the RP-SM. When this information is received, the system retrieves the identity of the parents of the children. This is so that the parents can compete with the children to determine who should survive in the new population. This is done by comparing the fitness values of the parents and the offspring to determine which is the fittest parent and which is the weakest child.

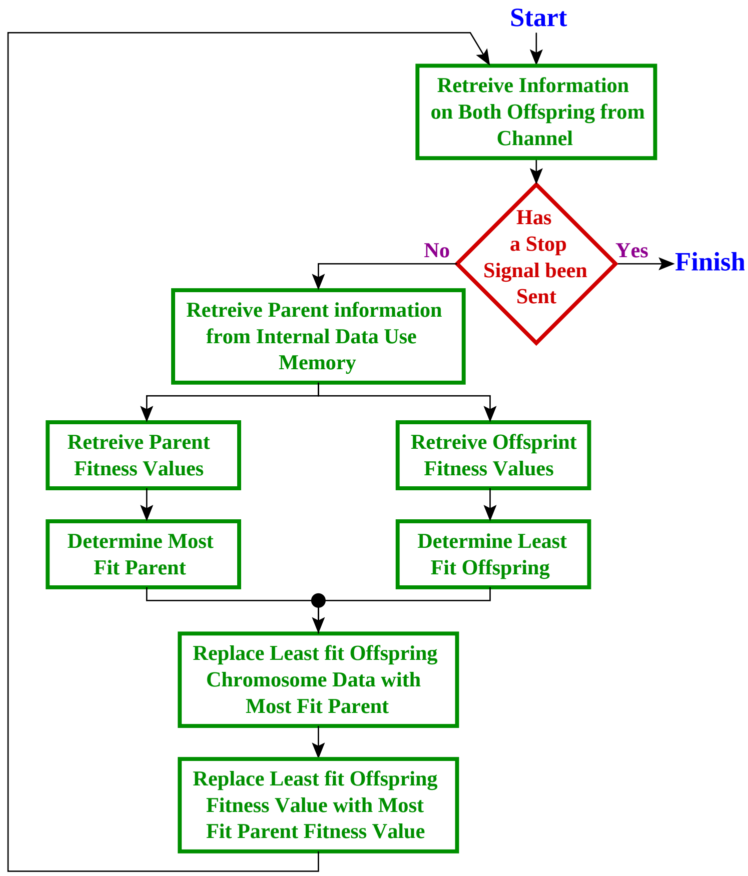


Figure 3.32: Replace Population Submodule Block Diagram

The weakest offspring chromosome is replaced with strongest parent's information, leaving the strongest of the parent and of the offspring to survive as part of the new population.

Once the parent has been stored into memory, the system returns to an idle state waiting for the next chromosome data to be passed through the channels. If a high signal is passed on the *ReplaceStop* then the submodule ends its processing and places a high signal on the *ReplaceDone* to inform the system that it has completed its task.

3.3.5 Copy-Parents-Submodule (CoP-SM)

The "Copy-Parents-Submodule" allows selected individuals from the old population to survive into the new population. This procedure is called by the Select Parent Submodule for two instances:

1. To copy the two fittest chromosome from the original population into the new population. This follows the concept of élitism which ensures the survival of the best chromosome into the new population[Reev02].

2. If no mating process is to take place. This occurs when the Select Parents Submodule determines, with a given probability, that the two selected parents should survive unchanged into the new population.

In these two cases, the CoP-SM will copy the chromosome and fitness data of the selected individuals from the current population into the new population.

Signal Organization

Figure 3.33 describes the signal interface between the CoP-SM with other submodules within the system. A description of the signals can be found in Appendix A.10. The submodule has one incoming channel communication, Copy Parent Channel, to receive information about which of the chromosomes within the old population are to be moved into the new population. This channel will also inform the system to when the new population is finished by sending a high signal on the *CopyStop*.

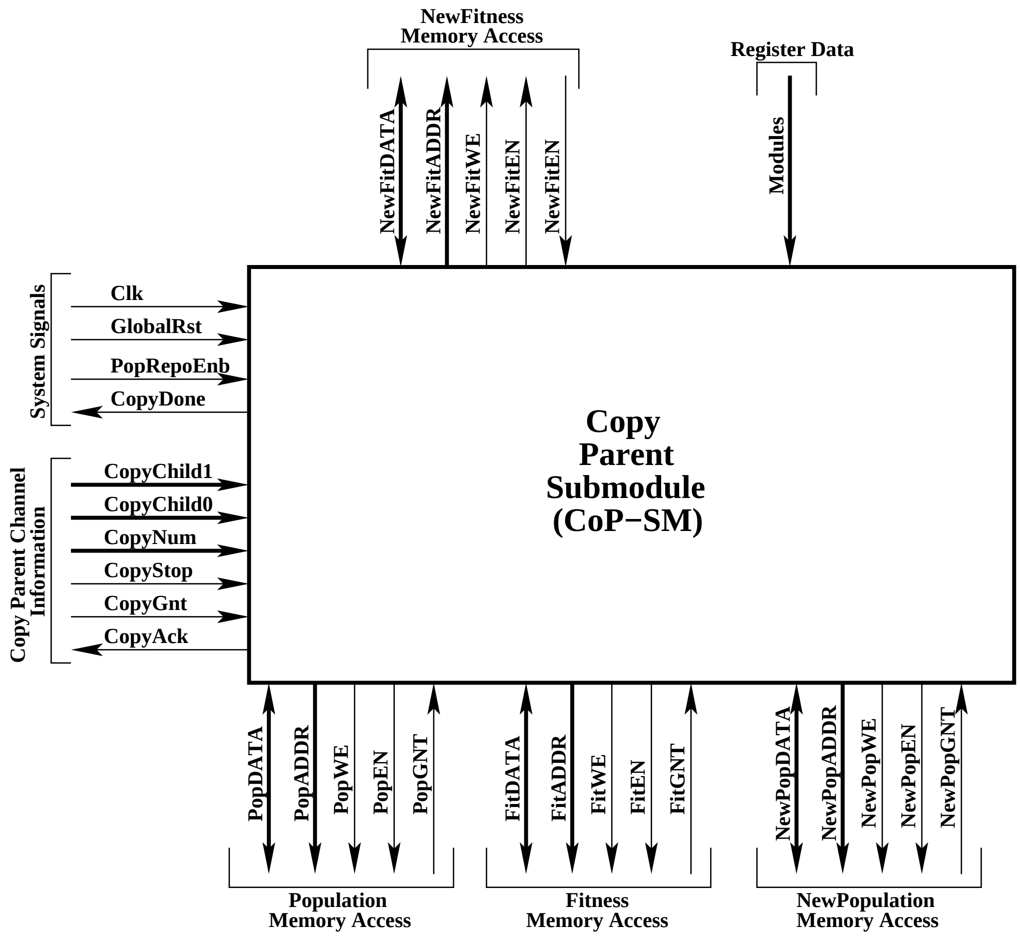


Figure 3.33: Copy-Parents-Submodule Signal Diagram

Functionality of CoP-SM

The task of the CoP-SM is to generate a copy of the chromosome and fitness data from the parents and store them into the new population. Once the system initiates the copying procedure, by driving the *PopRepoEnb* signal high, the process shown in Figure 3.34 is initiated.

The process begins by remaining in an idle state until information on the location of the two chromosomes is received. As information arrives, the process

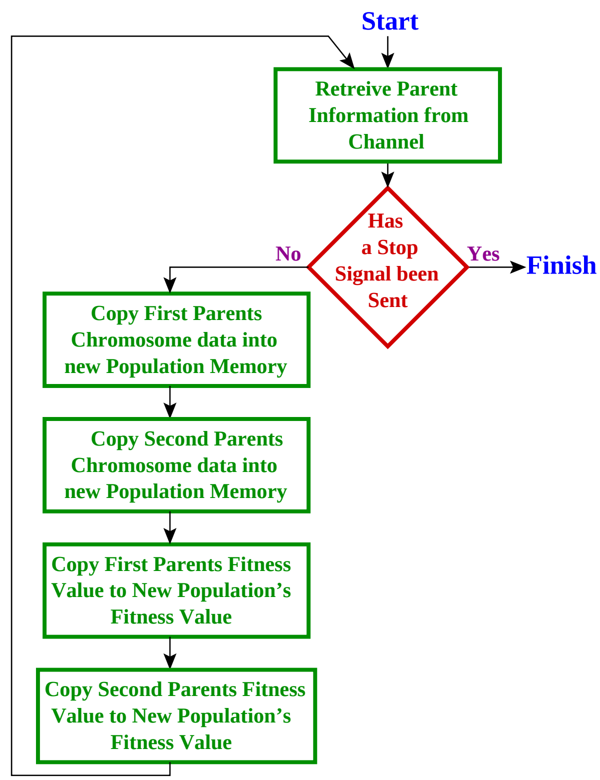


Figure 3.34: CoP-SM Block Diagram

retrieves the chromosome data and fitness values from the current population and stores them in the new population. When the data is copied the system returns to an idle state waiting for the next channel information to be passed. If a high signal is placed on the *CopyStop* then the submodule ends its processing and places a high signal on *CopyDone* to inform the system that it has completed task.

3.4 Simulation and Verification

The initial goal of the Genetic Algorithm design was to implement a complex pipelined architecture into a FPGA to achieve better performance than software implementations. Simulation and verification of functionality was achieved through the internal Handel-C simulator. In order to compare the solution and performance of the design, two different software implementations were used. The first implementation was designed to use the same methodology as the Handel-C design, with each bit within the unsigned integer representing a cell attached to a net. The second implementation was developed by [Arei01]. This algorithm is used for comparing solution qualities and performance issues of the design. In examining the execution time of the hardware implementation vs the software implementations, it was found that both software algorithms produced much faster results than the hardware design, as shown in Table 3.4 and Figure 3.35 respectively. The following

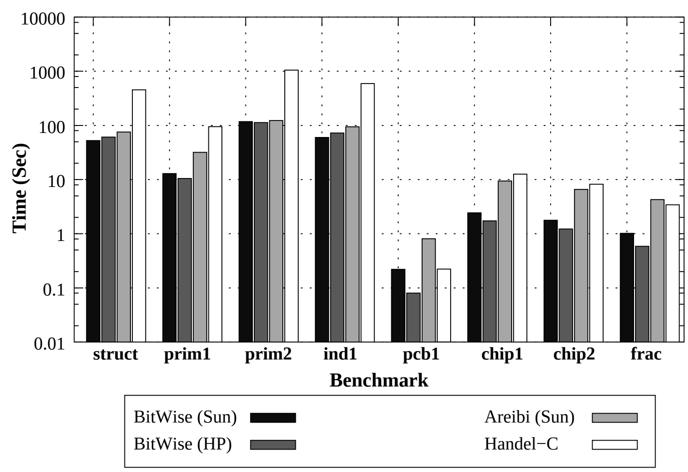


Figure 3.35: Software vs Hardware comparison graph

sections will discuss some of the potential problems found with the initial architecture resulting in lack of performance and modifications attempted to rectify these issues.

Benchmark	Bitwise Software Genetic Algorithm		Areibi[Arei01] Software Genetic Algorithm	Handel-C Hardware Genetic Algorithm
	Sun Workstation	HP Workstation	Sun Workstation	63 MHz
struct.dat	52.563 s	60.713 s	75.570 s	454.487 s
prim1.dat	12.903 s	10.367 s	31.670 s	94.791 s
prim2.dat	117.287 s	112.880 s	123.043 s	1048.710 s
ind1.dat	60.063 s	72.470 s	94.326 s	591.587 s
pcb1.dat	0.22 s	0.080 s	0.810 s	0.222 s
chip1.dat	2.43 s	1.733 s	9.420 s	12.553 s
chip4.dat	1.773 s	1.223 s	6.573 s	8.206 s
frac.dat	1.017 s	0.587 s	4.270 s	3.322 s

Crossover=99%, Mutation=0.35%, Population Size=128, Generations=200

Sun Blade 2000 : 900 MHz UltraSparc III Cu, 1024 MB Ram, Solaris 9

HP Workstation 2100: Intel P4 2.4 GHz, 1 GB Ram, Redhat Linux 9

Table 3.4: Software/Hardware timing

3.4.1 Performance Analysis and Tuning

As previously discussed, the main problem with the initial Genetic Algorithm implementation was a lack of execution speed. In analyzing the architecture, numerous bottlenecks were identified.

1. In profiling the software algorithm (see Table 3.5) the CalculateFitness function required the majority of the execution time. Although these results were based on a software implementation, it is expected that the Fitness Calculation submodule in hardware will also produce the greatest bottleneck of the system.

Name	Hardware Equivalent in Handel-C	% Execution Time			
		struct	prim2	prim1	chip1
CalculateFitness	Perform Fitness task	93.29	94.08	86.43	76.85
Random	Random Number Generator	3.96	3.73	7.72	14.81
Count	Count modules in Blk1	1.55	1.19	3.13	3.70
Mutation	Perform Mutation task	0.93	0.78	1.88	3.70
Replacement	Perform Replace task	0.06	0.00	0.00	0.00
Repair	Perform Repair task	0.03	0.00	0.21	0.93
Crossover	Perform Crossover task	0.03	0.00	0.42	0.00
Selection	Perform Selection task	0.00	0.00	0.00	0.00
Overhead		0.15	0.22	0.21	0.01

Table 3.5: Genetic Algorithm Software Profile

2. In simulating the system, it was found that memory access also contributed to the timing problem. One of the main objectives in developing the current Genetic Algorithm implementation was not to constrain the system to internal Block Rams. Consequently, all memory storage was implemented externally allowing for larger benchmarks to be solved. However, in using

external memory, only one submodule of the design may access each memory bank during a clock cycle. This causes the majority of submodules to become idle, waiting to gain memory access and resulting in less throughput. One possible solution to this problem is to use dual-port memory allowing two pieces of memory to be accessed at a single time. The memory accessing will still cause a bottleneck but will allow the FC-SM to be split into two pipeline stages causing the processing speed to double. However, the RC1000 development board does not support off-chip dual port Ram.

3. When comparing the timing of simulation and the actual results, we discovered that memory accessed by Handel-C plays a role in the timing of the design. In order to protect access to external memory, semaphores are used to protect each memory read and write cycle. Using semaphores has the consequence of increasing memory access time by one extra clock cycle. Therefore, two clock cycles are required to perform each operation on memory. Since Genetic Algorithms are extremely memory intensive and all current memory is stored off-chip, This contributes to further delay in the system.

4. A lack of parallelism is also found in the pipeline stage of the architecture. Within the Crossover Submodule two offspring are generated while only one can be passed on through the pipeline. This causes the crossover and replacement to suspend waiting to pass the second offspring through the pipeline. Consequently the throughput of the pipeline is limited which slows down the operation of the algorithm.

3.4.2 Design Enhancement

In an attempt to resolve the issues encountered with the original design and further enhance performance, the following improvements were implemented:

1. The Fitness Calculation submodule (as stated earlier) places the largest burden on the system. In analyzing this submodule, it was found that the majority of the execution occurred searching empty words of data within the Netlist. The current method of fitness calculation (developed by Stikoff et al.[Sitk95]) produces quick results for small benchmarks which have at least one cell for each word of data but is impractical as benchmarks increase in size. In Table 2.4 it is found that the majority of the nets are connected to fewer than 5 cells resulting in many data words of large benchmarks (ie. prim2) containing no useable data. To resolve this problem, a new method of storing the Netlist data was implemented. Unlike the previous method which uses a bit to represent each net, the new method stores the Netlist as integers representing each cell connected to a given net. An example of a sample Netlist can be found in Figure 3.36, where elements of data containing a '-1' value signals the end of the net entry. This form of storage allows the system to read only useful information about each net and no further time is spent searching empty data. For small benchmarks the new fitness method will most likely increase execution speed since more bytes of data are required to represent a single net. In examining the connectivity of the nets within the Netlist, see section 2.4.2, the majority of the nets are attached to five or fewer cells, meaning that as benchmarks become larger this new method

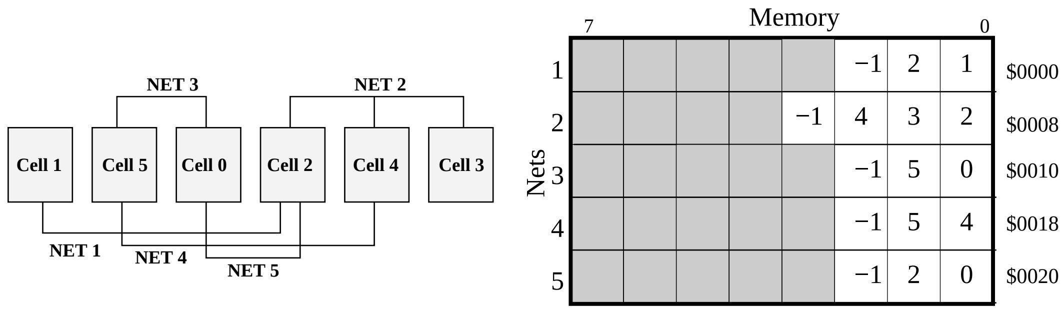


Figure 3.36: New Netlist Storage using Integer Values

should dramatically increase the execution speed compared to the original design.

In indexing an element in the solution data, the upper bits of the Netlist entry represent the byte containing the desired data and the lower bits represent the index of the bit within that byte. Figure 3.37 illustrates a simple Netlist entry using 8-bit byte of data. The lower 3 bits of the integer represent the bit location within the byte and the remaining upper bits determine the byte containing this cell.

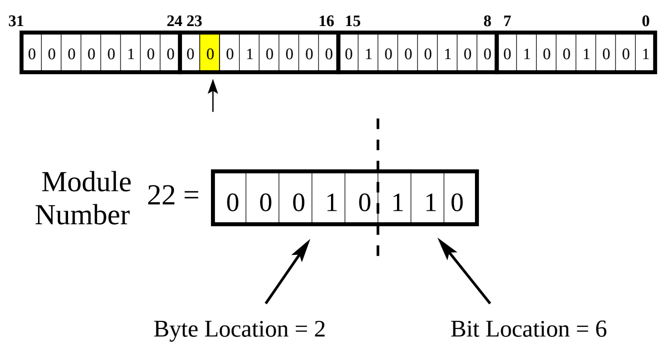


Figure 3.37: Bit Lookup using Integer Values

2. To resolve the issue of extensive memory accessing, the use of internal block

Rams were implemented into the design. These block Rams are used for internal storage of the offspring, allowing each submodule to have dedicated access. The block Rams are loaded with the initial offspring values created by the Crossover Parent submodule. This memory is then passed through the pipeline allowing each submodule to perform operations on the offspring. The offspring chromosome is then stored into off-chip memory by the Replace Routine submodule. The process is illustrated in Figure 3.38. Since the block Rams are dedicated to each submodule within the pipeline, memory conflicts are eliminated and the need of semaphores is reduced.

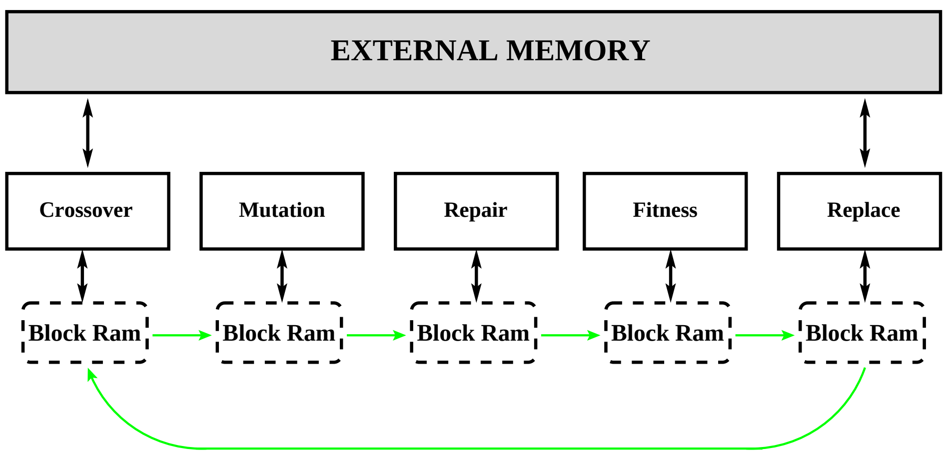


Figure 3.38: Population Reproduction with Block Rams

3. Examining the usage of semaphores within the system resulted in determining that reading from the Netlist memory is dedicated only to the Fitness Calculation submodule and does not require memory protection. Eliminating this semaphore from the algorithm tends to speedup the system, since the majority of the processing lies within this submodule.

4. In order to increase the throughput of the pipeline a second Mutation Process Submodule, Repair Chromosome Submodule and Fitness Calculation Submodule may be introduced into the system in parallel, as illustrated in Figure 3.39. This allows both offspring of the system to be processed simultaneously stalling the system.

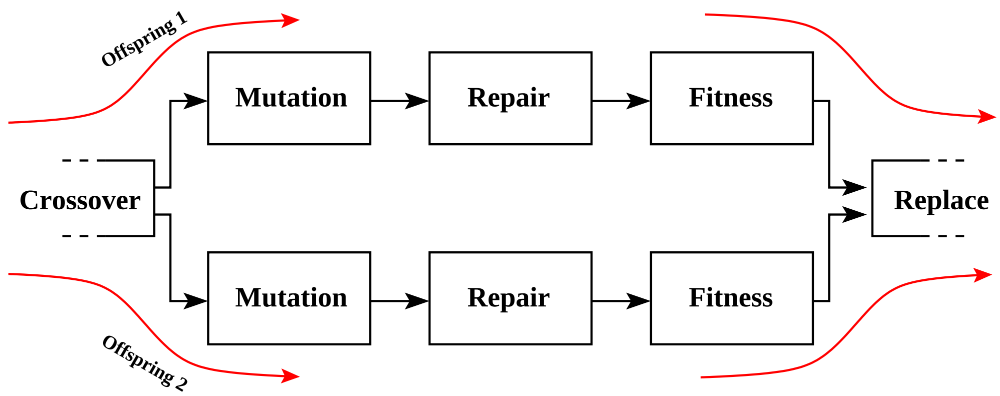


Figure 3.39: Parallel Pipeline Architecture

3.5 Computational Results

In designing the Genetic Algorithm, the aim was to optimize speed of the algorithm while producing good solution quality. As discussed in section 3.4.2 numerous designs were implemented to improve the execution speed of the algorithm. All these proposed designs were created using Celoxica DK Suite 2.0 and compiled using Xilinx ISE 6.1.03i. They were implemented on the Celoxica RC1000 development board using a Virtex E FPGA with 2 million gates. Results of these design performances can be found in Table 3.6.

In examining the data from different implementations it should be noted that

Benchmark	Original Design	New Fitness Function	Block Ram Memory	Pipeline (No Semaphores)	Parallel Pipeline
Maximum Clock	63 MHz	64 MHz	63 MHz	65 MHz	63 MHz
Equivalent Gates	61,731	61,731	363,915 (18 BlkRam)	363,725 (18 BlkRam)	510,954 (26 BlkRam)
struct.dat	454.487	38.616	24.006	13.156	22.059
prim1.dat	94.791	18.344	11.381	6.231	10.419
prim2.dat	1048.710	60.144	37.478	20.559	34.325
ind1.dat	591.587	42.756	26.731	14.687	24.466
pcb1.dat	0.222	0.631	0.397	0.218	0.359
chip1.dat	12.553	6.197	3.825	2.087	3.488
chip4.dat	8.206	4.384	2.734	1.506	2.519
frac.dat	3.322	3.141	1.938	1.059	1.797

Average time over 5 trials using base case parameters

Table 3.6: Genetic Algorithm Design Comparison

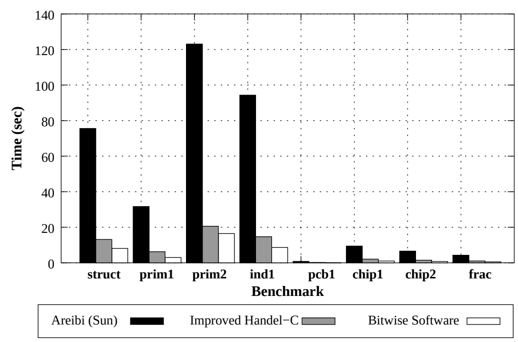
the single pipeline with no semaphores in the fitness calculation executed in nearly half the time needed by the parallel pipeline implementation. This is due to the fact that executing two Fitness Calculation submodules in parallel requires semaphores to protect the Netlist memory reads. As discussed earlier, semaphores add an extra clock cycle to the read process resulting in two clock cycles for a single read. Therefore, the remaining results of the thesis will be generated by a single pipeline method with no semaphores.

In order to analyze the solution qualities of the design properly it is necessary to examine the effect of various parameters. In testing these effects a base case of these parameters was used, shown in Figure 3.7.

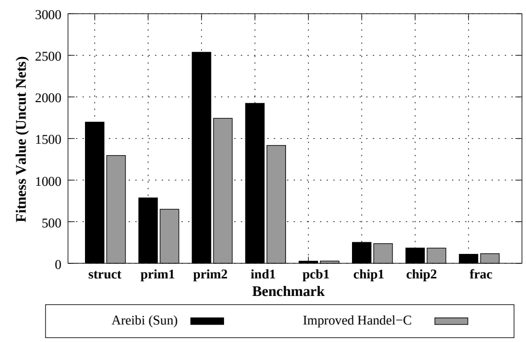
Parameter	Default Values
Population Size	128
Number of Generations	200
Crossover Rate	99%
Mutation Rate	0.36%
Balancing Difference	2

Table 3.7: Base Case parameters for Handel-C Genetic Algorithm

Results of the hardware and the software using the base case parameters can be found in Appendix C.1 and illustrated in Figure 3.40. From Figure 3.40(a) it can be noticed that the hardware architecture produces significant speedups over the software version developed by [Arei01] while still executing more slowly than the software which used the same bitwise representation. The lack of performance can be attributed to the fitness calculation function. The issue with the fitness function is that it operates in a sequential manner (ie. processing time increased as benchmark sizes increases). Therefore in order to achieve enough speedup to



(a) Software vs Hardware Time Comparison



(b) Software vs Hardware Fitness Comparison

Figure 3.40: Hardware vs Software Results

outperform the bitwise software more pipelining and parallelism within this function are required.

An improvement comparison of the hardware over Areibi's software can be found in Table 3.8. The bitwise software implementation has been excluded from this comparison since it generates the same fitness results as the hardware. From these results, it can be seen that the Areibi software produced significantly better fitness values (on average 13% better) than the hardware solutions. This improvement can be attributed to three factors:

1. The random number generator[Pres92] may have a different effect on the hardware architecture than the random number generator used within the software implementation.

2. The difference in the two algorithms' crossover technique may play a crucial role in the results. The software algorithm utilizes a 2-point crossover in-

Benchmark	Hardware performance improvement over Software implementation	Hardware solution quality improvement over Software implementation
struct.dat	574.4%	76.4%
prim1.dat	508.3%	82.7%
prim2.dat	598.5%	68.7%
ind1.dat	642.2%	73.6%
pcb1.dat	371.6%	106.4%
chip1.dat	451.4%	94.3%
chip4.dat	436.5%	99.6%
frac.dat	403.2%	107.0%
Average	498.3%	88.6%

Table 3.8: Hardware improvement over Software

stead of a uniform crossover implemented in hardware. The 2-point crossover method would better satisfy the schema theory[Reev02] attempting to maintain gene sequences within the chromosome. This may be the reason why the Standard Deviation is larger for the software implementation, keeping the population more diverse and better searching the solution space.

3. The software algorithm utilizes a more advanced method for repairing the chromosomes.

In tuning the design the base case was used while altering only one parameter to view its effect on the system. Sections 3.5.1 to 3.5.5 discuss the results of the tuning process. All numerical results can be found in Appendix C.

3.5.1 Effect of Generation Size on Solution Quality

In order to determine the role that the generation size plays on the solution quality, the base case parameters were used while modifying the generation size. Figure 3.41 shows the mean objective value generated at different generation sizes and demonstrates that the quality of the solution increases with larger generation size.

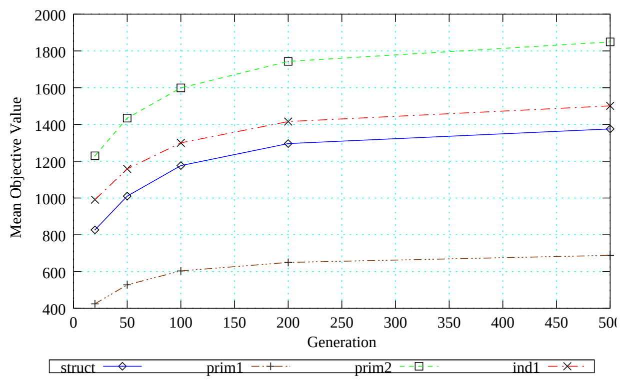


Figure 3.41: Effect of Number of Generations on Mean Objective Value

The greatest change in the solution quality occurs with values below 200. As generations move above 200, it would be expected that the population is converging onto a single solution. This is illustrated in Figure 3.42 which shows that 200 generations is the low point in the standard deviation curve meaning that the majority of the solutions are converging towards a single fitness value. The improvement in solution quality and the standard deviation as generations move above 200 are the result of random walking within the system caused by mutation.

In determining the effect that the generation size has on the execution time, it was found that increasing the value results in a linear increase in processing time, as shown in Figure 3.43. This linear increase is a result of the FCS which was

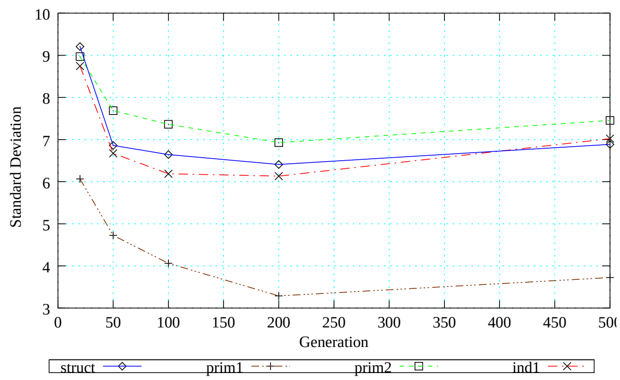


Figure 3.42: Effect of Number of Generations on Standard Deviation

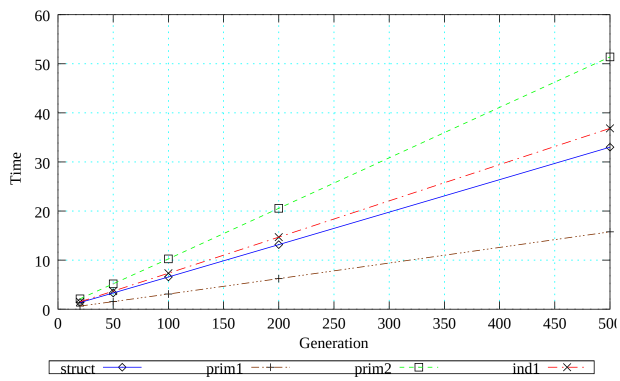


Figure 3.43: Effect of Number of Generations on Execution time

previously found to be the bottleneck of the pipeline and determined the execution time of the pipeline. Since the fitness calculation executes at a near constant rate for each fitness value produces a linear time increase would be expected.

3.5.2 Effect of Crossover Rate on Solution Quality

In examining the effects of the Crossover Rate on the solution quality, the base case parameters were used while modifying the crossover rate. Through Figure 3.44, it was found that increasing the crossover rate has a small effect on the solution quality. The cause of this small increase in solution quality is a result of more

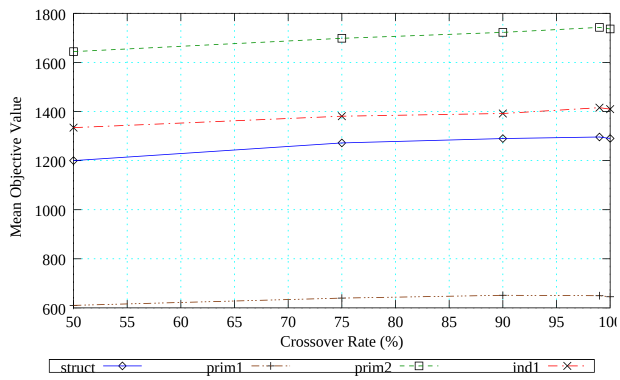


Figure 3.44: Effect of Crossover Rate on Mean Objective Value

newly generated chromosomes in the population which leads to a higher probability of producing fit individuals. A low crossover rate results in more chromosomes being copied from the current population into the new population resulting in fewer offspring generated.

In examining Figure 3.45 it is found that the crossover rate has a linear effect on the execution time of the system. As the crossover rate decreases, fewer offspring cause the system to spend less time executing the reproduction pipeline.

3.5.3 Effect of Mutation Rate on Solution Quality

In examining the effects of the mutation rate on the solution quality, the base case parameters were used while modifying the mutation rate. Through Figure 3.46 it is found that increasing the mutation rate has a negative effect on the solution quality. As the mutation rate increases each offspring undergoes more mutation

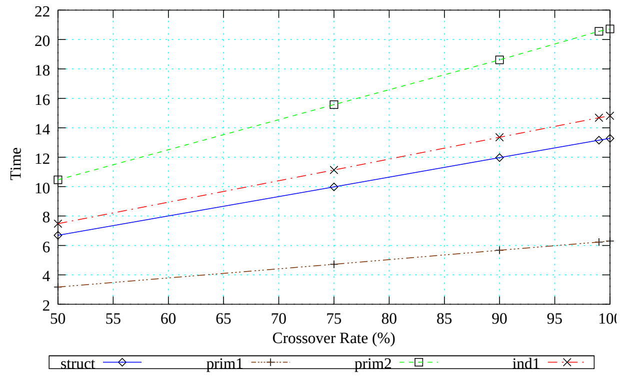


Figure 3.45: Effect of Crossover Rate on Execution time

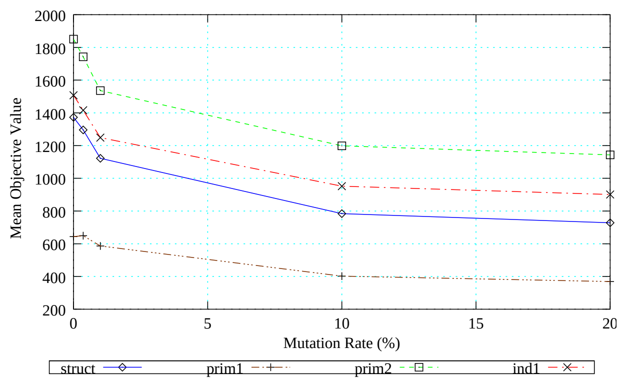


Figure 3.46: Effect of Mutation Rate on Mean Objective Value

causing them to become genetically less like the parent. This causes the system to randomly search the solution space and limits the convergence of the population resulting in lower fitness values.

From Figure 3.47 it is noticed that there is a slight decrease in execution time as the mutation rate increases. This is an indirect effect of the mutation rate and is a result of the decrease in mean fitness value. As the fitness value is decreased the number of cut nets increases resulting in lower processing time spent in calculating the fitness value.

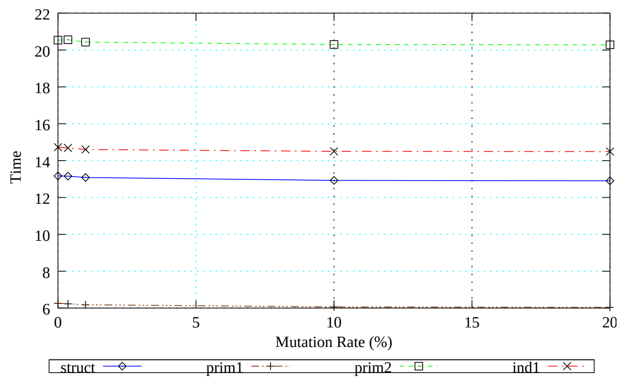


Figure 3.47: Effect of Mutation Rate on Execution Time

3.5.4 Effect of Population Size on Solution Quality

In examining the effects of the population size on the solution quality, the base case parameters were used while modifying the size of the population. The results are illustrated in Figure 3.48. It was found that increasing the population size caused an increase in mean objective value. The reason for this is that increasing the size of

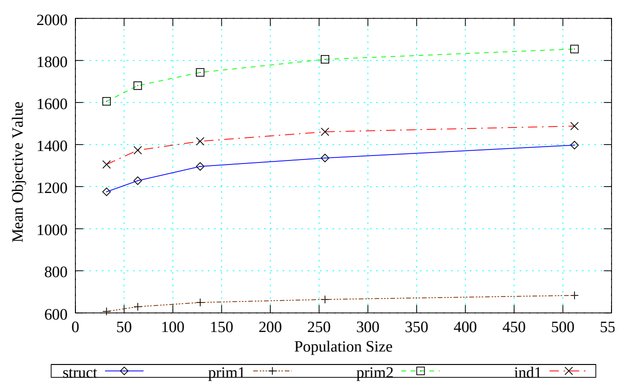


Figure 3.48: Effect of Population Size on Mean Objective Value

the population allows for a larger number of random initial chromosomes resulting in a higher probability of having good starting positions. A larger population size also allows for higher diversity within the population, shown by the standard deviation curve in Figure 3.49. Having a high diversity within the population allows

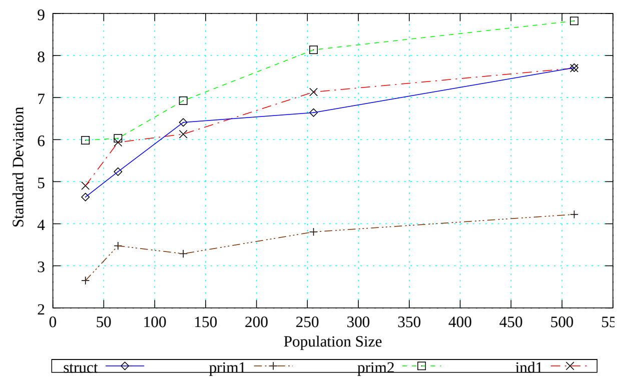


Figure 3.49: Effect of Population Size on Standard Deviation

the population to search a larger area of the solution space for good solutions.

From Figure 3.50 it can be seen that in increasing the population size results

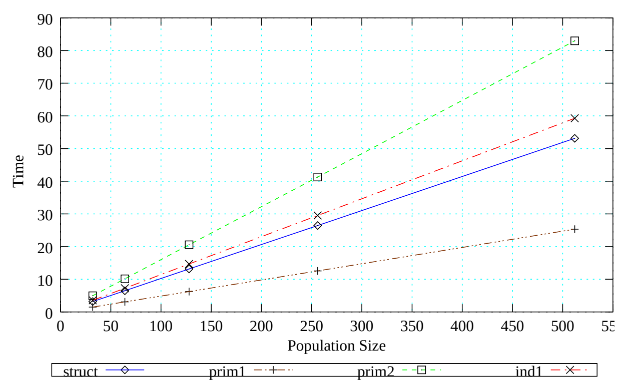


Figure 3.50: Effect of Population Size on Execution Time

in a linear increase in execution time. This can be expected since increasing the population size causes a increase in the number of offspring generated.

3.5.5 Effect of Balancing criteria on Solution Quality

In examining the effect of the balancing criteria of the system, the base case parameters were used while modifying the size of the balancing criteria. Figures 3.51 and 3.52 show that changing the size of the balancing criteria has little effect on both the mean objective value and the execution time of the system. The reason for the

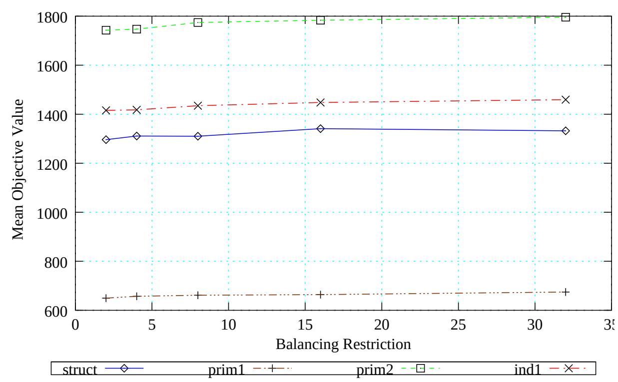


Figure 3.51: Parallel Pipeline Architecture

minimal effect is that the balancing criteria is a restriction on the system and not a tuning parameter. This means the balancing criteria will only cause an effect on the system if the criteria is broken and is not designed to assist in improving the solution quality.

3.6 Summary

In this chapter, an initial design of a pipelined Genetic Algorithm system was presented. This design was analyzed to determine bottlenecks and was then modified to improve performance. The proposed architectures were compiled and implemented

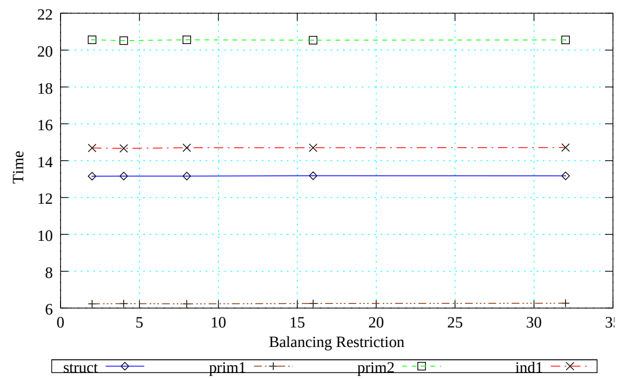


Figure 3.52: Parallel Pipeline Architecture

on the Celoxica RC1000 development platform and further analyzed to determine execution time and final solution quality obtained.

From the experimental data shown in Figure 3.40(a) it was clear that the improved hardware design had a significant performance increase over the software program developed by [Arei01] but did not perform as well as the software implementation using the same bit-wise representation. Figure 3.40(b) shows that the [Arei01] software produced better results than the hardware by roughly 13%. This difference in solution quality is attributed to the random number generator used in the hardware implementation, repair algorithm and the difference in crossover techniques.

In examining the results generated by the hardware implementation, the overall average mean solutions generated only 73.3% of the nets in the benchmark uncut. This low quality of solutions is a result of Genetic Algorithm's failure to exploit the solution space. In order for the Genetic Algorithm to become an effective search technique it must improve its capability to fine-tune the search.

Chapter 4

Local Search and Memetic Architecture

In general, most real world problems are too complex for any single processing technique to solve in isolation. The modern trend and philosophy for constructing fast, globally convergent algorithms is to combine a simple globally convergent algorithm with a fast locally convergent heuristic to form a more suitable and faster hybrid. Genetic Algorithms are well known for exploring the solution space effectively but are unable to fine tune the search. In order to improve Genetic Algorithms' search capabilities, a Local Search technique is often integrated with a Genetic Algorithm to form a hybrid called Memetic Algorithms. Accordingly, the hybrid Memetic Algorithm tends to incorporate the exploration capability of Genetic Algorithms with the exploitation features of Local Search.

Local Search heuristics are iterative techniques that improve a solution towards a local minima. This approach often uses neighbourhood search to find a better solution than the current solution. The main disadvantage of Local Search is that they get trapped in a local minimum/maximum, as shown in Figure 4.1.

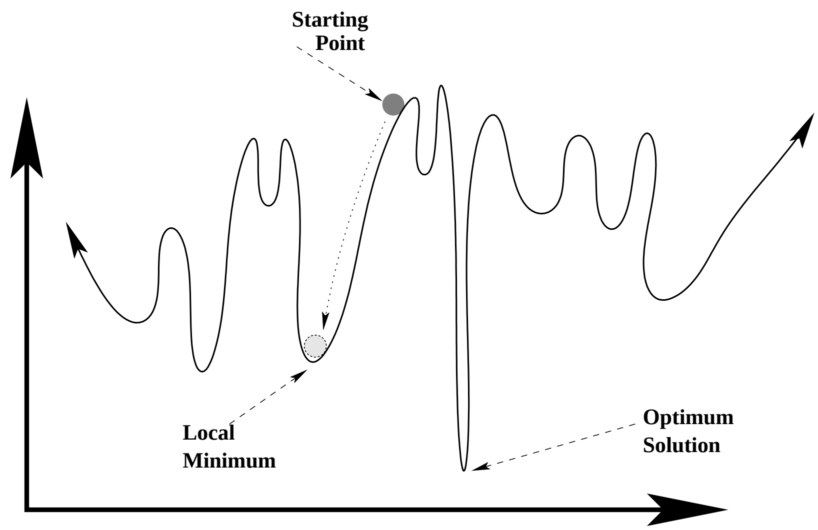


Figure 4.1: Local vs Optimum Solution

4.1 Basic Local Search Procedure

In solving the circuit partitioning problem, the goal of the solver is to maximize the number of uncut nets, as defined in section 2.4.1. Therefore, the initial design phase of the Local Search involved developing a technique which forces nets exclusively into one partition, while preserving the balance criteria. The general template for this technique is illustrated in Figure 4.2 and can be described as following:

1. **Generate Initial Solution** - Produce an initial starting point either randomly or through a constructive based technique.

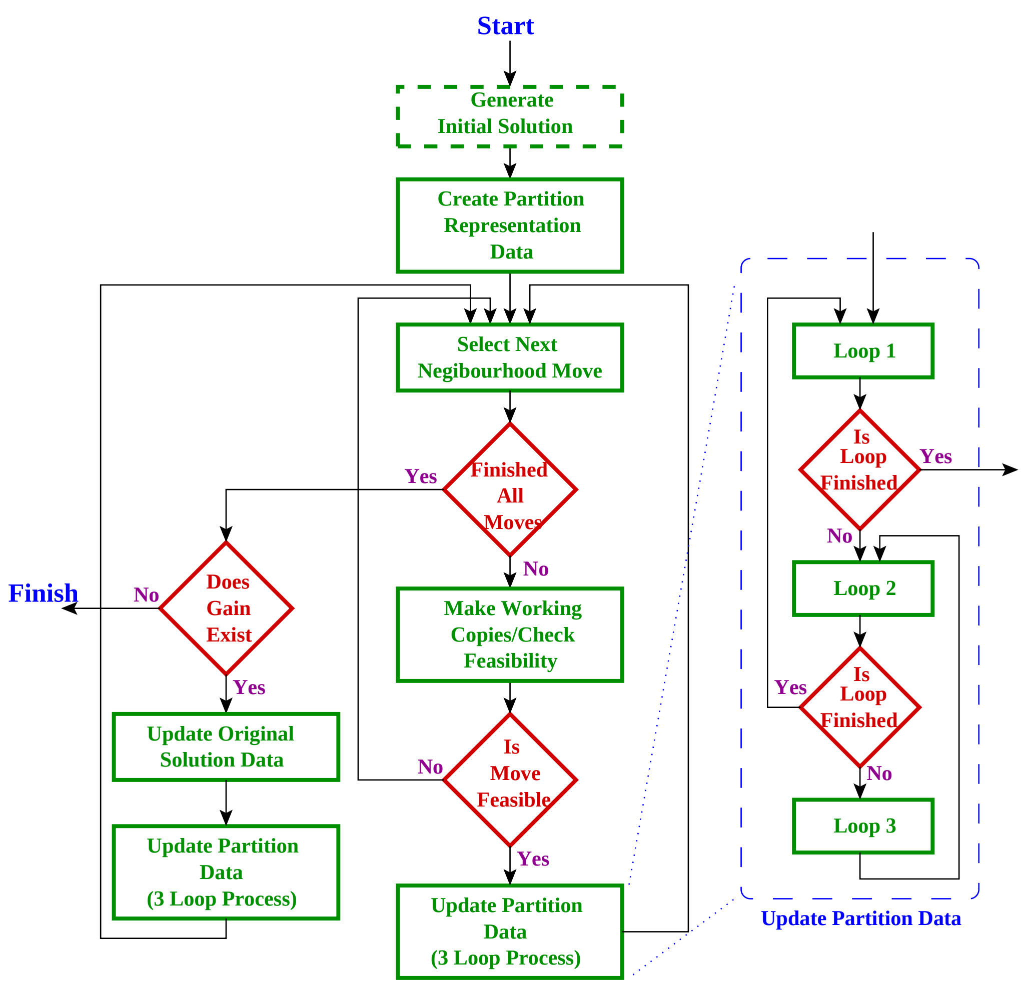


Figure 4.2: Local Search Block Diagram

2. **Create Partition Representation Data** - Determine which nets are currently uncut given an initial solution. This is established by checking all nets within the Netlist and determining which nets are absorbed within one partition. The resulting information is stored into memory allowing the system to determine with ease the status of all nets. This process is similar to the fitness calculation task within the Genetic Algorithm process described in section 3.2.3, except that the status of each net is stored for future reference instead of accumulating the number of uncut nets to obtain a fitness value.

   To maintain the cut status for each net, two arrays called "Partition Data" are used for each partition. When the system determines that a net is contained within a partition the corresponding element within the Partition Data array is set to '1'. This informs the system as to which nets are contained within each partition and allows for easy calculation of the objective value by summing '1's in both arrays. The representation of the Local Search data is illustrated in Figure 4.3. The "Solution Data" array as seen in Figure 4.3b illustrates how the six cells within the Netlist are separated equally into two partitions. The "Partition Data" (Figure 4.3c) indicates that Net_3 is completely absorbed within $Partition_0$ while Net_2 is consumed within $Partition_1$.

3. **Select Next Neighbourhood Move** - Determines the next possible neighbourhood move for the Local Search procedure. In performing the neighbourhood searching process, all nets within "Partition Data" containing a '0' value are potential neighbourhood moves, even if the net is contained entirely within the other partition. This can be attributed to the following: moving a net

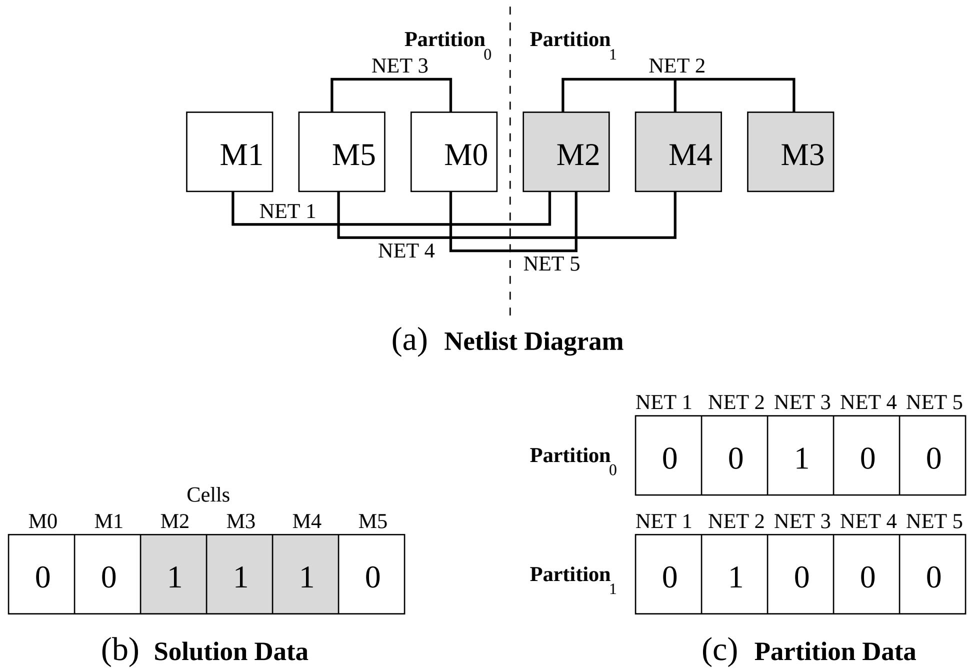


Figure 4.3: Local Search Data

that is currently absorbed within a partition may allow for achieving higher gains if absorbed by a different partition. The drawback of this approach is that the searching process becomes extremely computationally intensive.

An alternative approach that is less computationally intensive is to select nets that have a value of '0' in both arrays of the "Partition Data", as illustrated in Figure 4.4 and forcing the net to a value '1'.

4. **Perform Working Copies/Check Feasibility** - Generates a working copy of the current solution and "Partition Data" such that reversing a move can be easily achieved. Once a copy of the data is generated, the neighbourhood move is applied to the new copy of the "Solution Data" and the feasibility

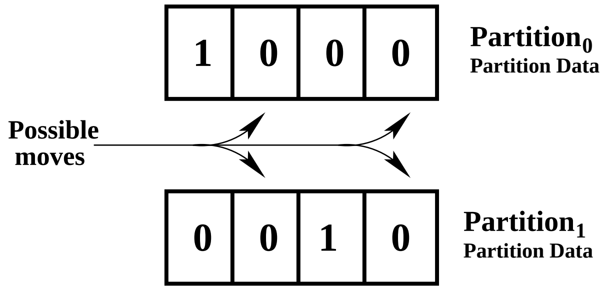


Figure 4.4: Determining Feasible Move as only cut nets

of the move is determined. In applying the neighbourhood move all modules of the selected net are transferred to the desired partition. The feasibility of a move is determined by the balancing criteria. The balancing constraint is enforced by counting the number of '1's that lie within the "Solution Data" and ensuring that equations 4.1 and 4.2 are satisfied.

$$BALANCE \geq Modules - 2 \times (Number\ of\ 1's) \tag{4.1}$$

and

$$BALANCE \geq 2 \times (Number\ of\ 1's) - Modules \tag{4.2}$$

5. **Update Partition Data (3 Loop Process)** - Determines which nets were affected by the neighbourhood move. Updating the Partition Data information is accomplished via a three loop process as shown in Figure 4.5. which demonstrates the process of updating the Partition Data when the Local

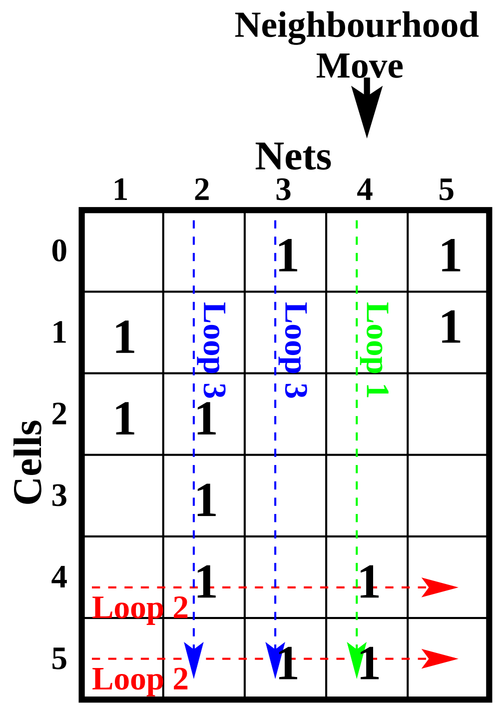


Figure 4.5: Update Partition Data process

Search forces Net_4 (as previously demonstrated in Figure 4.3) into one of the partitions.

(a) *Loop1*: Identify cells connected to a net being moved. As seen in Figure 4.5 if Net_4 were to be absorbed within a partition this step would identify $Cell_4$ and $Cell_5$ as candidates.

(b) *Loop2*: Identify all nets that are connected to the cells defined in the previous step. In Figure 4.5 this process would identify Net_2, Net_4 as being attached to $Cell_4$, and Net_3, Net_4 as being attached to $Cell_5$.

(c) *Loop3*: Determine if the cut status of these nets has changed. This is done by calculating the status of the net after the neighbourhood move has been made and comparing it to the information stored in the working copy of the Partition Data. If the neighbourhood move has caused the status of this net to change, then the information within the working copy of the Partition Data is updated to the new status and a relative gain is calculated.

6. **Update Original Cell Data** - The move with the highest relative gain is applied to the original "Solution Data". The Local Search process terminates when no positive gain can be achieved. This indicates that the heuristic is stuck in some local maxima.

4.2 Hardware Design

In designing the Local Search algorithm in hardware, the goal was to implement highly computationally intensive portions of the software algorithm in parallel to improve execution time.

Profiling the software algorithm, as shown in Table 4.1, presents three main routines that require the majority of processing. These routines consist of the **CopyData** function, performing the task of "**Make Working Copies**" of the Cell and Partition data; the **three loop update process**; and the **Count** process used to determine the feasibility of the solution. In software, the **Count** function is the most time consuming requiring around 80% of the overall execution time. The purpose of this routine is to determine the number of cells of the solution that exist in $Partition_1$ by comparing each bit within the solution data individually. Developing this operation in hardware allows for this counting process to occur in

Name of Software Function	Equivalent Functionality	% Execution Time			
		struct	prim2	prim1	chip1
Count	Count '1' for feasibility	81.31	84.79	88.71	70.00
Loop3	Update Partition Data (Loop 3)	8.64	7.13	2.69	20.00
CopyData	Make Working Copies	6.03	5.12	5.91	10.00
Loop2	Update Partition Data (Loop 2)	3.15	2.25	1.61	0.00
Loop1	Update Partition Data (Loop 1)	0.54	0.59	1.08	0.00
LocalSearch	Select Next Neighbourhood move	0.05	0.03	0.00	0.00
ApplyBestMove	Update Original Cell Data	0.00	0.00	0.00	0.00
UpdateBlocks	Create Partition Representation	0.00	0.01	0.00	0.00
Overhead		0.28	0.08	0.00	0.00

Table 4.1: Local Search Software Profile

parallel, therefore reducing the execution time.

The next computationally intensive portions of the software algorithm consist of the "Make Working Copies" and the "Update Partition Data." In optimizing these sections of code, the original aim of the system was to implement a pipeline architecture, similar to the Genetic Algorithm, to increase the system's throughput. However, several problems occur with data dependencies: the working copy of the Partition Data stored in memory cannot be altered until the "Update Partition Data" process has completed, causing the function to stall. Therefore, a true pipeline cannot be used. This results in implementing a small pipeline for the three loop update process. Figure 4.6 is a simple illustration to help describe the layout of the process. The pipeline design is expected to allow each SearchLoop to perform its task in parallel with minimal communication hence increasing throughput.

4.2.1 Local Search Memory Management

The memory banks used for the Local Search algorithm are slightly different from those used by the Genetic Algorithm. While the Local Search is searching for a

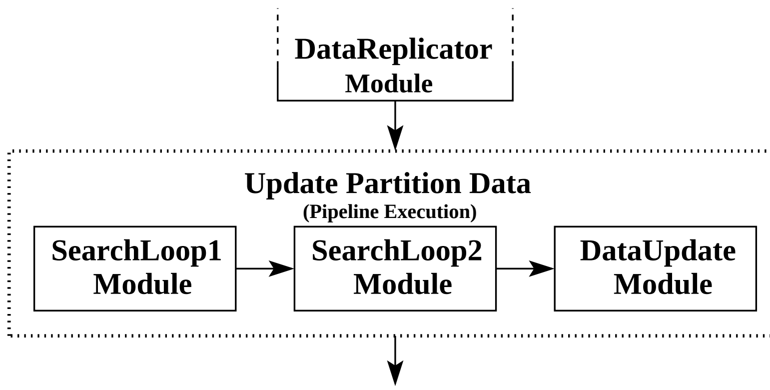


Figure 4.6: Local Search implemented in Hardware

better solution, it is necessary to keep a copy of the original data in memory in order for the best move to be applied. To accomplish this task, two memory banks are split into an original (to hold the best solution found so far) and working copies (to search for a better solution). The memory can be described as follows:

1. *Solution Data* : The Solution memory holds the current solution. It follows the same format as the Genetic Algorithm chromosomes representation, described in section 3.1.2. This memory is located at the beginning of the memory block and has an allowable size of 131072×32-bits.

2. $Partition_1$ *Data* : This memory holds information on status of nets that are completely contained within $Partition_1$. A '1' indicates that the corresponding net is uncut and lies within $Partition_1$. The memory starts at location 262144 and has an allowable size of 131072×32-bits.

3. $Partition_0$ *Data* : This memory serves the same purpose of that of $Partition_1$ Data for $Partition_0$. A '1' indicates that the corresponding net is uncut and

lies within $Partition_0$. The memory starts at location 262144 and has an allowable size of 131072×32-bits.

4. *Register Data* : The register data hold the same register information as in the Genetic Algorithm, section 3.1.2 and is located at location 523,264.

Figure 4.7 illustrates the memory map for the Local Search Algorithm.

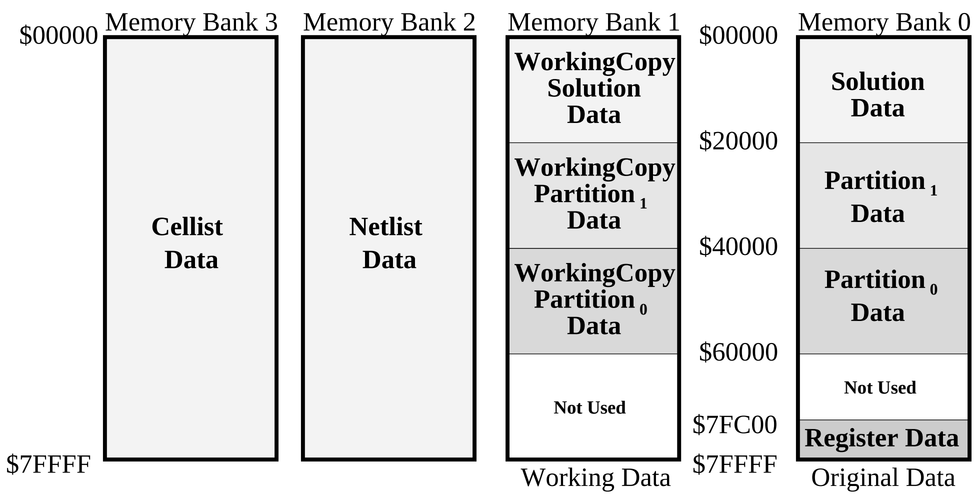


Figure 4.7: Local Search Memory Map

4.3 Local Search Design and Architecture

In order to implement the given Local Search procedure, the design is broken down into several components, as shown in Figure 4.8. Although all components are necessary in operating the local search, the bulk of the processing time occurs in copying data and modifying/updating the Partition Data once a net is absorbed within one of the partitions (ie. Partition Information Update). For this reason, pipelining and parallelism are used in an attempt to improve execution performance of these modules.

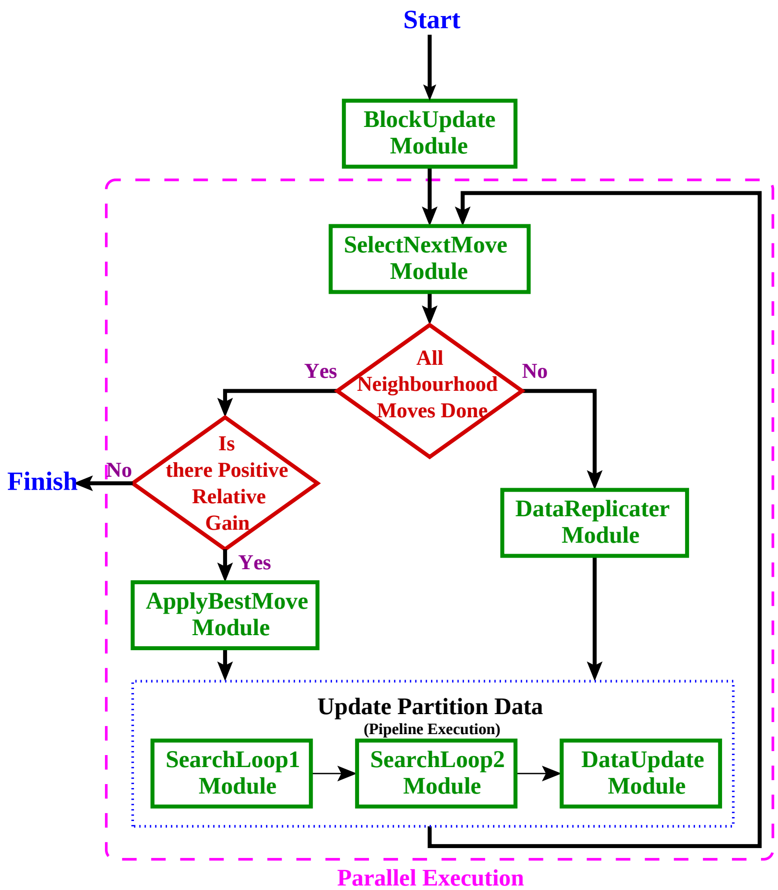


Figure 4.8: Local Search Pipeline

4.3.1 Partition-Update-Module (PU-M)

The "Partition-Update-Module" initializes the Local Search routine by generating the Partition Data given that an initial solution is provided. The Partition Data consists of two parts, $Partition_0$ and $Partition_1$, with each bit within the data representing a net of the Netlist. A value '1' value is assigned to a bit location if the corresponding net is completely contained within the given partition. This information informs the Local Search algorithm which nets are currently uncut in order to determine possible neighbourhood moves. This information also allows the system's easy calculation of the objective function value.

Pin Description

Figure 4.9 describes the pin interface between the PU-M, memory and other modules. A more detailed description of the pins can be found in Appendix B.1.

Functionality of PU-M

The task of PU-M is to create the initial values for the Partition Data. Once the system initiates the module, by driving pin *UpdateEnb* high, the process shown in Figure 4.10 starts.

The process begins by reading a byte of data for the first net in the Netlist and a byte of data from the initial solution, placing them both into registers (see Figure 4.11). These registers are then compared to see if the net is cut or not. In determining if the net is cut, at least one module in each partition is connected to the net.

The process of reading and comparing the data is continued until the entire net is compared with the initial solution and all cells lie within one partition or until it is determined that a net is cut (no further processing is required). The technique used to determine the status of the net is similar to the fitness calculation in the Genetic Algorithm.

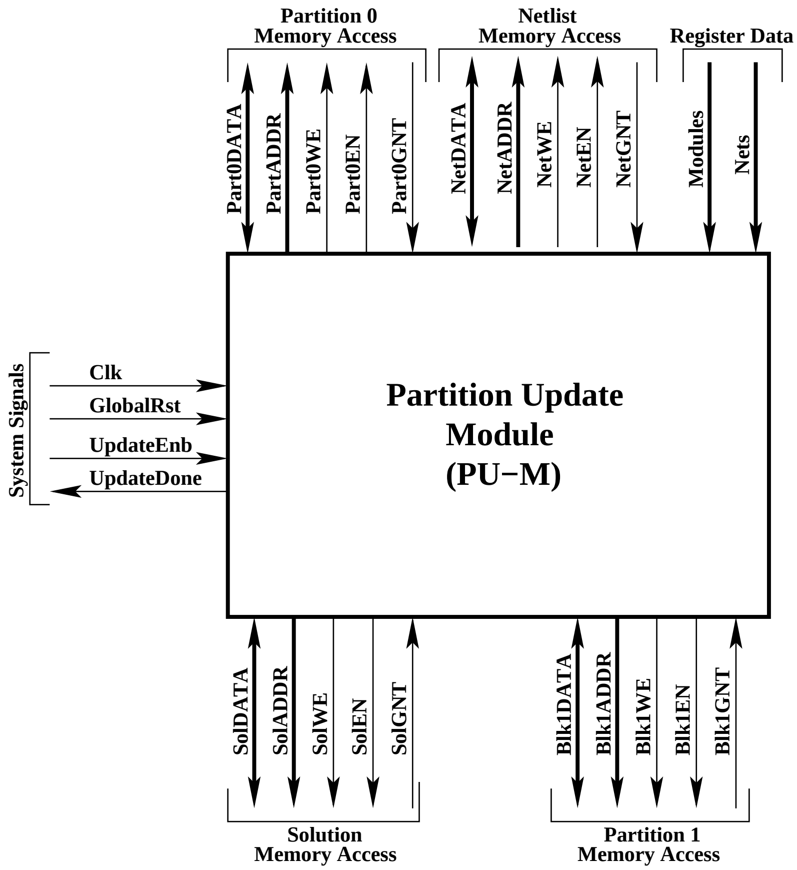


Figure 4.9: Partition-Update-Module Signal Diagram

If a net lies entirely in one of the two partitions, the corresponding bit in the Partition Data is updated with a '1' value. This process is further illustrated in Figure 4.11. A bit shifter is used to determine the bit that is set to '1' within the Partition Data. If the net is uncut, this bit is stored in the Partition Register of the block within which the net lies. When the registers are completely updated their content is written to memory.

This process is repeated for all nets within the Netlist in order to complete the Partition Data for the system. Upon completion of processing all nets, the

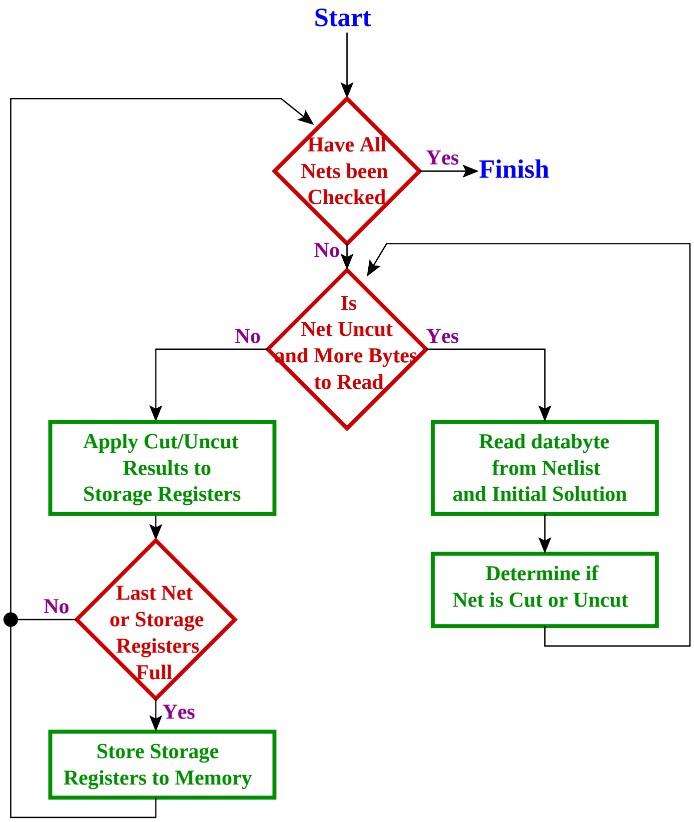


Figure 4.10: Partition-Update-Module (PU-M) Block Diagram

UpdateDone pin is driven high to inform the system that the process is complete.

4.3.2 Select-Next-Neighbourhood-Move-Module(SNNM-M)

The "Select-Next-Neighbourhood-Move-Module" determines the next possible neighbourhood move when searching through the search space. The process loops through the Partition Data and selects the next possible move as being any net that is not completely contained within the partition. This is done by selecting all '0' values as possible neighbourhood moves, as illustrated in Figure 4.12

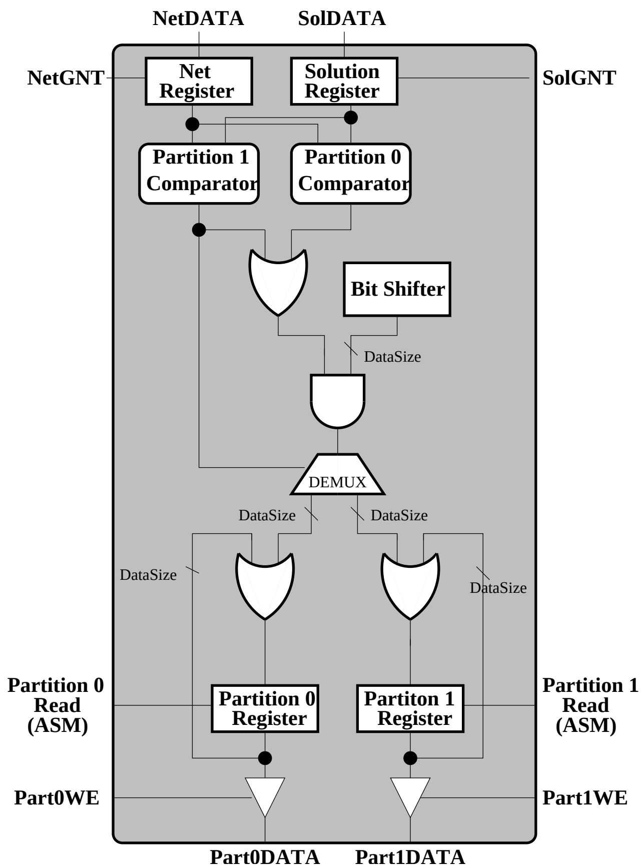


Figure 4.11: Update Logic for Partition Data

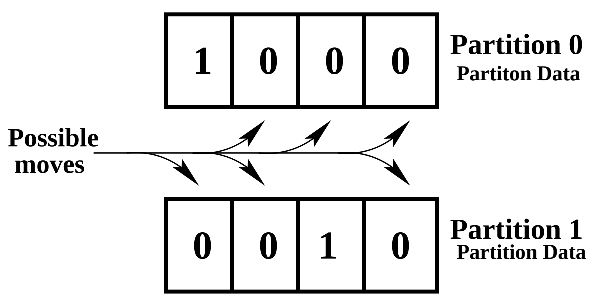


Figure 4.12: Determining Feasible Move

Pin Description

Figure 4.13 describes the pin interface between the SNNM-M and memory/other modules. A description of the pins can be found in Table B.2.

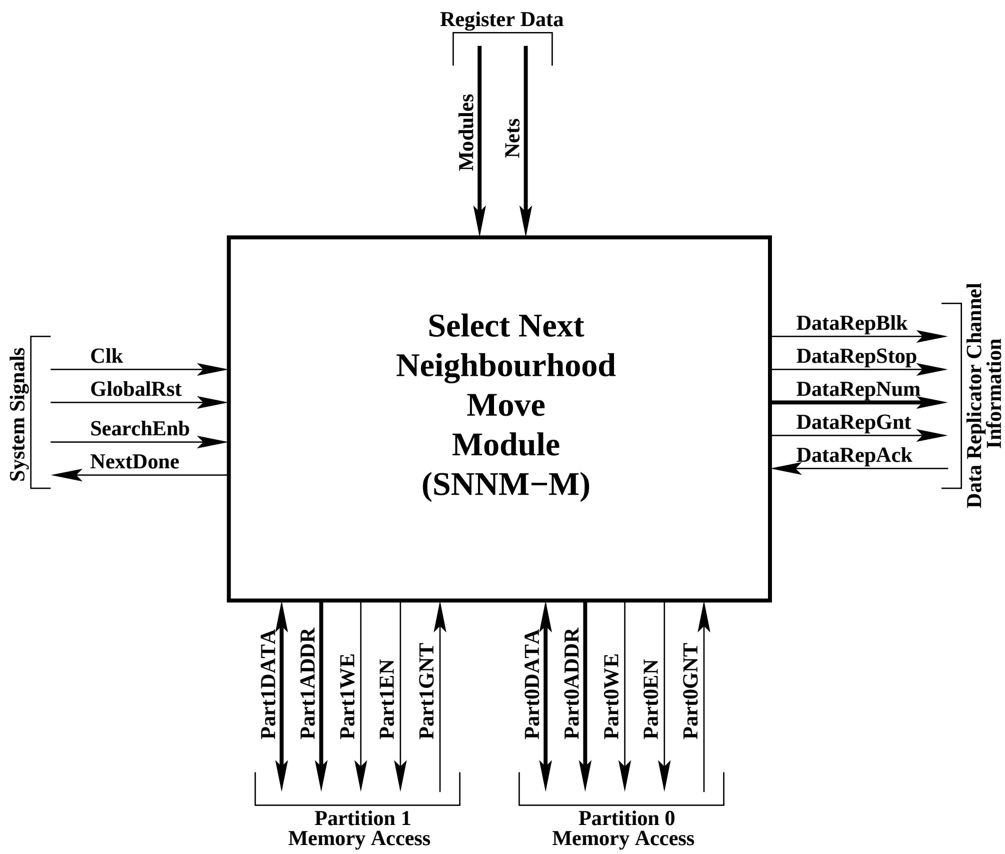


Figure 4.13: Select-Next-Neighbourhood-Move-Module Signal Diagram

Functionality of SNNM-M

The task of the SNNM-M is to select the next potential neighbourhood move based on information stored in the Partition Data. Once the system initiates the selecting procedure, by driving the *SearchEnb* pin high, the process shown in Figure 4.14 is started.

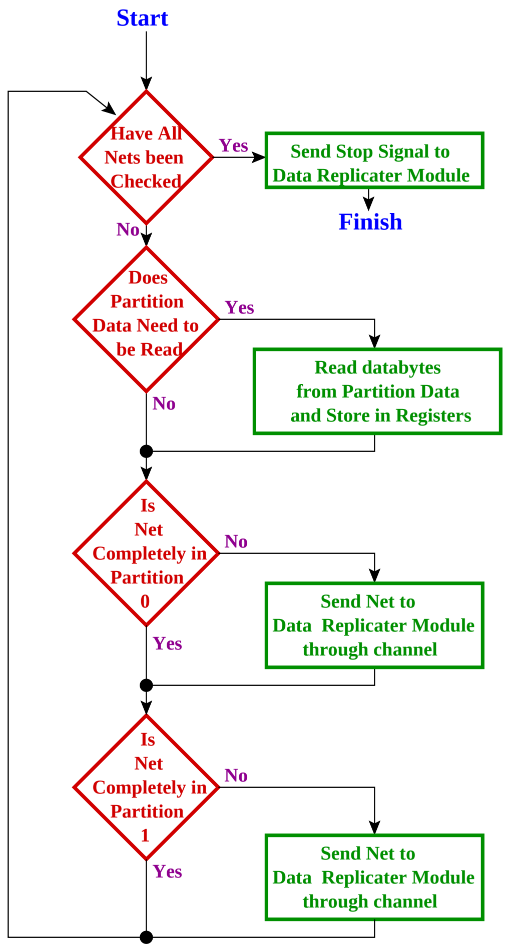


Figure 4.14: Select-Next-Neighbourhood-Move-Module (SNNM-M) Process Flow

The process begins by reading the first byte of data from each Partition Data and storing them into registers. The purpose is to identify nets that are entirely contained within a partition rendering them invalid neighbourhood moves. A detailed internal logic diagram of the SNNM-M is presented in Figure 4.15. A net is considered a candidate if it is found to be cut or not contained within the partition. If this is a potential move the net and partition are passed to the “Data-Replicator-

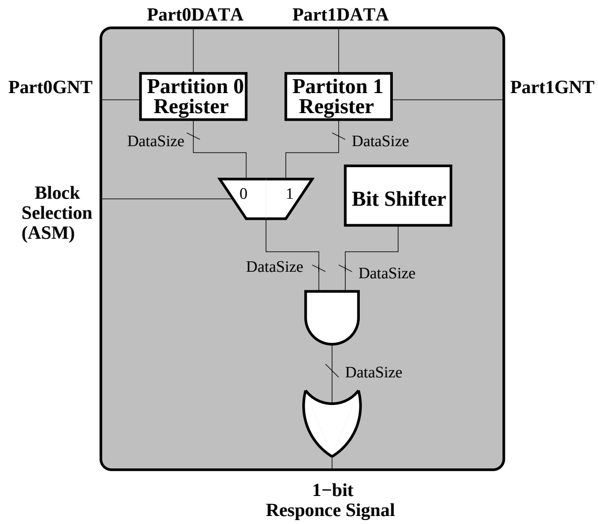


Figure 4.15: Select neighbourhood move

Module" (DR-M) through the Copy Data channel and the process repeats with the next net. The SNNM-M process is repeated until all possible neighbourhood moves have been determined.

Once all neighbourhood moves have been tested, a high is sent on the *NextDone* pin to inform the system that the SNNM-M has terminated. At this time a stop signal is sent through the outgoing channel to inform the DR-M that all moves have been made.

4.3.3 Data-Replicator-Module (DR-M)

The "Data-Replicator-Module" accomplishes three tasks. The first is to make working backup copies of the Solution and Partition Data. This is to ensure that the original information stored in memory is not altered during the searching process. The second task is to apply the neighbourhood move, selected by the SNNM-M, to the working copy of the Solution Data. The last task is to determine if this neighbourhood move selected by the SNNM-M is feasible. This is done by counting the number of '1's that are contained within the working copy of Solution Data and determining if they meet the balancing criteria. In order for the move to be considered feasible, equations 4.1 and 4.2 must be satisfied. If the move is infeasible the module receives a new neighbourhood move from the Copy Data Channel; otherwise, the Partition Data Update process is executed to determine the relative gain of the move.

Pin Description

Figure 4.16 describes the pin interface between the DR-M and memory/other module. A description of the pins can be found in Appendix B.3.

Functionality of DR-M

The task of the DR-M is to create backups of the current data and check the feasibility of neighbourhood moves. Once the system initiates the copying procedure, by driving the *SearchEnb* pin high, the process shown in Figure 4.17 begins.

Upon starting the process, the system remains in an idle state until all necessary information (i.e which neighbourhood move to make) is received on the Copy Data channel. This information includes: (i) the target net to be moved, (ii) the target partition.

Following the retrieval of the net/partition information, the process begins creating working copies of the original information. While generating the working copy of the Solution Data, the neighbourhood move is applied and the number of

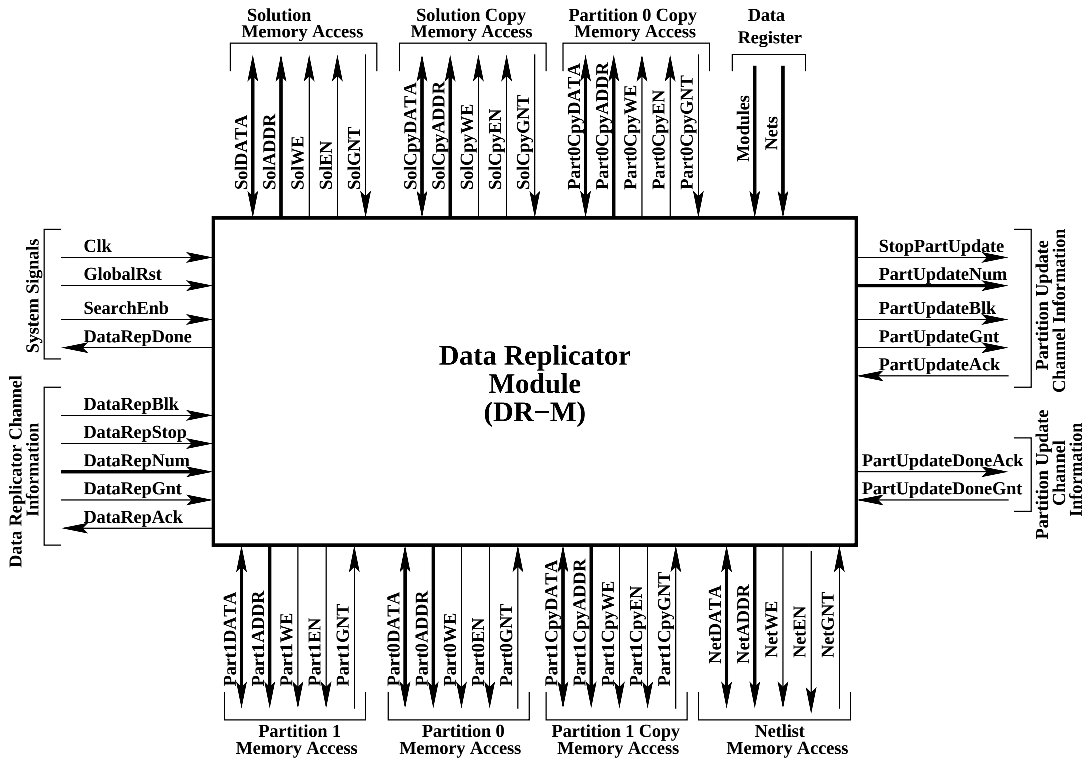


Figure 4.16: Data-Replicator-Module Signal Diagram

cells in $Partition_1$ are counted so that feasibility can be determined. The logic for generating the working copy of Solution Data is illustrated in Figure 4.18.

Once the process has completed generating the working copies of the data, it must determine the feasibility of the move. If the move is feasible, then the net location is passed to the "Partition-Data-Update" (PDU) to update the affected nets in the Partition Data and simultaneously calculate the relative gain. While the PDU is executing, the DR-M enters an idle state waiting for the update process to complete. When the PDU process has completed, the relative gain is compared with the best relative gain found so far. If the newly calculated relative gain is better than any previous gain, the information on the current neighbourhood

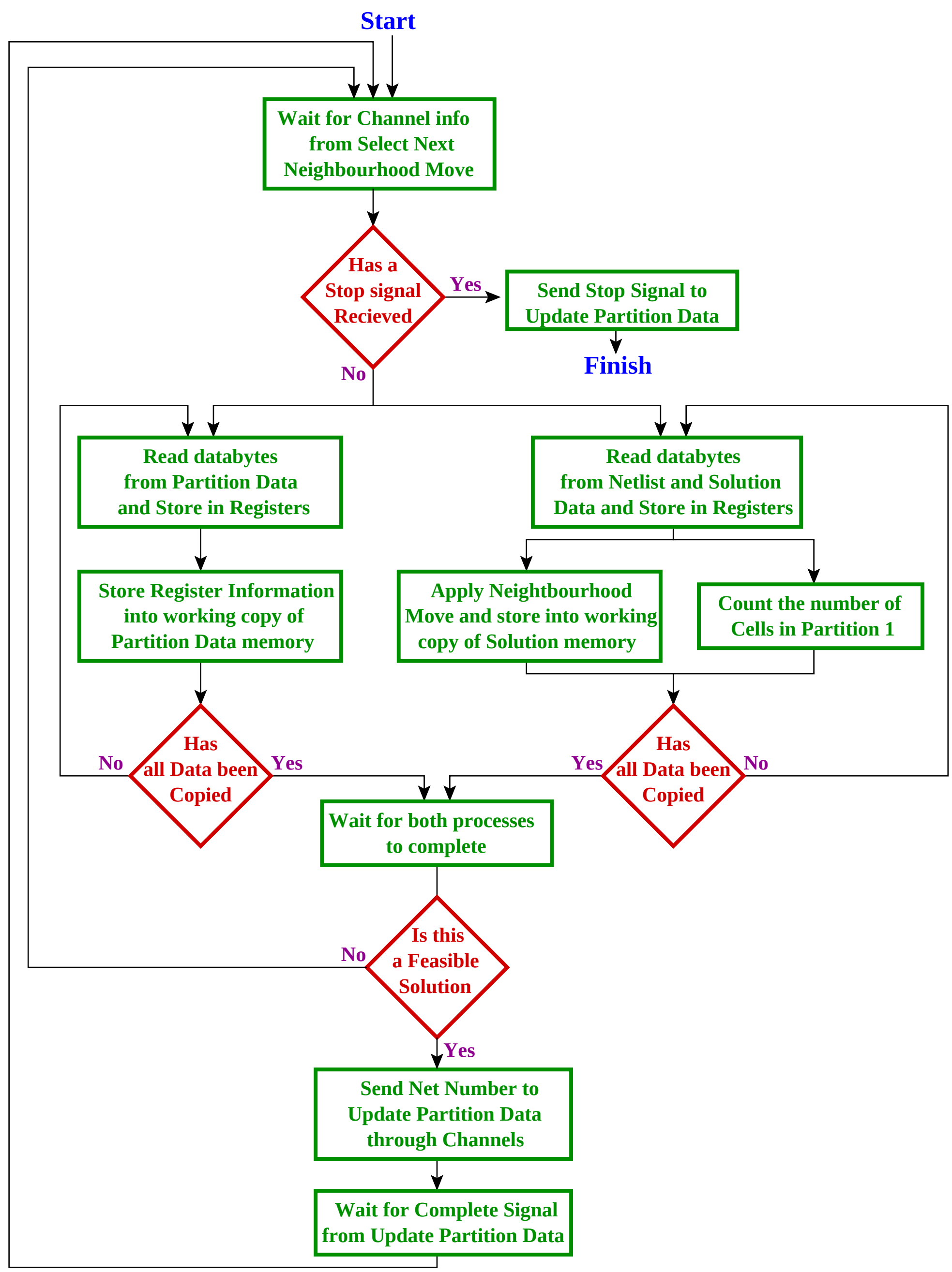


Figure 4.17: Data-Replicator-Module Block Diagram and flow

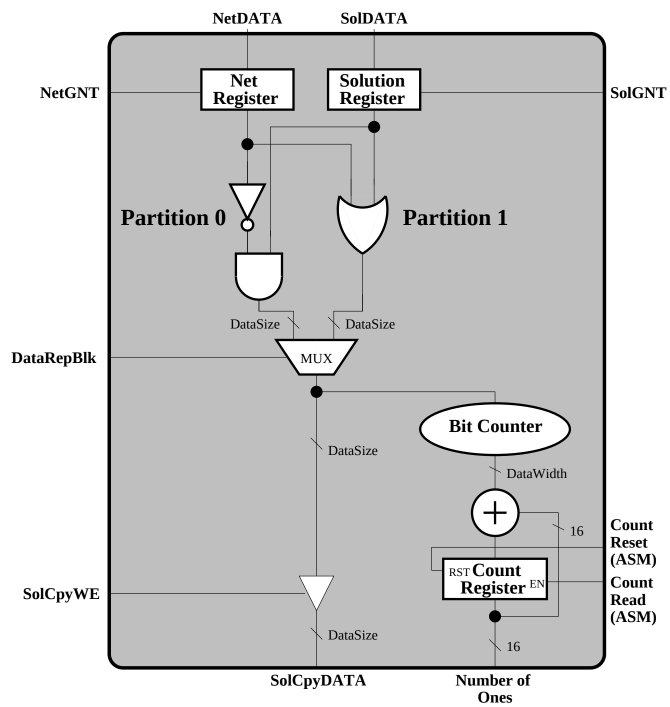


Figure 4.18: Applying Neighbourhood move to Solution Copy

move is stored as a best potential move. The DR-M repeats its process with new neighbourhood moves until a stop signal is received on the Copy Data channel and a high is placed on the *DataRepDone* pin to inform the system that the DR-M is finished.

4.3.4 Search-Loop-Module (SL-M)

The "Search-Loop-Module" is used in two instances which have virtually the same functionality but applied to different data. The goal of the SL-M is to determine affected nets after attempting to make a neighbourhood move. In finding these nets, the first SL-M must determine which cells are connected to the net being forced into a partition. These cells are found by searching through the Netlist for a given net to determine which bits contain '1' value. These locations represent the cells that have the potential of being moved from one partition to the other during the neighbourhood move process.

The second SL-M then determines other nets connected to these cells. This follows a similar procedure as the previous instance but is applied to the Cellist (a duplicate of the Netlist data but referenced by modules) to find nets connected to these cells.

Pin Description

Figure 4.19 describes the pin interface between the SL-M and memory/other modules. A description of the pins can be found in Appendix B.5.

Functionality of SL-M

The task of the SL-M is to search for nets/cells affected by the neighbourhood move. Once the system initiates the searching procedure, by driving the *UpdtEnb* pin high, the process shown in Figure 4.20 is started.

Upon starting the process, the system remains in an idle state waiting to receive the location of the net/cell from the Loop In channel. When the elements of the Netlist/Cellist have been identified, the searching process can commence. The searching criteria browses through the net/cell entry and identifies bits that contain a '1' value.

This searching procedure is accomplished by initially loading bytes of the Netlist or Cellist data into a register and determining if this register is zero. If a non-zero

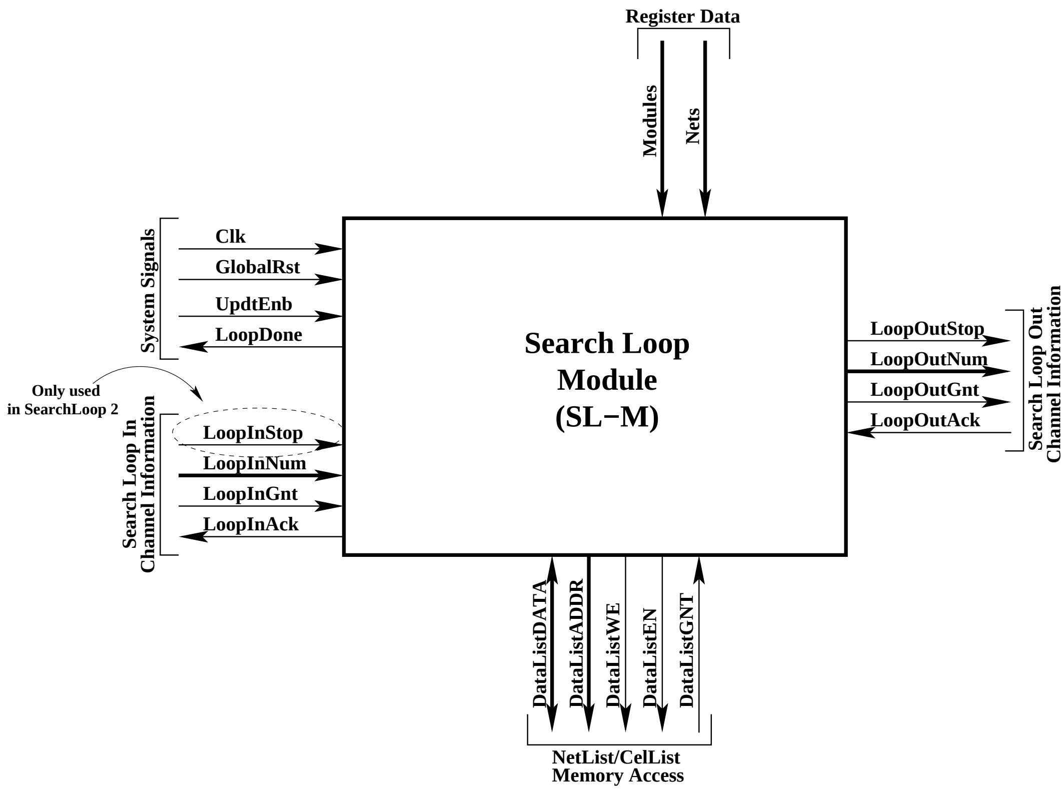


Figure 4.19: Search-Loop-Module Signal Diagram

value occurs, then there must be at least one bit within the register that contains a '1' value. To find the location of this bit a logical right shift is applied to the register until the least significant bit of the register is a '1'. This location of the net/cell is then passed to the next module through the Loop Out channel. This process is repeated until the locations of all '1' values have be found within the list entry.

A high is then placed on the *LoopDone* pin to inform the system that the SL-M has terminated.

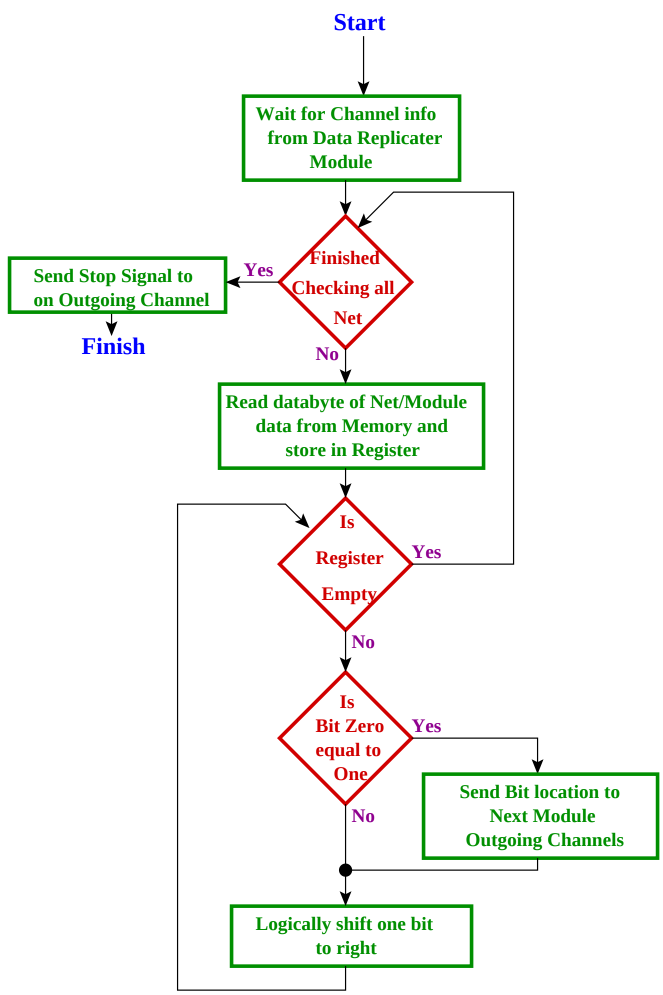


Figure 4.20: Search-Loop-Module (SL-M) Block Diagram and flow

4.3.5 Data-Update-Module (DU-M)

The main task of "Data-Update-Module" is determining both the status of the nets affected by the neighbourhood move as well as the relative gain of the move. This is accomplished by looping through the affected nets and the Solution Data to determine if these nets have been completely absorbed into the partitions. If a net was cut before the move and is now contained exclusively within a partition, the relative gain for this neighbourhood move is increased. On the other hand, if a net was previously uncut and due to the move has become cut, the relative gain decreases.

Pin Description

Figure 4.21 describes the pin interface between the DU-M and memory/other modules. A description of the pins can be found in Appendix B (Table B.6).

Functionality of DU-M

The task of the DU-M is to determine the status of effected nets and update the Partition Data accordingly. Once the system initiates the updating procedure, by driving the *UpdtEnb* pin high, the process shown in Figure 4.22 is initiated.

Upon starting the process, the system remains in an idle state waiting for information regarding a potentially affected net to be passed into the system through channel communication. Once such information has been received, the system begins the process of determining the status of this net. This process follows a similar procedure as that of the PU-M.

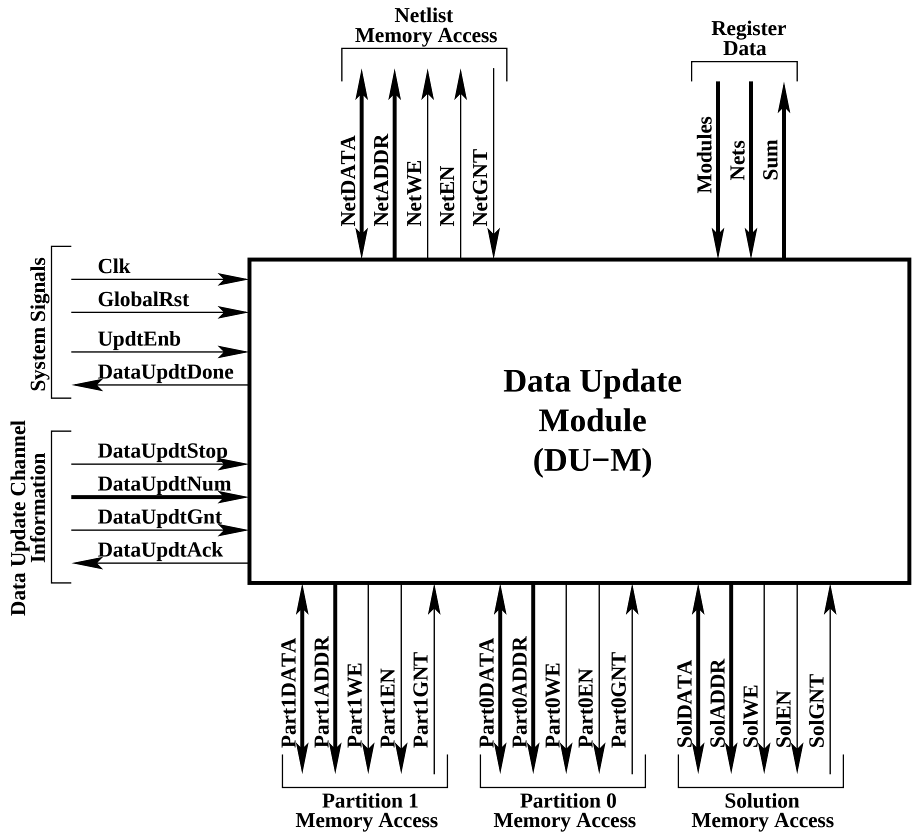


Figure 4.21: Data-Update-Module Signal Diagram

The system reads a byte of data from the net entry of the Netlist and a byte of data from the Solution Data, placing them both into registers. Once the data is stored into the registers they are compared to see if the net becomes cut. The process of reading and comparing the data is continued until the status of a net is determined (i.e cut/uncut).

When the status of a net is determined, it is necessary to determine the previous status of this net and whether the neighbourhood move has changed this status. This is accomplished by reading the previous status of the net from the Partition

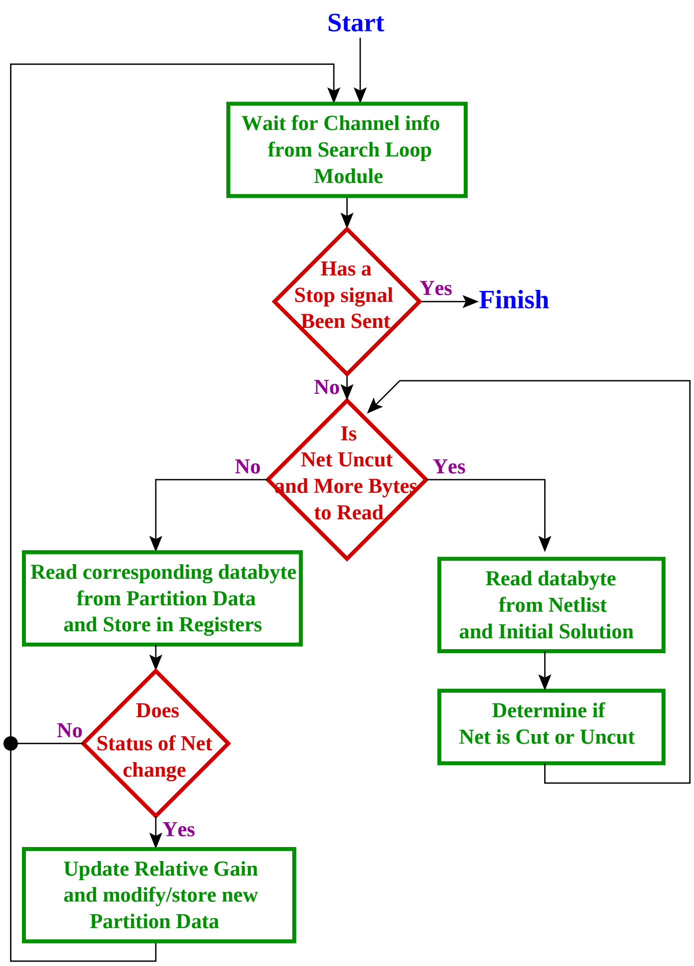


Figure 4.22: Data-Update-Module Block Diagram

Data.

Four possible cases can arise which would cause the status of the net to be affected, resulting in changing the relative gain of the system.

1. *A net previously cut is currently completely absorbed by* $Partition_1$

2. *A net previously absorbed by* $Partition_1$ *is currently cut*

3. *A net previously cut is currently completely absorbed by* $Partition_0$

4. *A net previously absorbed by $Partition_0$ is currently cut*

If any of the four cases above occur, then the system must update the Partition Data to the effects of the neighbourhood move. The relative gain must also be modified to account for the increase or decrease in gain for these four situations. Any net that becomes absorbed into a partition, will increase the relative gain and any net that becomes cut will decrease the relative gain.

When all affected nets have been tested and the Partition Data has been updated, a high signal is placed on the *DataUpdtDone* pin to inform the system that it has completed its process.

4.3.6 Apply-Best-Move-Module (ABM-M)

The final module in the Local Search Architecture is the "Apply-Best-Move-Module" which applies the best neighbourhood move to the original Solution Data. This is done by modifying the data so that the best net move becomes uncut and is absorbed within the determined partition. Once the best move has been applied the Partition Data for this move must be updated. This is done by executing the PDU process on the original Partition Data.

Pin Description

Figure 4.23 describes the pin interface between the ABM-M and memory/other modules. A description of the pins can be found in Appendix B (Table B.7).

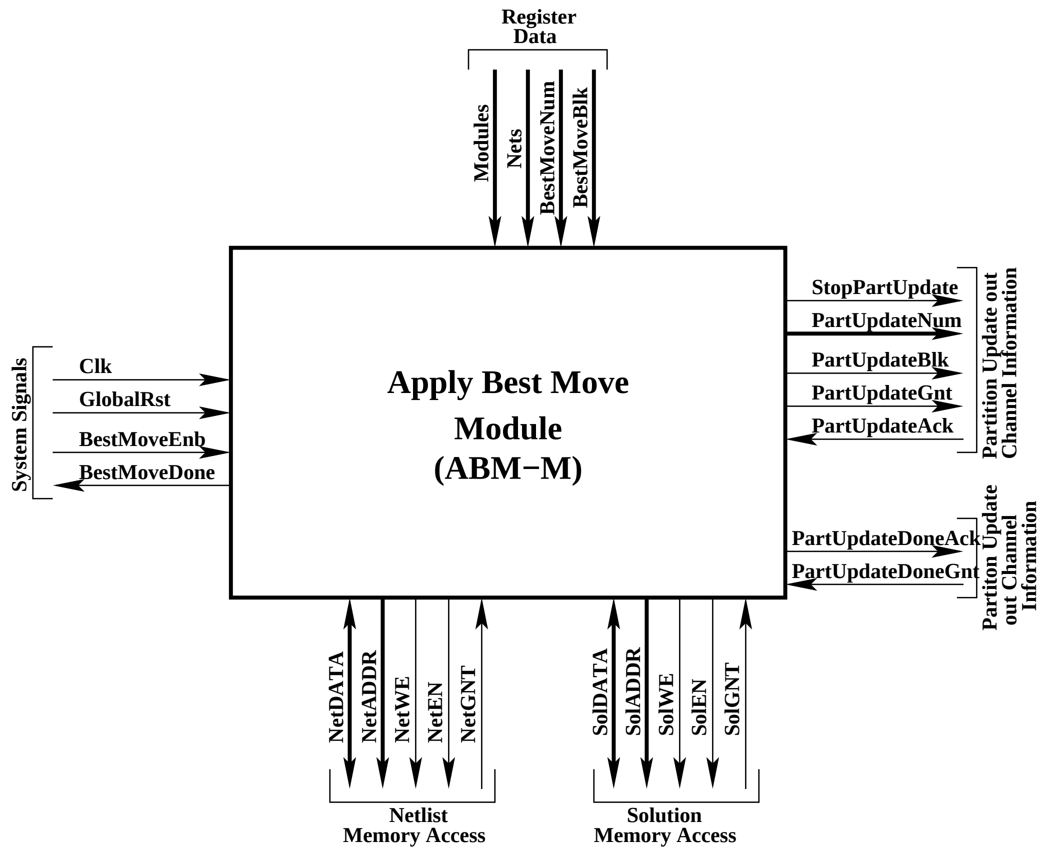


Figure 4.23: Apply-Best-Move-Module Signal Diagram

Functionality of ABM-M

The task of the ABM-M is to apply the best net move to the original Solution Data. Once the system initiates the updating procedure, by driving the *BestMoveEnb* pin high, the process shown in Figure 4.24 is initiated.

The system begins by reading a byte of data from the best net information in the Netlist. This byte is then applied to the original "Solution Data". The process is repeated until all bytes of Solution Data have been updated to incorporate the best move into the given partition.

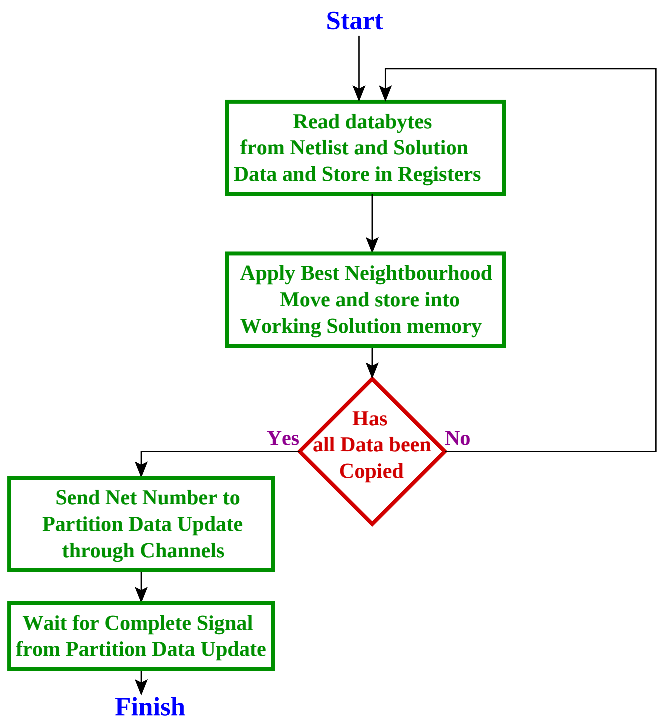


Figure 4.24: Apply-Best-Move-Module (ABM-M) Block Diagram

Following the update of Solution Data, the ABM-M executes the “Partition-Data-Update” process to update the Partition Data and incorporate the new move. During the update process, the system remains in an idle state waiting for all three PDU modules to complete their processing task simultaneously. A high signal is then placed on the *BestModeDone* pin to inform the system that the process is completed and that it may begin searching for another neighbourhood move.

4.4 VHDL vs Handel-C implementation of Local Search Architecture

In developing the above design, two different design languages were used: a high-level language (Handel-C) and a low-level language (VHDL). The main objective was to compare the difference in efficiency between the architectures. Prior to development little was known of either language; however, familiarity with ISO-C did exist. The development and debugging stages of the VHDL architecture took nearly five weeks to complete, creating almost 8,000 lines of code. This was due to the lack of experience with the language. The goal while designing the architecture was to achieve proper functionality, with minimal time spent on improving bottlenecks. The development and testing time of the Handel-C Local Search took roughly one week, $\frac{1}{5}$ of the time of the VHDL architecture, while creating 1,400 lines of code.

4.4.1 Memory Management

As described in section 2.2, in order to communicate with off-chip memory, Handel-C requires an internal clock of $\frac{1}{4}$ the frequency of the external clock. This is to allow the system to execute the required signaling to communicate with the off-chip memory. The drawback was that all non-memory commands were executing at a fraction of their potential frequency. Unlike the Handel-C, VHDL utilizes the full external clock with the use of multiple clock cycles to execute the required signal communication with the memory. Consequently all commands are able to operate at their full external clock potential.

One problem found in creating the architecture was conflicting memory accessing. Handel-C handles these conflicts with semaphores which protect critical sections of the architecture. The drawback of using semaphores is that they require one extra internal (four external) clock cycle. Due to intensive memory usage the semaphores tend to slow down the architecture by a factor of two.

The VHDL protects memory conflicts by using a priority state machine. The state machine grants memory access to different components without requiring the one clock setup/release needed by semaphores. Priority is given to different modules based on the status of the state machine to ensure that each component has equal access to the memory. An example of the priority state machine can be found in Figure 4.25 and Table 4.2. However, a flaw was identified in the VHDL

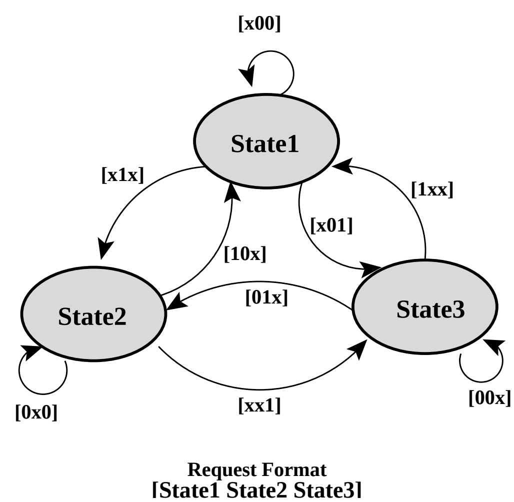


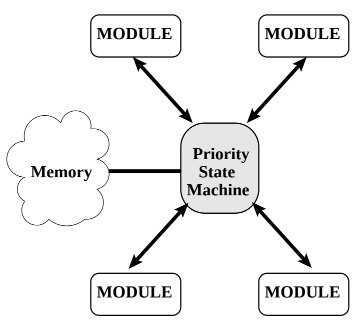
Figure 4.25: Priority State Machine

based architecture. Initially the VHDL architecture was designed such that each module might access its own dedicated memory lines. These memory lines were then combined through the priority state machine at the top level to control the access of different memory banks. Upon completing the VHDL design and analyzing the

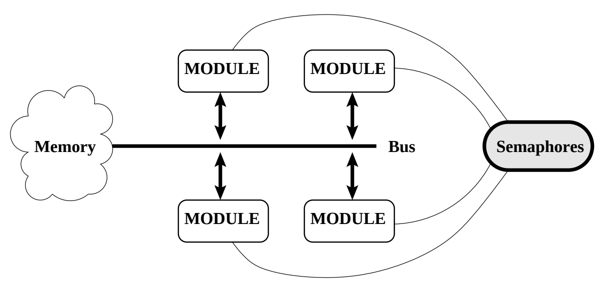
Current State	Request Values			Next State
	State1	State2	State3	
State1	x	1	x	State2
	x	0	1	State3
	x	0	0	State1
State2	x	x	1	State3
	1	x	0	State1
	0	x	0	State2
State3	1	x	x	State1
	0	1	x	State2
	0	0	x	State3

Table 4.2: Priority State Machine Truth Table

delays, it was determined that using tristate busses for memory accessing, as is used by Handel-C, would more likely improve the net delays. As shown in Figure 4.26(a) the methodology used for implementing the Local Search design in VHDL requires much more logic and routing resources than the Handel-C methodology, shown in Figure 4.26(b). It would be expected that this extra logic would have a negative effect on the clock frequency.



(a) VHDL



(b) Handel-C

Figure 4.26: Memory Communication

4.4.2 Resources

In compiling both architectures, the Celoxica DK Suite software and the Xilinx ISE tools were configured to optimize for speed and with the highest effort. Table 4.3 shows the resources used for each architecture.

Logic Utilization	Total Avaiable	VHDL LS		Handel-C LS	
		Amount	% Total	Amount	% Total
Number of Slice Flip Flops	38,400	989	2 %	1,379	3 %
Number of 4 input LUTs	38,400	3,131	8 %	2,640	6 %
Number of occupied Slices	19,200	1,744	9 %	2,118	11 %
Total equivalent gates		32,515		30,659	

Table 4.3: VHDL vs Handel-C Local Search Resources

From this table it can be seen that fewer resources were used by the VHDL based architecture. These results are expected since VHDL places more emphasis on designing the hardware to perform a specific task and does not generalize like Handel-C. Also if the memory management was developed using common bus technique, as suggested in section 4.4.1, the VHDL design could be further improved and fewer resources would have been used.

4.4.3 Delay Calculations of Local Search Architecture

In determining the timing for each of the two designs, the Xilinx timing analyzer was used. Unfortunately, there have been previously documented problems in determining net delays for Handel-C designs. As specified by Celoxica Support[Supp02], "results of the Xilinx timing analyzer (is) not always relevant". This was found to be true for the Handel-C Local Search architecture, which specified that the

"Minimum period is 103.618ns". From experimentation, the maximum successfully operated frequency was found to be 87MHz (or 11.5ns).

From the support document [Supp02], the problem can be attributed to how parameters are passed through functions. In Figure 4.27 it can be seen that the logic for passing parameters consists of a static wire, a multiplexor and a register. The static wire is utilized for cases where the parameters are used within the first

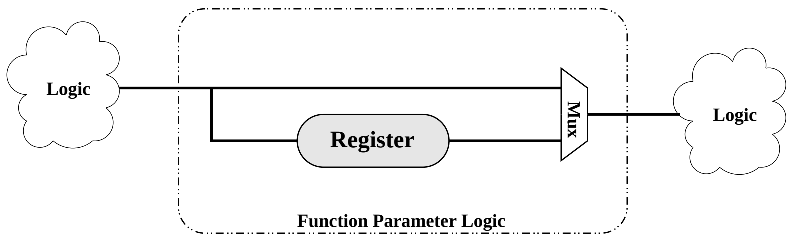


Figure 4.27: Handel-C Parameter Passing

command of the function. This is to bypass the one clock cycle delay of latching the values into the register. The registers allow access to the parameters at other clock cycles within the function. Although the static wire is only used within the first cycle of the function, the "Place And Route" (PAR) tools consider it still a valid path throughout the entire function. Therefore, the delay for the entire path is based on: (i) the logic that generates the initial parameters, (ii) the delay of the static wire and (iii) the delay of any final logic that utilizes this parameter's value. This means that the static path is calculated but is most likely never used and is, therefore, irrelevant to the timing of the design.

Determining the delay time for the VHDL architecture was straight forward. In compiling the design it was found that the optimum design was generated with minimum timing constraints. The results of the delay timing are shown in Tables 4.4

and 4.5. The maximum delay times for the VHDL design occurred as the memory

	Delay Time
Maximum Address Line Delay	24.498 ns
Maximum time for Read/Write	10.000 ns
Maximum Return Data Path Delay	42.617 ns
Total Read Delay	77.115 ns (12.9 MHz)

Table 4.4: VHDL Memory Read Timing

	Delay Time
Maximum Address/Data Delay	56.960 ns
Maximum time for Read/Write	10.000 ns
Total Write Delay	66.90 ns (14.9 MHz)

Table 4.5: VHDL Memory Write Timing

attempted to read/write to memory. In designing the architecture, communication with the off-chip memory is executed in three clock cycles to allow for necessary signal timing. A more accurate frequency calculation based on data from Table 4.4 is then given by: 12.9MHz $\times$ 3 = 38.7MHz. Although this is the theoretical calculated frequency, through experimentation the maximum allowable frequency (while still obtaining correct solution results) is 44MHz. This difference in frequency may be a result of the read timing being less than 10ns or that the net delay calculated by the timing analyzer is based on the worst case scenario for the Virtex-E FPGA chip causing the actual delay to be less than reported.

In examining the delays of the two architectures, it can be shown that the delay time for the Handel-C designs is significantly smaller than that of the VHDL implementation. As discussed in section 4.4.1, this is most likely caused by the extra logic used by the address and data lines of the VHDL architecture.

4.4.4 Timing Results of Local Search Architecture

In comparing the final performance of the two architectures, it was found that the VHDL based implementation significantly out-performs the Handel-C counterpart while operating at $\frac{1}{2}$ the frequency. Timing results can found in Table 4.6 and Figure 4.28. The improvement in execution time is attributed to the lack of semaphores

Benchmark	Handel-C	VHDL	Improvement
pcb1.dat	0.000 s	0.000 s	0%
frac.dat	0.031 s	0.016 s	194%
chip4.dat	0.122 s	0.94 s	130%
chip1.dat	0.247 s	0.140 s	176%
prim1.dat	4.700 s	2.641 s	178%
struct.dat	48.500 s	31.875 s	151%
ind1.dat	64.9 s	41.172 s	158%
prim2.dat	216.4 s	100.906 s	214%
Average			172%

Table 4.6: Execution Time of Development Languages

within the VHDL architecture and that non-memory dependent commands operate at the external clock frequency.

4.4.5 VHDL vs Handel-C: A comparison

In comparing the VHDL and the Handel-C based architectures we concluded that the VHDL generated significant improvements in both resources used by the FPGA and in execution time. In further analyzing the architectures, if modifications were made to allow for busses to be used for address and data lines, a further decrease

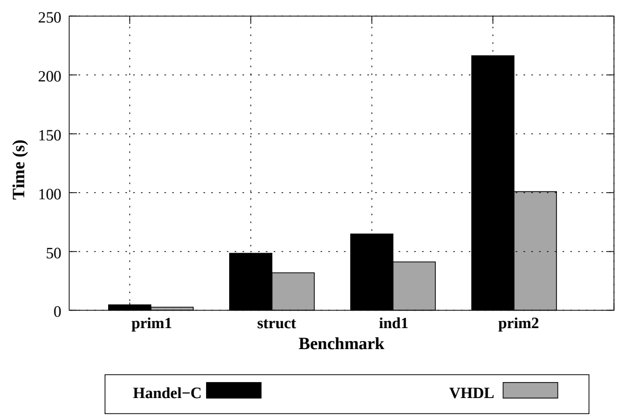


Figure 4.28: Handel-C vs VHDL Timing

in resources and processing time could be achieved. The VHDL improvement in execution time is contributed to the lack of use of semaphores and operating at its full clock potential. The speed of development and simplicity of debugging are the only advantages found in using the Handel-C language.

4.5 Simulation/Verification of Local Search Architecture

The aim of the hardware Local Search implementation is to develop a design that generates similar results to that obtained by the software implementation while improving execution time. The following will discuss problems found with the initial Handel-C architecture and a few modifications made to rectify these issues to achieve further improvement in performance.

4.5.1 Performance Analysis & Tuning

In designing and testing the Local Search Hardware processor, the main problem encountered with the original design was lack of execution performance. From Table 4.7, it is evident that the initial hardware implementation produced slower execution times than the software using the same bit representation. This may be a result of the sequential nature of the Local Search (ie. many operations within the algorithm depend on data generated on previous steps, limiting the algorithm from any parallelism). Therefore, in designing the original Local Search algorithm, little pipelining and parallelization could be achieved.

Benchmark	Software	Original Handel-C Design
Maximum Clock	N\A	87 MHz
Equivalent Gates	N\A	48,073
pcb1.dat	0.0 s	0.0 s
frac.dat	0.020 s	0.031 s
chip4.dat	0.063 s	0.122 s
chip1.dat	0.123 s	0.247 s
prim1.dat	3.2 s	4.7 s
ind1.dat	53.8 s	64.9 s
struct.dat	29.6 s	48.5 s
prim2.dat	126.7 s	216.4 s

Average time over 5 trials
Software run on Linux OS, HP Workstation x2100 P4 2.4 GHz, 1 Gig memory

Table 4.7: Local Search Software vs Hardware

As previously discussed in section 4.2 there were three functions of the software which produced the majority of the processing time. These functions consisted of counting the number of cells in $Partition_1$, copying the Partition and Cell Data to make working copies and performing the Partition Information Update process.

The largest bottleneck of the software is the counting of cells in $Partition_1$. In the hardware implementation this function required no overhead since all bits are counted in a single cycle and it is executed in parallel with the copying of Partition and Cell Data. This leaves only two main contributors to the bottleneck of the hardware. Using Amdahl's law, increasing the performance in these two areas will have the greatest effect on the architectures performance.

In analyzing the hardware modules a few issues were found that could have affected the performance:

1. The organization of the Netlist plays a role in performance limitation. When storing the Netlist information into memory, it is common for cells connected to a single net to appear in series (ie. $Cell_{10}$, $Cell_{11}$ and $Cell_{12}$ connected to a single net). This causes a problem when trying to utilize parallelism in the Partition Information Update process. The aim of this process is that the three loops can operate in parallel, allowing each to process different information with the results being passed between the loops. The drawback of using the current Netlist representation is that modules appearing in series cause an idle state to occur, waiting for the following loops to complete their tasks before sending any new information. This interrupts the parallel execution of the loop processes, and causes the system to operate in a more sequential manner, as illustrated in Figure 4.29. Figure 4.29(a) shows how the system idles when modules are located close to each other. Figure 4.29(b) shows the system when modules are separated from each other: the overall performance is slightly improved.

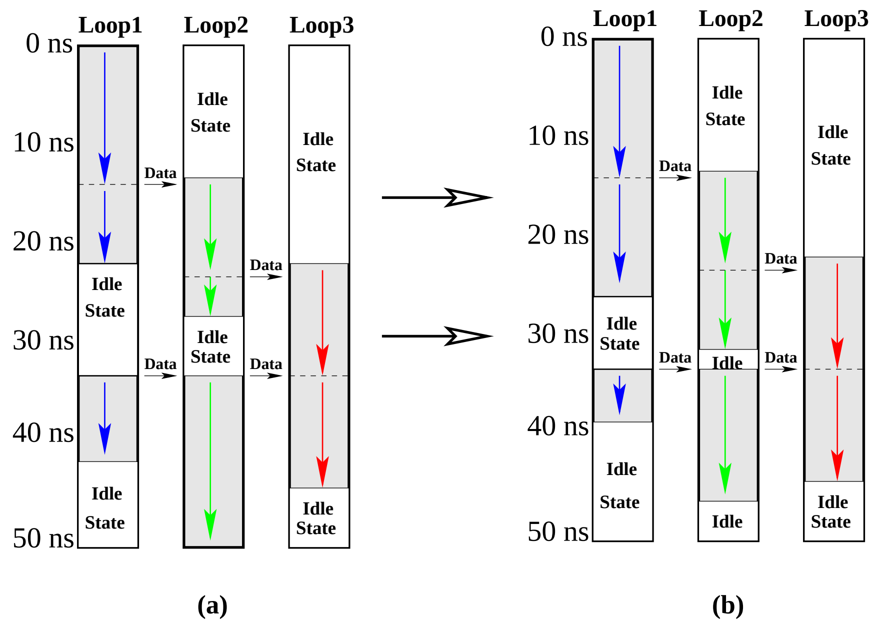


Figure 4.29: Local Search Update Timing

2. Similar to the fitness calculation in the Genetic Algorithm, section 3.4.1, the method by which Local Search calculates the status of the nets plays a huge role in determining performance. This method, shown in Figure 4.30[Sitk95, Gurw03], spends unneeded processing time searching empty bytes of data.

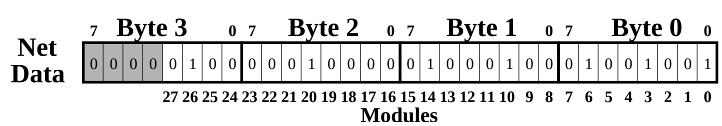


Figure 4.30: Bit Net Representation

3. In executing "Data-Replicator-Module" (CopyData) in software the module consumes around 6% of the execution time. In the hardware implementation, the limiting factor of this module is memory access. Since all Partition and

Solution Data are stored in the same off-chip memory and only one memory access is allowed at a time, a bottleneck occurs.

4.5.2 Design Modifications of Local Search Architecture

In analyzing the disadvantages mentioned above, numerous modifications were implemented to further enhance system performance:

Improvement #1 - In analyzing the "Partition-Data-Update" the same findings were found as in the original FC-SM: that is, the architecture is continuously searching empty bytes of data within the Netlist. To resolve this problem, the design was adapted to incorporate the Genetic Algorithm Fitness methodology presented in section 3.4.2. This method of searching the Netlist resolves the problem of searching empty bytes of data while making the Local Search compatible with the Genetic Algorithm. In addition, it resolves the cell order problem within the Netlist. Searching integer values having sequential cells (ie. $Cell_3$, $Cell_4$, etc) within the Netlist will not have any more drastic effect on performance than using any other order of cells.

Improvement #2 - As stated in section 4.5.1, the Local Search algorithm utilizes data that is dependent on other modules, limiting the ability of parallelization. In order to increase the level of parallelization within the process, block Rams have been used to hold the Solution and Partition Data. Block Rams allow the system to operate in a more pipeline manner allowing for multiple address and data buses, meaning that modules can operate on numerous block Rams at the same time. This causes three improvements within the DR-M

1. It allows all partition and solution data to be copied in parallel, eliminating waiting for the bus access.

2. It allows the creation of backup copies of the original data while the Partition Data Update process is operating on other block Rams.

3. It allows for multiple instances of the Partition Data Update process to occur in parallel, as shown in Figure 4.31, in an attempt to further increase throughput.

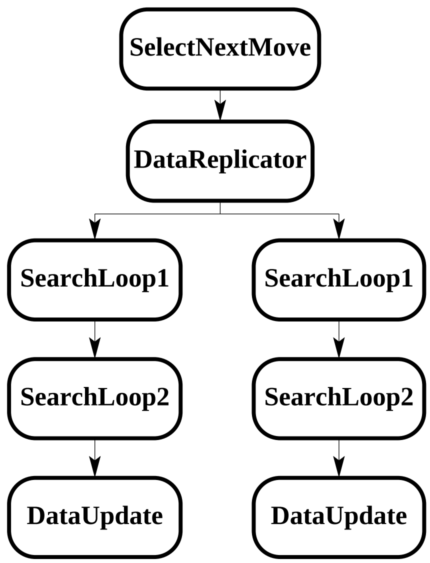


Figure 4.31: Parallel Partition-Data-Update

4.6 Computational Results of the Local Search Architecture

As discussed in the previous section numerous design errors were corrected and implemented to improve the execution speed of the algorithm. All these proposed designs were developed using Celoxica DK Suite 2.0 and compiled using Xilinx ISE 6.1.03i. They were implemented on the Celoxica RC1000 development board using a Virtex-E FPGA with 2 million gates. Results of these design improvements can be found in Table 4.8 and Figure 4.32.

Benchmark	Software	Original Design	Improvement #1	Improvement #2 (1,2)	Improvement #2 (3)
Maximum Clk	N\A	87 MHz	85 MHz	89 MHz	89 MHz
Equivalent Gates	N\A	48,073	49,212	657,726 (36 BlkRam)	673,263 (36 BlkRam)
pcb1.dat	0.00 s	0.0 s	0.0 s	0.0 s	0.0 s
frac.dat	0.020 s	0.031 s	0.028 s	0.016 s	0.012 s
chip4.dat	0.063 s	0.122 s	0.081 s	0.034 s	0.031 s
chip1.dat	0.123 s	0.247 s	0.150 s	0.056 s	0.056 s
prim1.dat	3.2 s	4.7 s	3.4 s	1.5 s	1.5 s
struct.dat	29.6 s	48.5 s	27.8 s	12.0 s	12.0 s
ind1.dat	53.8 s	64.9 s	46.2 s	20.7 s	20.9 s
prim2.dat	126.7 s	216.4 s	113.9 s	50.9 s	49.2 s

Average time over 5 trials

Software run on Linux OS, HP Workstation x2100 P4 2.4 GHz, 1 Gig memory

Table 4.8: Local Search Technique Comparison

In examining the data from different implementations it was found that the Local Search improvement #2 (1,2) with a single "Partition Data Update" process performed equally to that utilizing two "Partition Data Update" processes. Due to the fact that a new method for storing the Netlist is utilized in the new architecture, the bottleneck of the architecture shifted towards the copying of partition and

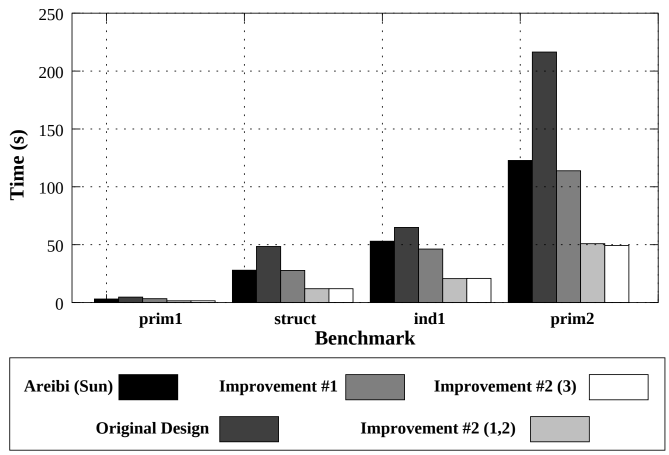


Figure 4.32: Timing Comparison of Local Search Modifications

solution data as evident in Table 4.9. Therefore, there is no need to implement more than one instance of the update Process unless the copying of the data is improved.

Name of Software Function	Equivalent Functionality	% Execution Time			
		struct	prim2	prim1	struct
Count	Count '1' for feasibility	89.81	77.37	89.94	100.00
CopyData	Make Working Copies	6.63	21.13	5.03	0.00
Loop3	Update Partition Data (Loop 3)	1.47	1.00	1.12	0.00
Loop2	Update Partition Data (Loop 2)	1.23	0.26	1.12	0.00
Loop1	Update Partition Data (Loop 1)	0.12	0.05	0.00	0.00

Table 4.9: New Local Search Software Profile

It is important to note that the balancing criteria is the only user defined parameter that can have an effect on solution quality. As stated in section 3.5.5 this parameter is not a tuning parameter but is a constraint on the system. Figure 4.33 shows the average result of a five trial run of the Local Search for each benchmark while varying the allowable balancing difference. Numerical data describing the

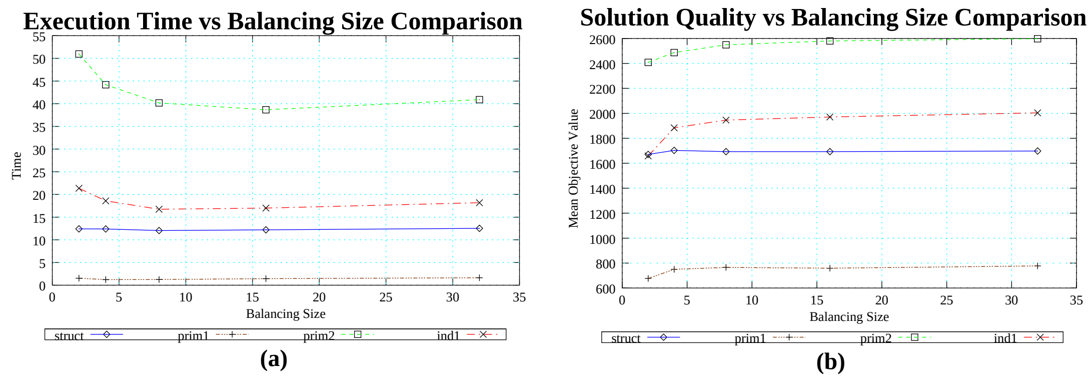


Figure 4.33: Effect of Balancing Size on Local Search design

mean (μ), the best result, the worst result, and the standard deviation (σ) for each benchmark can be found in Appendix D.1.

In examining Figure 4.33(a), it is noticed that as the balancing size increased the effect on the execution time decreased. This finding is linked to the increase in objective value found in Figure 4.33(b). Increasing the balancing criteria allows for moves that were infeasible with lower balancing size to become feasible resulting in a higher objective value. These feasible moves also allow the Local Search to accept moves with higher gains, forcing the solution to a local maximum with fewer neighbourhood searches.

4.7 A Memetic Algorithm Hardware Accelerator

By successfully implementing a Genetic Algorithm (described in chapter 3) and a Local Search Algorithm (described earlier) we can combine both architectures and develop a Memetic based architecture, illustrated in Figure 4.34. Memetic Algorithms, containing Genetic Algorithm's ability to search the solution space and Local Search's ability to fine tune solutions, are able to produce better results in suitable amounts of time.

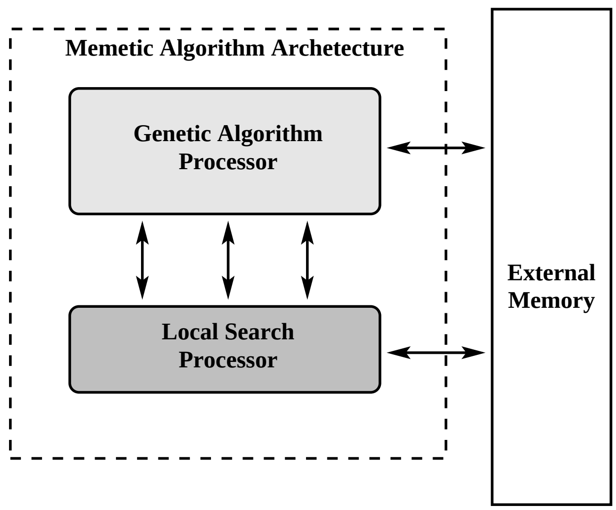


Figure 4.34: Memetic Algorithm Architecture Block Diagram

In designing the Memetic Algorithm architecture, two different hybrids were created using the final enhanced Genetic Algorithm and Local Search architectures. The first, called the Exhaustive Memetic Algorithm (EMA), uses the Local Search to improve the final solution of the Genetic Algorithm, as illustrated in 4.35. A few individuals are selected from the population in the final generation and further improved using the Local Search, as illustrated in Figure 4.36.

The second Memetic Algorithm, called the Intermediate Memetic Algorithm (IMA), is more complex than EMA. This technique applies a few iterations of the Local Search algorithm to a small number of random individuals in the population. This occurs after a predetermined number of generations of the Genetic Algorithm. The block diagram of IMA is shown in Figure 4.37. The goal of this technique is

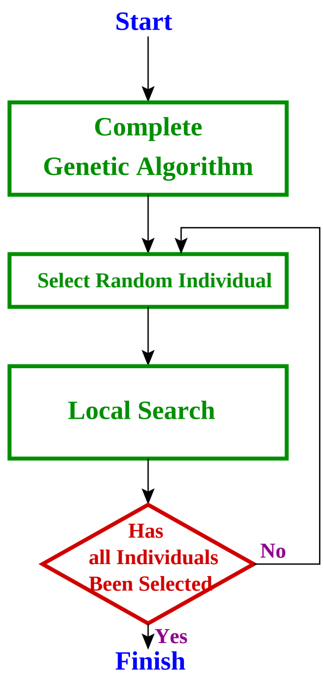


Figure 4.35: Exhaustive Memetic Algorithm Block Diagram

to steer the population of the Genetic Algorithm toward better solutions without taking the solutions to local maximums.

4.7.1 Memetic Algorithm Registers

In order to control the execution of the Memetic Algorithm and maintain the algorithm flexibility, three new user parameters are introduced. These parameters are stored in internal registers and are programmed through the memory. The definition of these registers can be found in Table 4.10.

To accurately compare the results generated by the two algorithms it is necessary to determine the effect of each parameter on the solution quality and execution time. The base case values for the Genetic Algorithm, found in Table 3.7, were used for

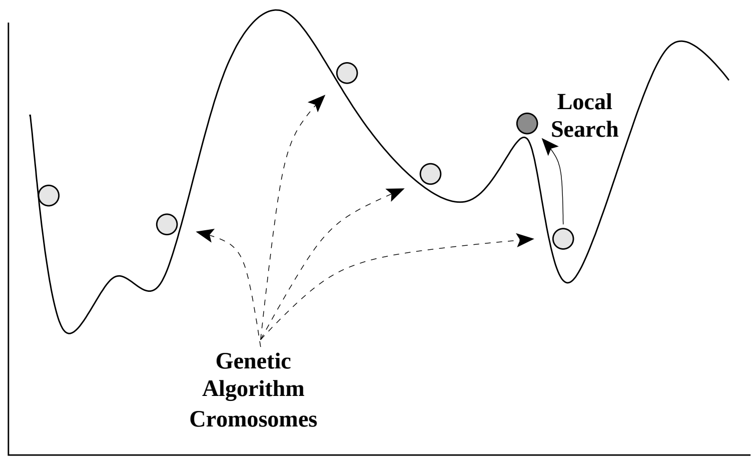


Figure 4.36: Memetic Algorithm Solution Landscape

Register Name	Register Size	Description
Generation Number	16 bits	The number of generations of the Genetic Algorithm to execute before applying the Local Search
Iteration Number	16 bits	The number of Local Search neighbourhood searches to execute
Individual Number	16 bits	The number of random individuals to apply Local Search algorithm to

Table 4.10: Memetic Algorithm Registers

tuning the three new Memetic Algorithms parameters. The base case values for the new parameters can be found in Table 4.11.

Parameter	Default Values
Generations GA per Local Search	10
Random Individuals Selected	8
Iterations of Local Search per Individual	9

Table 4.11: Base Case parameters for Handel-C Memetic Algorithm

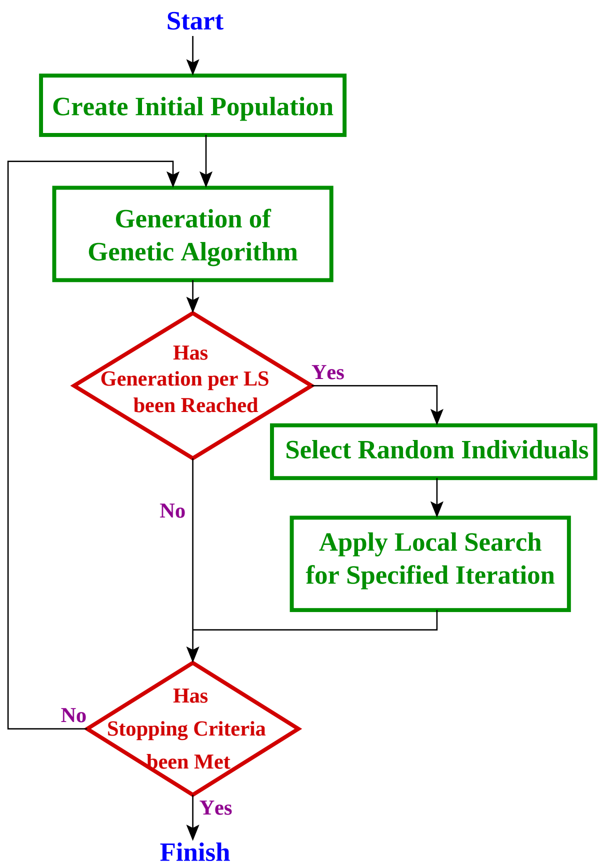


Figure 4.37: Intermediate Memetic Algorithm (IMA)

Effect of the tuning parameters on Exhaustive Memetic Algorithm

For tuning the Exhaustive Memetic Algorithm there is only one parameter that changes the resulting output. Numerical data from the tuning process can be found in Appendix D.2. From Figures 4.38 and it is evident that applying the Local Search algorithm to an increasing number of random individuals has an effect on both the solution quality and execution time of the algorithm. The figures demonstrates that as the number of random selected individuals increases there is a linear increase in execution time and a slight increase in the solution quality. This means that, for

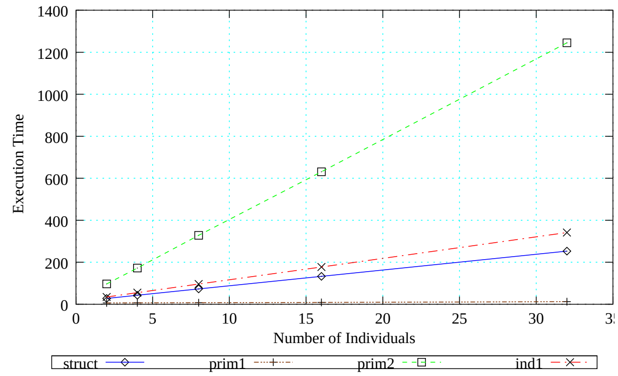


Figure 4.38: Effect of Number of Random Individuals on Time (EMA)

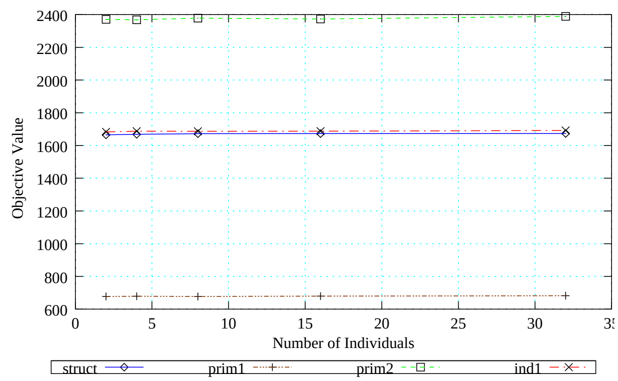


Figure 4.39: Effect of Number of Random Individuals on Best Objective Value (EMA)

the Exhaustive Memetic Algorithm, it is impractical to select a large number of individuals to apply to the Local Search. As a result, two random individuals will be selected for the final comparison.

Effect of the tuning parameters on Intermediate Memetic Algorithm (IMA)

In tuning the intermediate Memetic Algorithm, one parameter is altered while the remaining two parameters are set to their base case values. Numerical data for the IMA tuning process can be found in Appendix D.3, D.4 and D.5.

In examining Figures 4.40 and 4.41 it is clear that increasing the number of random individuals selected from the Genetic Algorithm population has a large linear effect on the execution time of the system with little increase in solution quality.

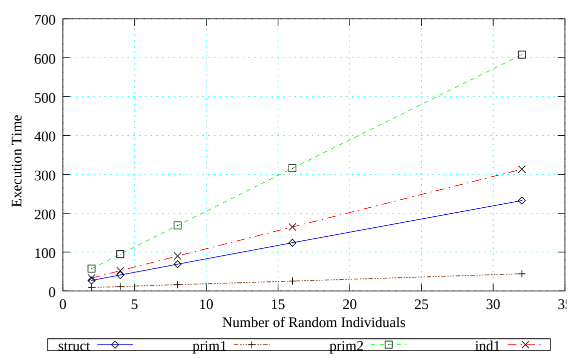


Figure 4.40: Effect of Number of Random Individuals on Time (IMA)

Increasing the number of iterations for each Local Search also has a linear increasing effect on the execution time as seen in Figure 4.42. It is also clear from Figure 4.43 that increasing the number of Local Search iterations has a larger positive impact on solution quality.

In examining the effect on the number of Genetic Algorithm generations between each Local Search, it was found that there was a negative, non-linear effect on both the execution time and the solution quality, as illustrated in Figures 4.44 and 4.45

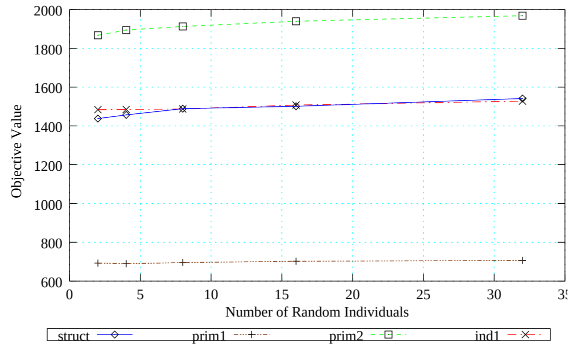


Figure 4.41: Effect of Number of Random Individuals on Best Objective Value (IMA)

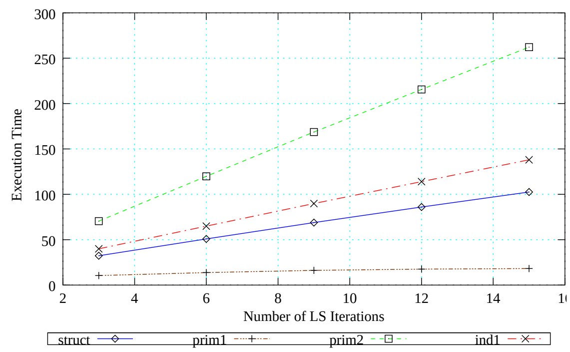


Figure 4.42: Effect of Number of Local Search Iterations on Time (IMA)

respectively. This could be expected since a fewer number of Local Search iterations will be performed on the population as the number of generations increase.

Tuning the parameters for the Intermediate Memetic Algorithm revealed that the base parameters generated an acceptable balance of good quality solutions and execution time.

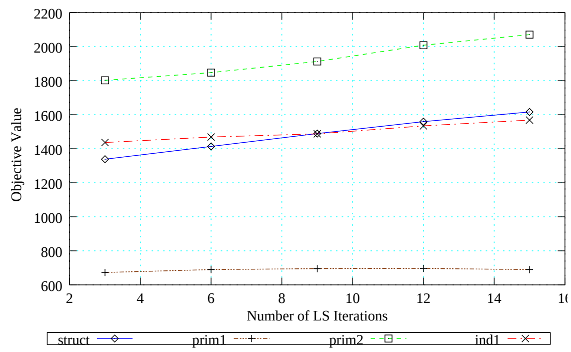


Figure 4.43: Effect of Number of Local Search Iterations on Best Objective Value (IMA)

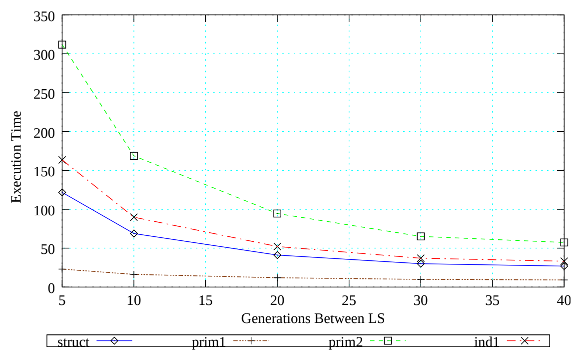


Figure 4.44: Effect of Number of GA Generations on Time (IMA)

4.7.2 Computational Results

In combining the Genetic Algorithm and the Local Search, the goal was to enhance solutions produced by the Genetic Algorithm. Figure 4.46 illustrates the final solution and execution speed of software Genetic Algorithm[Arei01] and the four different hardware designs implemented. From Figure 4.46(b), it can be seen that the Genetic Algorithm created by [Arei01] produced better results than any of the hardware implementations. As mentioned in section 3.6, the hypothetical reason for these findings is the result of using a different RNG and different crossover tech-

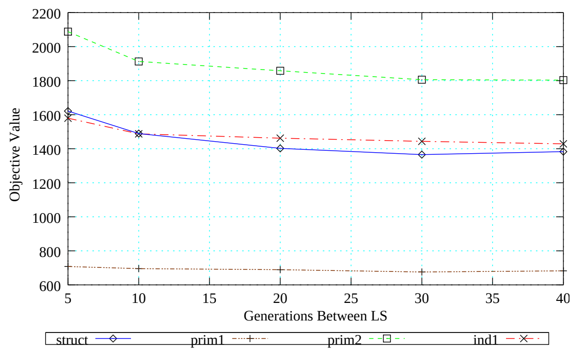


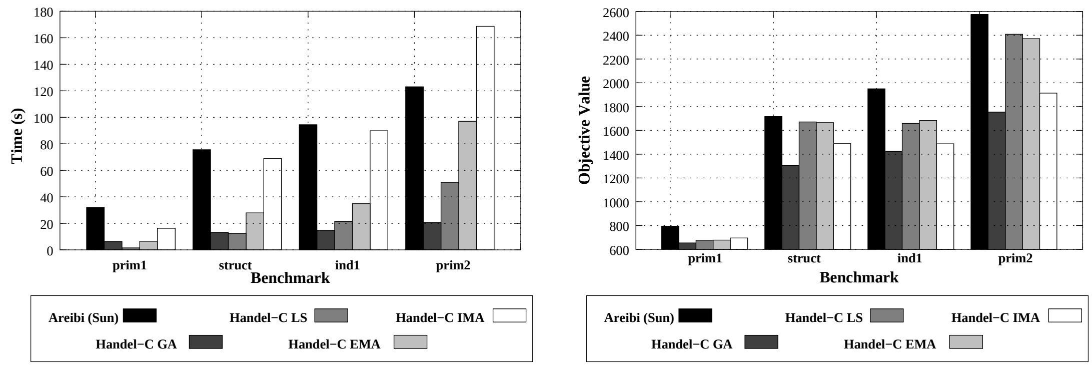
Figure 4.45: Effect of Number of GA Generations on Best Objective Value (IMA)

niques in addition to the repair mechanism. It can also be noticed from the figure that the Local Search implementation produced nearly the same solution quality as the Exhaustive Memetic Algorithm and much better results than the Intermediate Memetic Algorithm. As shown in Figure 4.46(a) the execution time of the Local Search algorithm is significantly less than that of either Memetic Algorithm.

These performance findings can be accredited to the poor solutions produced by the hardware Genetic Algorithm. An enhanced repair mechanism within the Genetic Algorithm is expected to produce better solution quality, comparable to those obtained by software.

4.8 Limitation of Hardware Implementation

In comparing the results of the hardware designs, it is found that the hardware implementations executed slower than expected. The final Genetic Algorithm executed at speeds nearly $\frac{1}{2}$ that of a software implementation using the same bit representation. In comparing the software and hardware Local Search implemen-



Software vs Hardware Time Comparison
(a)

Software vs Hardware Results Comparison
(b)

Figure 4.46: Final Performance results of Algorithms

tations, the speedup of hardware over the software is the result of the balancing function (counting the number of bits within a unsigned number) which contributed to over 80% of the software execution time. Excluding this function from the software implementation would enable it to greatly outperform the hardware based approach. Reasons for the lack of performance achieved by the hardware based architectures include the following:

1. In designing the hardware algorithms, one of the constraints placed on the system was the requirement to handle any size of benchmarks. In order to comply with this stringent constraint external memory was used to store the benchmark data. In implementing the design to use off-chip memory, two factors contributed to speed limitation:

   (a) The Handel-C language forces the system to operate at $\frac{1}{4}$ the maximum clock rate. This is to handle the signaling required to communicate with the off-chip memory while still executing each command in one clock cycle. Accordingly, the Local Search algorithm with an external clock of 89 MHz would only operate at 22.25 MHz.

(b) Long routing is required to interface the memory I/O pins to the algorithm. When developing large designs that require extensive memory accessing, the length of the address and data lines are often extremely long resulting in large delays in memory communications. This limits the maximum clock rate of the system and forces the system to operate at slower speeds.

2. Developing an algorithm in a high-level language (such as Handel-C) is expected to result in inefficient designs. On the other hand, hardware development based on low-level languages (such as VHDL) allow for greater flexibility in the design and allows the design to be directly targeted to the hardware leading to a more efficient design.

3. The size of the design implemented places a limitation on the speed of the system designed. The expectation of many is that, as FPGAs become larger and faster, larger designs can be implemented more easily within the devices. As designs become larger, the path between the logic becomes increasingly longer placing a larger connection delay on these wires and accordingly a direct effect on the operating clock frequency.

4. The results may also reflect the technology of current computers. Advances in computer technology may make it more difficult for larger FPGA designs to outperform software designs. In past literature [Zhon98a, Hauc98, Comp99,

Wrig03, Gurw03], it has been stated that the FPGAs have created significant speedups over software implementations, ranging from 10 to 100 times faster. This may not be the case with today's technology, since many of these implementations were developed when the delay of transistors was the dominant factor on hardware designs. Computers now are created such that the length of the interconnect is optimized, allowing for frequencies greater than 3 GHz while FPGAs are still operating at frequencies around 100 MHz, ie. $\frac{1}{30}$ the speed of current computers. In the Genetic Algorithm, a clock rate of $\frac{65\ MHz}{4}$ was obtainable while in development of the the Local Search design the maximum clock frequency was $\frac{89\ MHz}{4}$, 1/100th the speed of current general purpose computers.

4.9 Summary

In this chapter, an initial design of Local Search algorithm for Circuit Partitioning was presented. From this design two different architectures were developed, using Handel-C and VHDL, to compare performance tradeoffs. In comparing the two development languages it was found that the VHDL implementation outperformed the Handel-C based approach by a factor of two, while operating at half the frequency. It was also found that in comparing the development time required by the two architectures, Handel-C required roughly one fifth of the time required by VHDL.

Upon completing the Handel-C architecture, further examination revealed numerous bottlenecks at which time modifications to the design were made. Once the design was finalized, it was compiled and implemented on a Celoxica RC1000 development platform for performance testing. In comparing the final Hardware design with a similar software implementation, the hardware design operating at 89MHz achieved a speedup of nearly 2.5 times that of the software implementation executing on a Intel P4 2.4 GHz workstation.

Upon completing the Local Search algorithm, two Memetic algorithms were developed to further improve solution quality. In comparing the results generated by software and hardware implementations it was found that software produced better results than any of the four hardware architectures. This is attributed to several factors such as: (i) the efficient software repair algorithm implementation, (ii) robust one point crossover operator utilized in software versus the uniform crossover used in the hardware implementation. It was also found that the Memetic Algorithms required more execution time to generate similar results to those obtained by the Local Search.

Chapter 5

Conclusions and Future Directions

Computer Aided Design (CAD) tools play an important role in the VLSI physical design process. With advances in today's technology, the physical design process is becoming increasingly complex allowing for more transistors to be integrated onto a single die. This creates increasing pressure for efficient CAD tools. Although new techniques are continuously being investigated, the common goal of each algorithm is to produce better results in less time. Traditionally, these algorithms have been created in software due to its flexibility and ease of development. One of the drawbacks of software based programs, however, is that they execute in a sequential manner resulting in inefficient time usage. This has lead to the investigation of hardware algorithms to replace traditional software tools.

This thesis investigated the feasibility of designing FPGA-based CAD tools in attempt to outperform software implementations. These CAD tools included a Genetic Algorithm, a Local Search and Memetic Algorithms with each focusing on the VLSI circuit partitioning problem. This thesis also investigated the tradeoffs of using a high-level hardware development language "Handle-C" over conventional low-level development languages.

5.1 Hardware CAD algorithms

In developing CAD algorithms for a FPGA-based platform, it was found that neither the quality of the solutions nor the execution time was comparable to the software implementations. In analyzing the quality of the solution for the Genetic Algorithm the software implementation[Arei01] produced on average 13% better results than the hardware based algorithm while executing nearly 5 times slower. When comparing a different software algorithm that utilizes the same bit representation and produces similar results, it was found that the software outperformed the hardware at nearly twice the speed. For the Local Search implementation it was found that the hardware implementation produced nearly 2.5 times faster results than the software implementation using the same bitwise representation. The final Memetic Algorithm, incorporating the Local Search and the Genetic Algorithm, executed in a fraction of the time of the software Genetic Algorithm[Arei01] but failed to produce similar results.

These findings were attributed to the limitations of using off-chip memory and the Handel-C programming language as mentioned in the thesis.

5.2 Hardware Development Languages

In comparing the two different development languages, it was found that Celoxica's claim of Handel-C allowing software engineers to develop hardware without learning lower-level languages was valid. It was found that for designers who are inexperienced with Hardware Descriptive Languages (HDL), Handel-C provides a quick and simplified method for developing hardware architectures. The disadvantage of the language is that Handel-C is only beneficial to new developers. In comparing the two architectures developed, the Handel-C design used more resources on the FPGA, required more time to execute and was significantly less flexible than the VHDL implementation. The only benefit of the design was that it required nearly $\frac{1}{5}$ of the development time and lines of code required by the VHDL approach.

5.3 Future Work

There are several extensions and improvements that can be expansions from the current work.

1. Investigate further differences that exist between the [Arei01] software CAD based approach and the hardware architecture. The crossover technique implemented within the hardware can be easily modified to incorporate a simple one or two-point crossover. The only modification needed to the architecture is that instead of randomly generating a uniform bit mask it is possible to generate a similar mask that would select different portions of the two parent chromosomes, as shown in Figure 5.1.

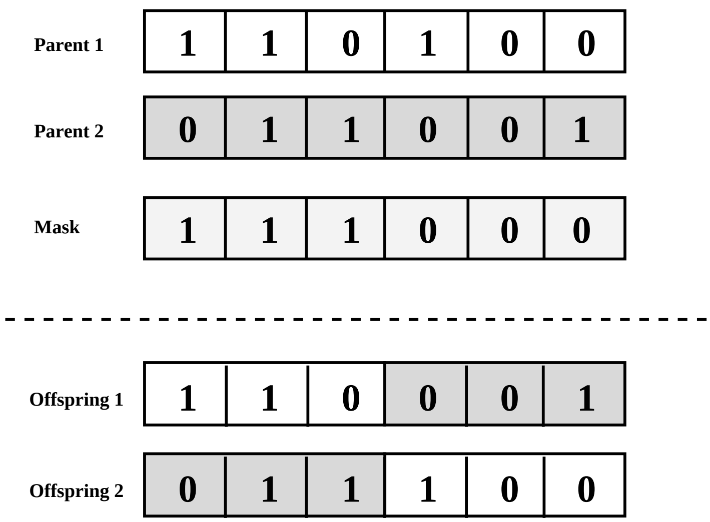


Figure 5.1: One Point Crossover Bit-mask

2. Investigate the improvement benefits of optimizing the hardware Genetic Algorithm using a low-level language. As shown in Table 4.28, designing the architecture in VHDL produced significant improvements in execution time over Handel-C. It is hypothesized that implementing the Genetic Algorithm using a low-level language would better optimize the architecture resulting in faster solutions.

3. Improve the Genetic Algorithm to include parallel Fitness Calculation. It is possible to implement a parallel fitness calculation which utilizes a single bus communication. As shown in Figure 5.2 if the fitness functions are synchronized together then it is possible to read the same Netlist values from a common bus. This would allow for nearly twice the throughput without the need for semaphores.

4. Use more off-chip memory dedicated to the Netlist which would allow for more

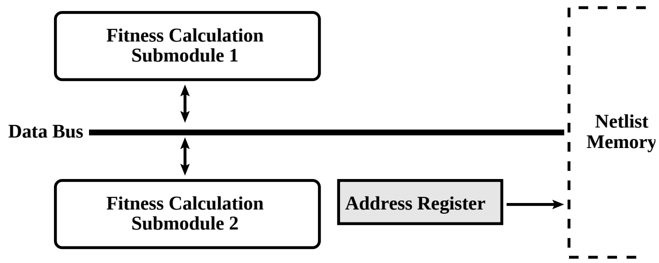


Figure 5.2: Parallel Fitness Calculation

stages of the pipeline to be dedicated to the fitness calculation. As shown in Figure 5.3 implementing the architecture using three off-chip memory banks

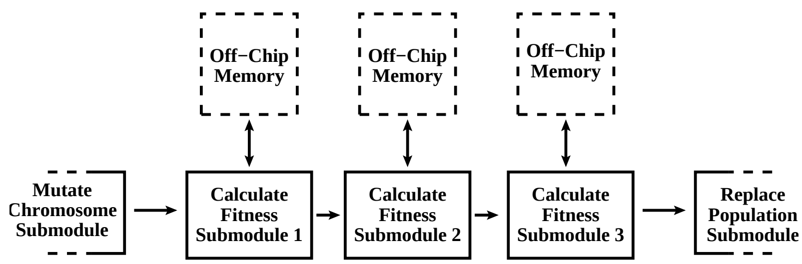


Figure 5.3: Pipelined Fitness Calculation

dedicated to the Netlist allow $\frac{1}{3}$ of the fitness value to be calculated in each fitness stage of the pipeline, resulting in higher throughput.

5. Improve the Local Search by implementing memory to eliminate repetitive searching. Using memory to store the improvement values of each neighbourhood move would allow the architecture to determine the best move by searching through the memory for the highest gain. This method of searching the solution space would significantly increase the algorithm's speed but would still require updating the memory values after each Local Search move.

6. Extend this work to solve harder problems such as VLSI placement and routing.

Appendix A

Genetic Algorithm Module Pin Descriptions

Table A.1: Signal Description of IPS

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
PopInitEnb	1 bit	Input	Start the Make Population process
InitDone	1 bit	Output	Notify the system that the population has been created
Population Memory Access Signals			
PopDATA	DataWidth	In/Out	Population memory read/write data bus
PopADDR	AddrWidth	Output	Population memory address bus
PopWE	1 bit	Output	Population memory write enable
PopEN	1 bit	Output	Population memory enable
PopGNT	1 bit	Input	Population memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each chromosome
PopSize	16 bits	Input	Number of chromosomes in the population
Repair Channel Signals			
The Repair Channel Signals are used to send information from IPS to the RCS on the location of the chromosome in the population that is to be repaired			
RepairNum	16 bit	Output	Number of the chromosome to be repaird
RepairStop	1 bit	Output	signal to tell the RCS that all chromosomes have been created
RepairGnt	1 bit	Output	Grant access to the repair function to read to read channel data
RepairAck	1 bit	Input	Acknowledgement from the repair function that it has read the data

Table A.2: Signal Description of RCS

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
PopInitEnb	1 bit	Input	Start the Make Population process
RepairDone	1 bit	Output	Notify the system that all chromosomes have been repaired
Repair Channel Signals			
The Repair Channel Signals are used to send information from IPS/CPS to the RCS on which chromosomes should be repaired			
RepairNum	16 bit	Input	The location of the chromosome that is to be repaired
RepairStop	1 bit	Input	A signal to inform the RCS that new population has been created
RepairGnt	1 bit	Input	Gain access to read channel information
RepairAck	1 bit	Output	Acknowledgment that the Repair Channel has been read
Population Memory Access Signals			
PopDATA	DataWidth	In/Out	Population Memory read/write data bus
PopADDR	AddrWidth	Output	Population memory address bus
PopWE	1 bit	Output	Population memory write enable
PopEN	1 bit	Output	Population memory enable
PopGNT	1 bit	Input	Population memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each chromosome
DIFFERENCE	16 bits	Input	The allowable difference between the number of cells in each partition
Fitness Channel Signals			
The Fitness Channel Signals are used to send information from RCS to the FCS on the location of the chromosome in the population that needs to be evaluated			
FitnessNum	16 bit	Output	The number of the chromosome to calculate fitness
FitnessStop	1 bit	Output	Signal to inform the FCS that all chromosomes have been calculated
FitnessGnt	1 bit	Output	Grant access to the FCS to read channel data
FitnessAck	1 bit	Input	Acknowledgement from the FCS that it has read the channel data

Table A.3: Signal Description of FCS

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
PopInitEnb	1 bit	Input	Start the Make Population process
FitnessDone	1 bit	Output	Notify the system that the FCS is finished
Fitness Channel Signals			
The Fitness Channel Signals are used to send information from RCS to the FCS on which chromosome in memory to evaluate			
FitnessNum	16 bits	Input	Number of the chromosome in memory to evaluate
StopFitness	1 bit	Input	Informs the FCS to end its processing
FitnessGnt	1 bit	Input	Grants access to read channel information
FitnessAck	1 bit	Output	Acknowledgment that the channel has been read
Population Memory Access Signals			
PopDATA	DataWidth	In/Out	Population memory read/write data bus
PopADDR	AddrWidth	Output	Population memory address bus
PopWE	1 bit	Output	Population memory write enable
PopEN	1 bit	Output	Population memory enable
PopGNT	1 bit	Input	Population memory grant
Netlist Memory Access Signals			
NetDATA	DataWidth	In/Out	Netlist memory read/write data bus
NetADDR	AddrWidth	Output	Netlist memory address bus
NetWE	1 bit	Output	Netlist memory write enable
NetEN	1 bit	Output	Netlist memory enable
NetGNT	1 bit	Input	Netlist memory grant
Fitness Memory Access Signals			
FitDATA	DataWidth	In/Out	Fitness memory read/write data bus
FitADDR	AddrWidth	Output	Fitness memory address bus
FitWE	1 bit	Output	Fitness memory write enable
FitEN	1 bit	Output	Fitness memory enable
FitGNT	1 bit	Input	Fitness memory GNT
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each cromosome
Nets	16 bits	Input	Number of Nets in the Netlist

Table A.4: Signal Description of FCS (Con't)

Pin Name	Bus Width	Direction	Description
Replace Channel Signals (Population Mating only)			
The Replace Channel Signals are used to send information from FCS to the RPS on the location at which the offspring is to be stored into			
ReplaceNum	16 bits	Output	Number of the offspring to be replaced into the population
ReplaceStop	1 bit	Output	Signal to inform RPS that all chromosomes have completed
ReplaceGnt	1 bit	Output	Grant access to the Replace function to read channel data
ReplaceAck	1 bit	Input	Acknowledgement from the Replace function that it has read the data

Table A.5: Signal Description of SPS

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
PopRepEnb	1 bit	Input	Start the Make Population process
SelectionDone	1 bit	Output	Notify the system that the SPS has completed
Fitness Memory Access Signals			
FitDATA	DataWidth	In/Out	Fitness memory read/write data bus
FitADDR	AddrWidth	Output	Fitness memory address bus
FitWE	1 bit	Output	Fitness memory write enable
FitEN	1 bit	Output	Fitness memory enable
FitGNT	1 bit	Input	Fitness memory grant
Register Data			
Used to send user defined variables into the submodule			
CVRRate	16 bits	Input	Number to hold the Crossover rate
PopSize	16 bits	Input	Number of Chromosomes in the population
Crossover Channel Signals			
The Crossover Channel Signals are used to send information from SPS to the CPS on which parents to perform the crossover on			
CrossChild1	16 bits	Output	The index of the first chromosome in the old population to be mated
CrossChild0	16 bits	Output	The index of the second chromosome in the old in the old population to be mated
CrossNum	16 bits	Output	The index of new chromosome to be created created
CrossStop	1 bit	Output	Signal to stop executing the Crossover process
CrossGnt	1 bit	Output	Grant access to the Mutation Crossover to read channel information
CrossAck	1 bit	Input	Acknowledgement from the Crossover function that it has read the channel data
Copy Parent Channel Signals			
The Copy Parent Channel Signals are used to send information from SPS to the CoPS on which parents to copy directly to the new population			
CopyChild1	16 bits	Output	The index of the first chromosome in the old population to be copied into New Population
CopyChild0	16 bits	Output	The index of the second chromosome in the old population to be copied into new population
CopyNum	16 bits	Output	Location of new population to store chromosomes
CopyStop	1 bit	Output	Inform CoPS to stop execution
CopyGnt	1 bit	Output	Grant access to read channel information
CopyAck	1 bit	Input	Acknowledgement that channel has been read

Table A.6: Signal Description of CPS

Signal Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clk
GlobalRst	1 bit	Input	Global system reset
PopRepEnb	1 bit	Input	Start the Repair Chromosome process
CrossDone	1 bit	Output	Notify the system that the crossover process is complete
Crossover Channel Signals			
The Crossover Channel Signals are used to send information from SPS to the CPS on who the parents are and where the children are to be stored in memory			
CrossChild1	16 bits	Input	Sends the location of the first parent in the current population
CrossChild0	16 bits	Input	Sends the location of the second parent in the current population
CrossNum	16 bits	Input	Informs the submodule to which offsprings are being created (Location in new population)
CrossStop	1 bit	Input	Signal to inform the CPS that the new population has been created
CrossGnt	1 bit	Input	Provides access to the channel information
CrossAck	1 bit	Output	Acknowledgment that the channel information has been read
Population Memory Access Signals			
PopDATA	DataWidth	In/Out	Population Memory read/write data bus
PopADDR	AddrWidth	Output	Population memory address bus
PopWE	1 bit	Output	Population memory write enable
PopEN	1 bit	Output	Population memory enable
PopGNT	1 bit	Input	Population memory grant
New Population Memory Access Signals			
NewPopDATA	DataWidth	In/Out	New Population Memory read/write data bus
NewPopADDR	AddrWidth	Output	New Population memory address bus
NewPopWE	1 bit	Output	New Population memory write enable
NewPopEN	1 bit	Output	New Population memory enable
NewPopGNT	1 bit	Input	New Population memory grant

Table A.7: Signal Description of CPS (con't)

Pin Name	Bus Width	Direction	Description
New Fitness Memory Access Signals			
NewFitDATA	DataWidth	In/Out	New Fitness Memory read/write data bus
NewFitADDR	AddrWidth	Output	New Fitness memory address bus
NewFitWE	1 bit	Output	New Fitness memory write enable
NewFitEN	1 bit	Output	New Fitness memory enable
NewFitGNT	1 bit	Input	New Fitness memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each cromosome
PopSize	16 bits	Input	Number of chromosomes in the population population
Mutation Channel Signals			
The Mutation Channel Signals are used to send information from CPS to the MCS on the location of the child chromosome that is to be mutated			
MutationNum	16 bits	Output	The number in the new population of the child that is to be mutated
MutationStop	1 bit	Output	signal to tell the MCS that all chromosomes have been mutated
MutationGnt	1 bit	Output	Grant access to Mutation Channel
MutationAck	1 bit	Input	Acknowledgement from the MCS that it has read the data

Table A.8: Signal Description of MCS

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
Clk	1 bit	Input	System clk
GlobalRst	1 bit	Input	Global system reset
PopRepEnb	1 bit	Input	Start the Repair Chromosome process
MutationDone	1 bit	Output	Notify the system that the mutation process is complete
Mutation Channel Signals			
The Mutation Channel Signals are used to send information from CPS to the MCS on which chromosomes within the new population are to be mutated			
MutationNum	1 bit	Input	Which chromosome in the population to mutate
MutationStop	16 bit	Input	Inform the MCS to stop execution
MutationGnt	1 bit	Input	Gain access to read the channel information
MutationAck	1 bit	Output	Acknowledgment that channels have been read
New Population Memory Access Signals			
NewPopDATA	DataWidth	In/Out	New Population Memory read/write data bus
NewPopADDR	AddrWidth	Output	New Population memory address bus
NewPopWE	1 bit	Output	New Population memory write enable
NewPopEN	1 bit	Output	New Population memory enable
NewPopGNT	1 bit	Input	New Population memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each chromosome
MUTERate	16 bits	Input	Number to hold Mutation Rate (out of 65,535)
Repair Channel Signals			
The Repair Channel Signals are used to send information from MCS to the RCS on which chromosomes within the new population are to be repaired			
RepairNum	16 bit	Output	Sends the location of the chromosome to be repaired
RepairStop	1 bit	Output	Signal to stop the RCS
RepairGnt	1 bit	Output	Grant access to read channel data
RepairAck	1 bit	Input	Acknowledgement that the channel has been read

Table A.9: Signal Description of RPS

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clk
GlobalRst	1 bit	Input	Global system reset
PopRepEnb	1 bit	Input	Start the Replace Chromosome process
ReplaceDone	1 bit	Output	Notify the system that all children have been replaced into the new population
Replace Channel Signals			
The Replace Channel Signals are used to send information from RCS to the RPS on which offsprings should be stored in the new population			
ReplaceNum	16 bit	Input	Index of the offspring in memory
ReplaceStop	1 bit	Input	Signal to stop the Replacement process
ReplaceGnt	1 bit	Input	Grant access to read the channel information
ReplaceAck	1 bit	Output	Acknowledgement signal that the channel information has been read
Population Memory Access Signals			
PopDATA	DataWidth	In/Out	Population Memory read/write data bus
PopADDR	AddrWidth	Output	Population memory address bus
PopWE	1 bit	Output	Population memory write enable
PopEN	1 bit	Output	Population memory enable
PopGNT	1 bit	Input	Population memory grant
Fitness Memory Access Signals			
FitDATA	DataWidth	In/Out	Fitness Memory read/write data bus
FitADDR	AddrWidth	Output	Fitness memory address bus
FitWE	1 bit	Output	Fitness memory write enable
FitEN	1 bit	Output	Fitness memory enable
FitGNT	1 bit	Input	Fitness memory grant
New Population Memory Access Signals			
NewPopDATA	DataWidth	In/Out	New Population Memory read/write data bus
NewPopADDR	AddrWidth	Output	New Population memory address bus
NewPopEN	1 bit	Output	New Population memory enable
NewPopGNT	1 bit	Input	New Population memory grant
New Fitness Memory Access Signals			
NewFitDATA	DataWidth	In/Out	New Fitness Memory read/write data bus
NewFitADDR	AddrWidth	Output	New Fitness memory address bus
NewFitWE	1 bit	Output	New Fitness memory write enable
NewFitEN	1 bit	Output	New Fitness memory enable
NewFitGNT	1 bit	Input	New Fitness memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each chromosome
PopSize	16 bits	Input	Size of the population

Table A.10: Signal Description of CoPS

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
PopRepoEnb	1 bit	Input	Start the CoPS process
CopyDone	1 bit	Output	Notify the system that submodule finished
Copy Parent Channel Signals			
The Copy Parent Channel Signals are used to send information from SPS to the CoPS on which parents to copy directly to the new population			
CopyChild1	16 bit	Input	Index of the first parent in memory
CopyChild0	16 bit	Input	Index of the second parent in memory
CopyNum	16 bit	Input	Location in memory to store the parents
CopyStop	1 bit	Input	Signal to stop the CoPS process
CopyGnt	1 bit	Input	Gain access to read channel information
CopyAck	1 bit	Output	Acknowledgment that the channel has been read
Population Memory Access Signals			
PopDATA	DataWidth	In/Out	Population Memory read/write data bus
PopADDR	AddrWidth	Output	Population memory address bus
PopWE	1 bit	Output	Population memory write enable
PopEN	1 bit	Output	Population memory enable
PopGNT	1 bit	Input	Population memory grant
Fitness Memory Access Signals			
FitDATA	DataWidth	In/Out	Fitness Memory read/write data bus
FitADDR	AddrWidth	Output	Fitness memory address bus
FitWE	1 bit	Output	Fitness memory write enable
FitEN	1 bit	Output	Fitness memory enable
FitGNT	1 bit	Input	Fitness memory grant
New Population Memory Access Signals			
NewPopDATA	DataWidth	In/Out	New Population Memory read/write data bus
NewPopADDR	AddrWidth	Output	New Population memory address bus
NewPopWE	1 bit	Output	New Population memory write enable
NewPopEN	1 bit	Output	New Population memory enable
NewPopGNT	1 bit	Input	New Population memory grant
New Fitness Memory Access Signals			
NewFitDATA	DataWidth	In/Out	New Fitness Memory read/write data bus
NewFitADDR	AddrWidth	Output	New Fitness memory address bus
NewFitWE	1 bit	Output	New Fitness memory write enable
NewFitEN	1 bit	Output	New Fitness memory enable
NewFitGNT	1 bit	Input	New Fitness memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each chromosome

Appendix B

Local Search Module Pin Descriptions

Table B.1: Signal Description of PUM

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
UpdateEnb	1 bit	Input	Start the PUM
UpdateDone	1 bit	Output	Notify the system that PUM is finished
Solution Memory Access Signals			
SolDATA	DataWidth	In/Out	Solution memory read/write data bus
SolADDR	AddrWidth	Output	Solution memory address bus
SolWE	1 bit	Output	Solution memory write enable
SolEN	1 bit	Output	Solution memory enable
SolGNT	1 bit	Input	Solution memory grant
Partition 1 Memory Access Signals			
Part1DATA	DataWidth	In/Out	Partition 1 memory read/write data bus
Part1ADDR	AddrWidth	Output	Partition 1 memory address bus
Part1WE	1 bit	Output	Partition 1 memory write enable
Part1EN	1 bit	Output	Partition 1 memory enable
Part1GNT	1 bit	Input	Partition 1 memory grant
Partition 0 Memory Access Signals			
Part0DATA	DataWidth	In/Out	Partition 0 memory read/write data bus
Part0ADDR	AddrWidth	Output	Partition 0 memory address bus
Part0WE	1 bit	Output	Partition 0 memory write enable
Part0EN	1 bit	Output	Partition 0 memory enable
Part0GNT	1 bit	Input	Partition 0 memory grant
Netlist Memory Access Signals			
NetDATA	DataWidth	In/Out	Netlist memory read/write data bus
NetADDR	AddrWidth	Output	Netlist memory address bus
NetWE	1 bit	Output	Netlist memory write enable
NetEN	1 bit	Output	Netlist memory enable
NetGNT	1 bit	Input	Netlist memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each Net
Nets	16 bits	Input	Number of Nets in the Netlist

Table B.2: Signal Description of SNNMM

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
SearchEnb	1 bit	Input	Start the SNNMM
NextDone	1 bit	Output	Notify the system that the SNNMM is finished
Partition 1 Memory Access Signals			
Part1DATA	DataWidth	In/Out	Partition 1 memory read/write data bus
Part1ADDR	AddrWidth	Output	Partition 1 memory address bus
Part1WE	1 bit	Output	Partition 1 memory write enable
Part1EN	1 bit	Output	Partition 1 memory enable
Part1GNT	1 bit	Input	Partition 1 memory grant
Partition 0 Memory Access Signals			
Part0DATA	DataWidth	In/Out	Partition 0 memory read/write data bus
Part0ADDR	AddrWidth	Output	Partition 0 memory address bus
Part0WE	1 bit	Output	Partition 0 memory write enable
Part0EN	1 bit	Output	Partition 0 memory enable
Part0GNT	1 bit	Input	Partition 0 memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each Net
Nets	16 bits	Input	Number of Nets in the Netlist
Data Replicator Channel Signals			
The Partition Update Channel Signals are used to send information from SNNMM to the DRM on which net to move and into which partition to move it			
DataRepNum	16 bit	Output	Number of the net to be moved
DataRepBlk	1 bit	Output	Which block to move net into
StopDataRep	1 bit	Output	Signal to stop DRM
DataRepGnt	1 bit	Output	Grant access to read channel data
DataRepAck	1 bit	Input	Acknowledgement that channel has been read

Table B.3: Signal Description of DRM

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
SearchEnb	1 bit	Input	Start searching for best move
DataRepDone	1 bit	Output	Informs when DRM is complete
Data Replicator Channel Signals			
The Partition Update Channel Signals are used to send information from SNNMM to the DRM on which net to move and into which partition to move it			
DataRepNum	16 bit	Input	Number of the net to be moved
DataRepBlk	1 bit	Input	Which block to move net into
DataRepStop	1 bit	Input	Signal to stop DRM
DataRepGnt	1 bit	Input	Grant access to read channel data
DataRepAck	1 bit	Output	Acknowledgement that channel has been read
Solution Memory Access Signals			
SolDATA	DataWidth	In/Out	Solution memory read/write data bus
SolADDR	AddrWidth	Output	Solution memory address bus
SolWE	1 bit	Output	Solution memory write enable
SolEN	1 bit	Output	Solution memory enable
SolGNT	1 bit	Input	Solution memory grant
Partition 1 Memory Access Signals			
Part1DATA	DataWidth	In/Out	Partition 1 memory read/write data bus
Part1ADDR	AddrWidth	Output	Partition 1 memory address bus
Part1WE	1 bit	Output	Partition 1 memory write enable
Part1EN	1 bit	Output	Partition 1 memory enable
Part1GNT	1 bit	Input	Partition 1 memory Grant
Partition 0 Memory Access Signals			
Part0DATA	DataWidth	In/Out	Partition 0 memory read/write data bus
Part0ADDR	AddrWidth	Output	Partition 0 memory address bus
Part0WE	1 bit	Output	Partition 0 memory write enable
Part0EN	1 bit	Output	Partition 0 memory enable
Part0GNT	1 bit	Input	Partition 0 memory grant
Solution Copy Memory Access Signals			
SolCpyDATA	DataWidth	In/Out	Solution Copy memory read/write data bus
SolCpyADDR	AddrWidth	Output	Solution Copy memory address bus
SolCpyWE	1 bit	Output	Solution Copy memory write enable
SolCpyEN	1 bit	Output	Solution Copy memory enable
SolCpyGNT	1 bit	Input	Solution Copy memory grant

Table B.4: Signal Description of DRM (Con't)

Pin Name	Bus Width	Direction	Description
Partition 1 Copy Memory Access Signals			
Part1CpyDATA	DataWidth	In/Out	Partition 1 Copy memory read/write data bus
Part1CpyADDR	AddrWidth	Output	Partition 1 Copy memory address bus
Part1CpyWE	1 bit	Output	Partition 1 Copy memory write enable
Part1CpyEN	1 bit	Output	Partition 1 Copy memory enable
Part1CpyGNT	1 bit	Input	Partition 1 Copy memory grant
Partition 0 Copy Memory Access Signals			
Part0CpyDATA	DataWidth	In/Out	Partition 0 Copy memory read/write data bus
Part0CpyADDR	AddrWidth	Output	Partition 0 Copy memory address bus
Part0CpyWE	1 bit	Output	Partition 0 Copy memory write enable
Part0CpyEN	1 bit	Output	Partition 0 Copy memory enable
Part0CpyGNT	1 bit	Input	Partition 0 Copy memory grant
Netlist Memory Access Signals			
NetDATA	DataWidth	In/Out	Netlist memory read/write data bus
NetADDR	AddrWidth	Output	Netlist memory address bus
NetWE	1 bit	Output	Netlist memory write enable
NetEN	1 bit	Output	Netlist memory enable
NetGNT	1 bit	Input	Netlist memory grant
Update Partition Channel Signals out			
This channel is used to inform the Update-Partition-Data to which net has been moved			
PartUpdateNum	16 bit	Output	Sends which net to perform the update on
PartUpdateStop	1 bit	Output	Signal to stop update process
PartUpdateBlk	1 bit	Output	Inform the Update process which block to the net into
PartUpdateGnt	1 bit	Output	Grant access to read channel data
PartUpdateAck	1 bit	Input	Acknowledgement channel data has been read
Update Partition Channel Signals in			
This channel is used to inform the ABMM when the Partition Update is finished			
PartUpdateDoneGnt	1 bit	Input	Receive access to continue
PartUpdateDoneAck	1 bit	Output	Acknowledgement of grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each Net
Nets	16 bits	Input	Number of Nets in the Netlist

Table B.5: Signal Description of SLM

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
UpdtEnb	1 bit	Input	Start the SLM
LoopDone	1 bit	Output	Notify the system that the SLM is finished
Search Loop in Channel Signals in			
The Search Loop Channel Signals are used to send information between the SLMs on which net/cell perform operations on			
LoopInStop	16 bit	Input	Signal to stop the SLM execution
LoopInNum	1 bit	Input	Number of net/cell to perform operations on
LoopInGnt	1 bit	Input	Grant access to read channel data
LoopInAck	1 bit	Output	Acknowledgement that channel has been read
Netlist/Cellist Memory Access Signals			
DataListDATA	DataWidth	In/Out	Net/Module list memory read/write data bus
DataListADDR	AddrWidth	Output	Net/Module list memory address bus
DataListWE	1 bit	Output	Net/Module list memory write enable
DataListEN	1 bit	Output	Net/Module list memory enable
DataListGNT	1 bit	Input	Net/Module list memory grant
Register Data			
Used to send user defined variables into the submodule			
Modules	16 bits	Input	Number of Modules in each Net
Nets	16 bits	Input	Number of Nets in the Netlist
Search Loop out Channel Signals out			
LoopOutStop	16 bit	Input	Signal to stop the following submodules execution
LoopOutNum	1 bit	Input	Number of net/cell to perform operations on
LoopOutGnt	1 bit	Input	Grant access to read channel data
LoopOutAck	1 bit	Output	Acknowledgement that channel has been read

Table B.6: Signal Description of DUM

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
Clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
UpdtEnb	1 bit	Input	Signal to start the DUM
DataUpdtDone	1 bit	Output	Notify the system the DUM is done
Data Update Channel Signals			
The Data Update Channel Signals are used to send information from SLM to the DUM on which nets need to be check for current cut status			
DataUpdtNum	16 bit	Input	The index of the net to be checked
DataUpdtStop	1 bit	Input	Signal to stop DUM
DataUpdtGnt	1 bit	Input	Grant access to read channel data
DataUpdtAck	1 bit	Output	Acknowledgement that channel has been read
Solution Copy Memory Access Signals			
SolCpyDATA	DataWidth	In/Out	Solution Copy memory read/write data bus
SolCpyADDR	AddrWidth	Output	Solution Copy memory address bus
SolCpyWE	1 bit	Output	Solution Copy memory write enable
SolCpyEN	1 bit	Output	Solution Copy memory enable
SolCpyGNT	1 bit	Input	Solution Copy memory grant
Partition 1 Copy Memory Access Signals			
Part1CpyDATA	DataWidth	In/Out	Partition 1 Copy memory read/write data bus
Part1CpyADDR	AddrWidth	Output	Partition 1 Copy memory address bus
Part1CpyWE	1 bit	Output	Partition 1 Copy memory write enable
Part1CpyEN	1 bit	Output	Partition 1 Copy memory enable
Part1CpyGNT	1 bit	Input	Partition 1 Copy memory grant
Partition 0 Copy Memory Access Signals			
Part0CpyDATA	DataWidth	In/Out	Partition 0 Copy memory read/write bus
Part0CpyADDR	AddrWidth	Output	Partition 0 Copy memory address bus
Part0CpyWE	1 bit	Output	Partition 0 Copy memory write enable
Part0CpyEN	1 bit	Output	Partition 0 Copy memory enable
Part0CpyGNT	1 bit	Input	Partition 0 Copy memory grant
Netlist Memory Access Signals			
NetDATA	DataWidth	In/Out	Netlist memory read/write data bus
NetADDR	AddrWidth	Output	Netlist memory address bus
NetWE	1 bit	Output	Netlist memory write enable
NetEN	1 bit	Output	Netlist memory enable
NetGNT	1 bit	Input	Netlist memory grant
Register Data			
User defined variables and internal registers values sent into the submodule			
Modules	16 bits	Input	Number of Modules in each Net
Nets	16 bits	Input	Number of Nets in the Netlist
Sum	16 bits	Output	Relative sum of uncut/cut nets

Table B.7: Signal Description of ABMM

Pin Name	Bus Width	Direction	Description
System Signals			
These are signals to control the execution of the submodule			
Clk	1 bit	Input	System clock
GlobalRst	1 bit	Input	Global system reset
BestMoveEnb	1 bit	Input	Start the ABMM
BestMoveDone	1 bit	Output	Notify the system that ABMM has finished
Solution Memory Access Signals			
SolDATA	DataWidth	In/Out	Solution memory read/write data bus
SolADDR	AddrWidth	Output	Solution memory address bus
SolWE	1 bit	Output	Solution memory write enable
SolEN	1 bit	Output	Solution memory enable
SolGNT	1 bit	Input	Solution memory grant
Netlist Memory Access Signals			
NetDATA	DataWidth	In/Out	Netlist memory read/write data bus
NetADDR	AddrWidth	Output	Netlist memory address bus
NetWE	1 bit	Output	Netlist memory write enable
NetEN	1 bit	Output	Netlist memory enable
NetGNT	1 bit	Input	Netlist memory grant
Register Data			
Used to send user defined variables and the Local Search move			
Modules	16 bits	Input	Number of Modules in each Net
Nets	16 bits	Input	Number of Nets in the Netlist
BestMoveNum	16 bits	Input	Which net will give the best gain
BestMoveBlk	1 bits	Input	Into which block to move net into
Update Partition out Channel Signals			
This channel is used to inform the Update-Partition-Data to which net has been moved			
PartUpdateNum	16 bit	Output	Sends which net to perform the update on
PartUpdateStop	1 bit	Output	Signal to stop update process
PartUpdateBlk	1 bit	Output	Inform the Update process which block to the net into
PartUpdateGnt	1 bit	Output	Grant access to read channel data
PartUpdateAck	1 bit	Input	Acknowledgement channel data has been read
Update Partition in Channel Signals			
This channel is used to inform the ABMM when the Partition Update is finished			
PartUpdateDoneGnt	1 bit	Input	Receive access to continue
PartUpdateDoneAck	1 bit	Output	Acknowledgement of grant

Appendix C

Genetic Algorithm Experimental Results

Benchmark	Number of Nets	Software Algorithm[Arei01] (Sun Blade)					Hardware Algorithm				
		Time	Best Result	Worst Result	Mean Result	σ	Time	Best Result	Worst Result	Mean Result	σ
struct.dat	1920	75.570	1716.6	1675.4	1696.559	8.093	13.156	1304.2	1275.4	1296.050	6.409
prim1.dat	902	31.670	794.6	765.0	785.827	5.898	6.231	653.8	638.8	649.628	3.289
prim2.dat	3029	123.043	2574.4	2493.6	2536.681	15.114	20.559	1753.6	1722.6	1742.900	6.927
ind1.dat	2192	94.326	1949.6	1889.0	1922.908	12.516	14.687	1423.4	1395.8	1415.859	6.131
pcb1.dat	32	0.810	25.4	19.4	24.984	1.157	0.218	26.6	25.4	26.587	0.120
chip1.dat	294	9.420	252.6	241.4	250.705	2.463	2.087	237.4	229.8	236.442	1.440
chip4.dat	221	6.573	158.4	174.8	183.286	2.2951	1.506	182.4	177.2	182.102	0.886
frac.dat	147	4.270	110.0	98.8	108.045	2.630	1.059	115.8	110.6	115.597	0.762

Crossover=99%, Mutation=0.36%, Population Size=128, Generations=200, Difference 2

Sun Blade 2000 : 900 MHz UltraSparc III Cu, 1024 MB Ram, Solaris 9

Table C.1: Hardware vs Software Comparison

Generation Number		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
20	Time	1.350	0.631	2.122	1.512	0.022	0.206	0.159	0.106
	Best	850.6	439.0	1254.6	1014.4	25.8	171.6	148.4	92.8
	Worst	802.6	406.6	1201.0	965.0	23.8	149.0	131.6	80.4
	Mean	826.753	424.170	1229.627	990.997	25.784	163.488	141.486	88.714
	SD	9.206	6.065	8.970	8.744	0.176	4.042	2.933	2.113
50	Time	3.297	1.550	5.160	3.694	0.056	0.519	0.378	0.259
	Best	1026.0	540.8	1453.6	1176.2	27.0	213.4	177.6	114.2
	Worst	989.8	513.0	1412.8	1136.0	26.0	201.4	168.0	106.0
	Mean	1010.088	527.472	1434.198	1158.144	26.992	209.448	174.411	113.013
	SD	6.860	4.726	7.686	6.672	0.088	1.876	1.635	1.203
100	Time	6.566	3.097	10.250	7.337	0.112	1.035	0.753	0.531
	Best	1187.0	608.8	1615.2	1311.6	25.4	231.4	184.2	117.2
	Worst	1157.0	588.4	1578.2	1279.2	22.8	223.0	178.6	111.2
	Mean	1176.523	603.372	1598.912	1300.228	25.363	230.319	183.656	116.983
	SD	6.644	4.063	7.363	6.188	0.292	1.400	1.015	0.847
200	Time	13.156	6.231	20.559	14.687	0.218	2.087	1.506	1.059
	Best	1304.2	653.8	1753.6	1423.4	26.6	237.4	182.4	115.8
	Worst	1275.4	638.8	1722.6	1395.8	25.4	229.8	177.2	110.6
	Mean	1296.050	649.628	1742.900	1415.859	26.587	236.442	182.102	115.597
	SD	6.409	3.289	6.927	6.131	0.120	1.440	0.886	0.762
500	Time	33.003	15.753	51.406	36.844	0.550	5.231	3.803	2.656
	Best	1381.8	691.2	1857.2	1507.8	26.0	236.6	191.2	118.0
	Worst	1355.6	676.4	1830.6	1482.0	22.8	230.6	185.0	113.2
	Mean	1375.500	688.270	1849.320	1500.948	25.959	236.012	190.709	117.798
	SD	6.885	3.724	7.455	7.022	0.334	1.254	1.077	0.726

Mutation Rate 0.36%, Crossover Rate 99%, Population Size 128, Difference 2

Average of 5 trials

Table C.2: Affect of Generation Size

Crossover Rate		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
50%	Time	6.687	3.169	10.459	7.478	0.113	1.053	0.763	0.534
	Best	1205.8	613.2	1648.8	1339.4	26.2	226.4	176.8	112.8
	Worst	1183.8	599.0	1626.4	1318.4	25.8	220.0	172.4	109.6
	Mean	1200.073	610.280	1643.878	1334.298	26.197	225.784	176.645	112.733
	SD	4.036	2.521	4.724	3.922	0.035	0.894	0.613	0.403
75%	Time	9.978	4.718	15.575	11.128	0.169	1.559	1.144	0.800
	Best	1280.0	643.0	1705.0	1387.6	25.4	225.0	185.2	115.4
	Worst	1249.4	629.0	1680.0	1363.6	24.8	218.8	179.8	111.0
	Mean	1272.081	640.011	1698.309	1380.734	25.395	224.225	184.942	115.2
	SD	5.871	3.145	5.491	5.399	0.053	1.072	0.873	0.729
90%	Time	11.972	5.672	18.619	13.353	0.203	1.881	1.372	0.959
	Best	1296.2	655.2	1731.2	1399.4	25.6	229.6	182.4	114.8
	Worst	1272.6	638.6	1707.0	1375.2	24.6	223.0	176.6	110.8
	Mean	1289.880	651.253	1722.569	1391.931	25.592	229.084	182.045	114.617
	SD	5.913	3.462	6.357	5.833	0.088	1.222	0.966	0.680
99%	Time	13.156	6.231	20.559	14.687	0.218	2.087	1.506	1.059
	Best	1304.2	653.8	1753.6	1423.4	26.6	237.4	182.4	115.8
	Worst	1275.4	638.8	1722.6	1395.8	25.4	229.8	177.2	110.6
	Mean	1296.050	649.628	1742.900	1415.859	26.587	236.442	182.102	115.597
	SD	6.409	3.289	6.927	6.131	0.120	1.440	0.886	0.762
100%	Time	13.275	6.303	20.719	14.816	0.225	2.094	1.522	1.081
	Best	1297.4	649.0	1744.2	1390.4	25.8	232.0	188.4	125.0
	Worst	1271.8	632.4	1715.2	1418.8	25.6	223.6	181.8	119.4
	Mean	1290.162	645.320	1736.270	1409.7	25.798	230.922	188.002	124.691
	SD	6.454	3.423	6.847	6.542	0.018	1.510	1.048	0.878

Mutation Rate 0.36%, Population Size 128, Generations 200, Difference 2

Average of 5 trials

Table C.3: Affect of Crossover Rate

Mutation Rate		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
0.0%	Time	13.168	6.262	20.538	14.728	0.225	2.056	1.500	1.069
	Best	1375.0	643.4	1854.4	1510.0	26.0	211.0	178.8	117.4
	Worst	1370.0	643.2	1846.8	1504.0	26.0	211.0	172.8	117.4
	Mean	1372.709	643.398	1850.914	1507.669	26.000	211.000	172.800	117.400
	SD	0.762	0.018	1.187	0.911	0.000	0.000	0.000	0.000
0.36%	Time	13.156	6.231	20.559	14.687	0.218	2.087	1.506	1.059
	Best	1304.2	653.8	1753.6	1423.4	26.6	237.4	182.4	115.8
	Worst	1275.4	638.8	1722.6	1395.8	25.4	229.8	177.2	110.6
	Mean	1296.050	649.628	1742.900	1415.859	26.587	236.442	182.102	115.597
	SD	6.409	3.289	6.927	6.131	0.120	1.440	0.886	0.762
1%	Time	13.085	6.178	20.428	14.606	0.222	2.072	1.510	1.050
	Best	1133.6	594.6	1549.0	1260.0	26.0	231.0	185.0	118.8
	Worst	1094.2	562.6	1504.6	1218.8	22.0	216.6	174.4	110.0
	Mean	1122.298	586.656	1535.802	1248.508	25.919	228.494	183.255	117.714
	SD	11.681	7.664	12.410	11.176	0.484	3.442	2.443	1.946
10%	Time	12.928	6.060	20.300	14.503	0.219	1.991	1.472	1.022
	Best	813.4	422.20	1236.4	982.2	26.2	165.4	140.6	94.4
	Worst	709.8	355.8	1113.6	883.4	14.8	119.0	103.8	63.4
	Mean	784.20	401.891	1198.752	952.087	24.172	152.081	129.516	85.225
	SD	30.604	20.839	35.936	31.384	2.846	13.267	11.491	9.782
20%	Time	12.903	6.040	20.281	14.491	0.219	1.78	1.463	1.009
	Best	761.0	394.4	1184.8	938.2	26.4	146.4	124.6	80.6
	Worst	656.8	317.6	1053.4	822.0	10.4	105.4	85.2	47.0
	Mean	728.006	369.211	1143.269	901.277	22.298	132.277	112.109	70.252
	SD	33.604	25.989	42.065	37.671	4.560	13.937	12.956	10.442

Crossover Rate 99%, Population Size 128, Generations 200, Difference 2

Average of 5 trials

Table C.4: Affect of Mutation Rate

Population Size		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
32	Time	3.172	1.497	4.953	3.553	0.063	0.503	0.360	0.250
	Best	1179.4	610.0	1612.2	1310.2	24.4	226.2	179.8	112.8
	Worst	1164.2	599.0	1591.4	1291.2	24.2	222.2	176.0	109.0
	Mean	1175.588	606.969	1605.731	1305.506	24.225	225.600	179.481	112.606
	SD	4.634	2.650	5.984	4.904	0.066	1.029	0.839	0.736
64	Time	6.497	3.072	10.131	7.259	0.109	1.025	0.741	0.528
	Best	1234.6	632.6	1688.0	1380.8	24.8	228.6	181.0	120.4
	Worst	1214.0	618.4	1664.4	1355.4	24.8	223.6	177.2	115.2
	Mean	1228.144	629.022	1679.972	1372.991	24.800	228.081	180.306	120.100
	SD	5.239	3.475	6.032	5.934	0.000	1.188	0.788	0.960
128	Time	13.156	6.231	20.559	14.687	0.218	2.087	1.506	1.059
	Best	1304.2	653.8	1753.6	1423.4	26.6	237.4	182.4	115.8
	Worst	1275.4	638.8	1722.6	1395.8	25.4	229.8	177.2	110.6
	Mean	1296.050	649.628	1742.900	1415.859	26.587	236.442	182.102	115.597
	SD	6.409	3.289	6.927	6.131	0.120	1.440	0.886	0.762
256	Time	26.440	12.559	41.281	29.531	0.450	4.181	3.025	2.128
	Best	1343.0	667.8	1816.2	1470.4	25.8	233.6	188.0	111.8
	Worst	1315.4	648.6	1778.0	1439.8	25.2	225.2	181.6	117.2
	Mean	1335.968	663.812	1805.094	1460.909	25.795	232.732	187.623	116.986
	SD	6.641	3.808	8.134	7.131	0.053	1.371	0.985	0.806
512	Time	53.138	25.319	82.963	59.256	0.897	8.422	6.087	4.294
	Best	1408.6	688.0	1866.4	1498.8	26.6	242.2	182.6	120.8
	Worst	1372.6	665.6	1825.0	1460.8	24.4	231.2	6.087	112.8
	Mean	1397.480	682.934	1854.175	1488.087	26.594	241.396	190.192	120.538
	SD	7.710	4.222	8.822	7.700	0.105	1.506	1.102	0.981

Mutation Rate 0.36%, Crossover Rate 99%, Generations 200, Difference 2

Average of 5 trials

Table C.5: Affect of Population Size

Generation Number		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
2	Time	13.156	6.231	20.559	14.687	0.218	2.087	1.506	1.059
	Best	1304.2	653.8	1753.6	1423.4	26.6	237.4	182.4	115.8
	Worst	1275.4	638.8	1722.6	1395.8	25.4	229.8	177.2	110.6
	Mean	1296.050	649.628	1742.900	1415.859	26.587	236.442	182.102	115.597
	SD	6.409	3.289	6.927	6.131	0.120	1.440	0.886	0.762
4	Time	13.166	6.244	20.516	14.666	0.222	2.062	1.509	1.053
	Best	1320.2	660.8	1756.2	1424.8	26.6	235.4	187.6	116.4
	Worst	1292.6	646.0	1729.2	1399.4	26.0	229.6	183.2	112.4
	Mean	1310.942	657.027	1746.861	1417.600	26.595	234.880	187.311	116.244
	SD	5.999	3.127	6.677	5.940	0.053	1.153	0.816	0.619
8	Time	13.165	6.231	20.562	14.700	0.221	2.072	1.528	1.063
	Best	1318.2	665.4	1784.8	1441.2	26.8	236.4	192.0	119.2
	Worst	1293.8	652.4	1756.2	1419.0	26.8	229.0	187.4	115.2
	Mean	1310.247	661.570	1774.266	1434.686	26.800	235.177	191.512	118.886
	SD	5.621	2.764	6.598	5.626	0.000	1.300	0.843	0.593
16	Time	13.178	6.247	20.535	14.700	0.231	2.071	1.500	1.062
	Best	1347.6	667.0	1791.0	1454.8	28.2	229.6	181.6	118.0
	Worst	1323.0	654.2	1764.0	1430.2	27.6	224.6	186.4	114.4
	Mean	1341.230	663.933	1783.509	1447.694	28.195	229.036	185.914	117.842
	SD	5.631	2.700	6.383	5.402	0.053	0.880	0.820	0.599
32	Time	13.175	6.266	20.553	14.710	0.231	2.069	1.516	1.056
	Best	1339.6	678.0	1805.4	1465.6	31.4	237.2	190.4	118.8
	Worst	1316.0	664.2	1778.0	1443.4	30.6	230.8	185.6	114.2
	Mean	1332.283	674.653	1795.847	1459.513	31.394	236.189	189.878	119.489
	SD	5.591	2.865	6.642	5.356	0.070	1.151	0.857	0.639

Mutation Rate 0.36%, Crossover Rate 99%, Generations 200, Population Size 128

Average of 5 trials

Table C.6: Affect of Balancing Difference

Appendix D

Local Search and Memetic Algorithm Experimental Results

Difference Size		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
2	Time	12.406	1.534	50.962	21.372	0.000	0.062	0.038	0.013
	Best	1687	698	2447	1684	25	244	195	119
	Worst	1646	663	2377	1643	23	231	185	99
	Mean	1670.8	677.2	2408.4	1658.8	24	237.6	191.2	107.4
	SD	13.673	11.923	29.486	18.236	0.632	5.161	3.709	6.946
4	Time	12.381	1.219	44.188	18.581	0.000	0.081	0.041	0.019
	Best	1714.0	758	2514	1927	27	260	199	136
	Worst	1698	734	2467	1833	24	235	187	121
	Mean	1702.8	749.6	2487	1884.0	25.0	247.0	194.2	126.2
	SD	5.776	8.452	18.815	31.509	1.095	8.695	4.261	5.154
8	Time	12.050	1.238	40.175	16.741	0.000	0.106	0.056	0.028
	Best	1707	805	2597	1980	26	258	197	136
	Worst	1677	746	2516	1915	23	233	190	120
	Mean	1693.4	765.4	2549.2	1946.2	25.0	249.6	193.6	127.6
	SD	9.912	21.332	27.967	22.516	1.095	8.868	2.653	5.314
16	Time	12.187	1.434	38.672	17.000	0.003	0.122	0.069	0.050
	Best	1699	775	2625	2002	29	265	202	136
	Worst	1684	744	2541	1945	25	239	196	125
	Mean	1693.4	759.0	2580.2	1971	27.8	253.6	198.6	128.8
	SD	5.161	12.033	38.672	22.423	1.470	8.709	2.417	3.868
32	Time	12.522	1.641	40.900	18.184	0.003	0.125	0.069	0.056
	Best	1703	802	2666	2017	32	265	199	130
	Worst	1690	736	2519	1984	32	225	185	110
	Mean	1698.0	777.4	2597.8	2004.0	32.0	248.4	192.6	119.6
	SD	4.427	22.931	51.658	10.918	0.000	13.017	5.571	6.859

Average of 5 trials

Table D.1: Affect of Difference Size on Local Search Algorithm

Generation Number		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
2	Time	27.906	6.537	97.000	34.819	0.219	2.075	1.488	1.044
	Best	1665.0	677.8	2371.0	1683.2	26.6	246.0	185.4	116.2
	Worst	1275.4	638.8	1722.6	1395.8	25.4	229.8	177.2	110.6
	Mean	1301.763	650.061	1752.622	1420.009	26.587	236.569	182.131	115.600
	SD	45.894	4.645	77.525	34.819	0.120	1.872	0.954	0.773
4	Time	43.003	6.912	172.522	55.316	0.219	2.094	1.494	1.044
	Best	1668.4	679.0	2368.44	1687.6	26.6	247.4	185.2	115.8
	Worst	1275.4	639.6	1723.6	1395.8	25.4	229.8	177.2	110.6
	Mean	1307.544	650.472	1762.191	1424.123	26.587	236.700	182.161	115.598
	SD	64.335	5.646	107.494	46.651	0.120	2.275	1.017	0.762
8	Time	73.384	7.637	328.153	96.166	0.219	2.125	1.503	1.047
	Best	1672.2	677.8	2378.2	1686.8	26.6	248.6	185.8	115.8
	Worst	1275.4	638.8	1722.6	1396.6	25.4	229.8	177.2	110.6
	Mean	1319.133	651.234	1781.694	1432.405	26.587	236.978	182.250	115.606
	SD	89.698	6.968	150.509	64.569	0.120	2.833	1.118	0.593
16	Time	132.878	9.116	631.022	177.456	0.218	2.191	1.525	1.050
	Best	1672.8	679.2	2373.6	1687.8	26.6	248.2	187.8	115.8
	Worst	1275.4	638.8	1722.6	1397.2	25.6	229.8	177.2	110.6
	Mean	1342.042	652.767	1820.250	1448.813	26.589	237.472	182.436	115.627
	SD	121.663	9.088	204.927	87.761	0.102	3.434	1.406	0.704
32	Time	253.025	12.244	1244.731	341.675	0.225	2.319	1.566	1.059
	Best	1673.8	682.0	2389.2	1691.8	26.6	248.4	187.0	116.2
	Worst	1275.4	639.0	1723.2	1395.8	25.4	230.4	177.6	110.8
	Mean	1387.969	656.133	1897.809	1482.097	26.587	238.541	182.691	115.641
	SD	159.646	11.560	268.221	115.091	0.120	4.199	1.498	0.700

Crossover Rate 99%, Mutation Rate 0.36%, Population Size 128, Generations 200, Difference 2

Average of 5 trials

Table D.2: Exhausted Memetic Algorithm: Effect of Number of Random Individuals

Generation Number		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
2	Time	27.091	8.859	57.650	33.359	0.222	2.272	1.594	1.088
	Best	1437.8	693.0	1867.4	1483.8	26.6	254.8	194.6	118.8
	Worst	1393.6	675.2	1820.4	1449.0	26.0	247.4	188.4	114.0
	Mean	1417.719	688.230	1844.327	1467.673	26.595	253.984	194.248	118.583
	SD	7.885	3.697	8.155	6.915	0.053	1.371	0.995	0.798
4	Time	41.134	11.360	94.6782	52.175	0.231	2.419	1.688	1.131
	Best	1456.6	689.6	1894.4	1484.8	26.6	246.6	194.6	123.6
	Worst	1413.4	673.6	1847.2	1448.2	25.4	240.0	189.2	118.2
	Mean	1433.056	685.414	1870.266	1469.692	26.587	245.995	194.280	123.200
	SD	7.997	3.482	8.771	6.802	0.120	1.272	0.906	0.844
8	Time	68.869	16.253	168.641	89.875	0.244	2.750	1.869	1.200
	Best	1489.0	695.4	1913.2	1487.6	26.6	253.6	195.2	125.0
	Worst	1439.4	677.8	1865.6	1450.4	25.4	247.0	189.8	119.6
	Mean	1465.739	691.667	1888.138	1472.084	26.587	252.836	194.850	124.822
	SD	9.026	3.795	9.768	7.347	0.120	1.297	0.968	0.711
16	Time	124.050	25.481	315.738	164.560	0.269	3.397	2.197	1.335
	Best	1501.4	702.4	1939.6	1507.8	27.0	256.0	198.8	123.4
	Worst	1456.2	686.2	1889.8	1469.2	26.8	246.2	194.2	118.0
	Mean	1479.005	699.167	1915.484	1492.839	26.998	255.039	198.530	123.209
	SD	9.535	3.555	10.791	8.012	0.018	3.397	2.197	0.809
32	Time	232.634	44.353	607.694	313.090	0.325	4.644	2.962	1.600
	Best	1541.4	706.4	1968.4	1527.8	27.0	254.4	199.8	120.4
	Worst	1495.2	685.6	1917.4	1489.0	26.2	248.2	194.2	115.4
	Mean	1522.258	701.981	1945.864	1513.689	26.994	253.919	199.477	120.238
	SD	10.788	3.796	12.374	8.539	0.070	1.192	0.956	0.698

Crossover Rate 99%, Mutation Rate 0.36%, Population Size 128, Generations 200, Difference 2

Average of 5 trials

Table D.3: Intermediate Memetic Algorithm: Effect of Number of Random Individuals

Generation Number		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
5	Time	121.687	23.119	311.681	163.469	0.253	3.175	2.122	1.312
	Best	1621.6	707.8	2088.4	1579.8	26.6	249.2	197.8	117.2
	Worst	1573.6	688.8	2028.4	1536.0	26.4	242.0	191.8	112.4
	Mean	1600.189	703.455	2062.164	1559.652	26.598	248.594	197.341	117.011
	SD	10.746	3.819	12.747	9.549	0.018	1.344	1.020	0.732
10	Time	68.869	16.253	168.641	89.875	0.244	2.753	1.869	1.200
	Best	1489.0	695.4	1913.2	1487.6	26.6	253.6	195.2	125.0
	Worst	1439.4	677.8	1865.6	1450.4	25.4	247.0	189.8	119.6
	Mean	1465.739	691.667	1888.138	1472.084	26.587	252.836	194.850	124.822
	SD	9.026	3.795	9.768	7.347	0.120	1.297	0.968	0.711
20	Time	41.147	12.034	94.519	52.131	0.234	2.466	1.703	1.119
	Best	1402.0	689.2	1858.4	1462.0	26.6	248.4	196.6	118.8
	Worst	1357.0	668.6	1807.6	1423.0	25.4	241.0	190.8	114.6
	Mean	1375.853	681.348	1831.906	1441.834	26.587	247.178	196.255	118.628
	SD	8.237	3.778	9.577	7.388	0.120	1.373	1.008	0.669
30	Time	29.990	9.947	65.081	37.109	0.225	2.353	1.634	1.084
	Best	1365.4	675.8	1805.8	1443.6	26.6	253.6	192.4	112.2
	Worst	1339.4	660.0	1774.2	1416.4	25.8	245.2	186.8	107.4
	Mean	1358.736	672.753	1797.297	1436.211	26.594	252.805	192.053	112.011
	SD	6.363	3.672	7.536	6.265	0.070	1.457	0.960	0.709
40	Time	26.953	9.197	57.313	33.219	0.225	2.269	1.603	1.069
	Best	1382.8	682.6	1803.6	1428.6	26.6	244.0	194.0	118.0
	Worst	1329.0	656.4	1749.6	1388.2	25.4	234.2	187.8	112.8
	Mean	1350.305	670.917	1771.213	1405.692	26.587	240.877	193.052	117.831
	SD	9.617	4.358	9.820	7.124	0.120	1.316	1.030	0.727

Crossover Rate 99%, Mutation Rate 0.36%, Population Size 128, Generations 200, Difference 2

Average of 5 trials

Table D.4: Intermediate Memetic Algorithm: Effect of Generation Size between Local Search

Generation Number		struct	prim1	prim2	ind1	pcb1	chip1	chip4	frac
3	Time	32.275	10.468	70.497	39.969	0.244	2.509	1.750	1.153
	Best	1338.4	673.0	1802.2	1436.8	26.6	250.0	194.0	120.0
	Worst	1303.6	653.0	1761.0	1407.0	25.4	241.4	189.0	115.8
	Mean	1324.120	667.308	1785.816	1424.098	26.587	249.164	193.836	119.834
	SD	6.876	3.791	8.018	6.286	0.120	1.496	1.003	0.641
6	Time	50.850	13.803	119.856	64.988	0.244	2.681	1.825	1.194
	Best	1413.4	689.8	1847.6	1468.8	26.6	252.6	194.0	126.6
	Worst	1370.2	673.2	1804.0	1439.2	25.4	244.4	188.6	121.2
	Mean	1394.509	686.078	1828.450	1455.889	26.587	252.600	193.634	126.381
	SD	7.909	3.563	8.322	6.525	0.120	1.404	0.986	0.828
9	Time	68.869	16.253	168.641	89.875	0.244	2.753	1.869	1.200
	Best	1489.0	695.4	1913.2	1487.6	26.6	253.6	195.2	125.0
	Worst	1439.4	677.8	1865.6	1450.4	25.4	247.0	189.8	119.6
	Mean	1465.739	691.667	1888.138	1472.084	26.587	252.836	194.850	124.822
	SD	9.026	3.795	9.768	7.347	0.120	1.297	0.968	0.711
12	Time	86.007	17.588	215.597	114.031	0.244	2.788	1.897	1.203
	Best	1559.2	696.8	2008.4	1534.6	25.4	246.0	198.2	117.2
	Worst	1512.8	680.4	1949.4	1495.6	26.6	239.0	191.0	112.8
	Mean	1538.889	694.175	1979.005	1515.731	26.587	245.436	197.639	116.813
	SD	9.237	3.492	11.019	7.703	0.120	1.257	1.071	0.718
15	Time	102.553	18.228	262.197	138.044	0.244	2.841	1.910	1.197
	Best	1616.2	689.8	2070.8	1568.4	26.6	255.2	194.2	116.8
	Worst	1568.2	674.6	2009.4	1523.8	25.4	247.6	189.0	111.6
	Mean	1593.486	686.570	2044.948	1547.180	26.587	254.545	193.850	116.600
	SD	9.904	3.590	11.936	8.783	0.120	1.498	0.960	0.737

Crossover Rate 99%, Mutation Rate 0.36%, Population Size 128, Generations 200, Difference 2

Average of 5 trials

Table D.5: Exhausted Memetic Algorithm: Effect of Number of Iterations of Local Search

Bibliography

[Abra97] David Abramson, Paul Logothetis, Adam Postula, and Marcus Randall, "Application specific computers for combinatorial optimisation," In *Australasian Computer Architecture Conference, Sydney, Australia*, pp. 29–44, Springer-Verlag, Singapore, 1997.

[Ambl89] A.P. Ambler, "Hardware accelerators for cad," In *Computer-Aided Engineering Journal*, 1989.

[Apor01] Chatchawit Aporntewan and Prabhas Chongstitvatana, "A hardware implementation of the compact genetic algorithm," In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pp. 624–629, IEEE Press, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 2001.

[Arei00] S. Areibi, "A review of circuit partitioning," School of Engineering, Technical Report, University of Guelph, 2000.

[Arei01] Shawki Areibi, "Memetic Algorithms for VLSI Physical Design: Implementation Issues," In *Genetic and Evolutionary Computation Conference (GECCO)*, 2001.

[Arei93] Shawki Areibi and Anthony Vannelli, "Circuit partitioning using a tabu search approach," In *ISCAS*, pp. 1643–1646, 1993.

[Arei94] S. Areibi and A. Vannelli, "Advanced search techniques for circuit partitioning," In P. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, pp. 77–96, AMS, 1994.

[Baza99] Kia Bazargan, Ryan Kastner, and Majid Sarrafzadeh, "3-d floorplanning: Simulated annealing and greedy placement methods for reconfigurable computing systems," In *IEEE International Workshop on Rapid System Prototyping*, pp. 38–, 1999.

[Beas93a] David Beasley, David R. Bull, and Ralph R. Martin, "An overview of genetic algorithms: Part 1, fundamentals," *University Computing*, vol. 15, No. 2, pp. 58–69, 1993.

[Beas93b] David Beasley, David R. Bull, and Ralph R. Martin, "An overview of genetic algorithms: Part 2, research topics," *University Computing*, vol. 15, No. 4, pp. 170–181, 1993.

[Bish98] William D. Bishop, "Reconfigurable Hardware Objects for Dynamic Memory Management," In *Proceedings of the 1998 CITO Researcher Retreat*, Hamilton, Ontario, Canada, May 1998.

[CD88] T.B.M. Carlstedt-Duke, "A solution to high performance acceleration of digital system design," In *IEEE Colloquium on Hardware Accelerators for VLSI CAD - A Tutorial*, 1988.

[Celo03a] Celoxica, "Handel-c language reference manual for dk 2.0," 2003.

[Celo03b] Celoxica, "http://www.celoxica.com/ (accessed: July 7, 2004)," 2003.

[Chan97] Pak K. Chan and Martine D. F. Schlag, "Acceleration of an FPGA router," In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 175–181, IEEE Computer Society Press, Los Alamitos, CA, 1997.

[Coho03] James Cohoon, John Karro, and Jens Lienig, "Evolutionary algorithms for the physical design of vlsi circuits," In *Advances in evolutionary computing: theory and applications*, pp. 683–711, Springer-Verlag New York, Inc., 2003.

[Comp00a] K. Compton and S. Hauck, "An introduction to reconfigurable computing," 2000.

[Comp00b] K. Compton and S. Hauck, "An introduction to reconfigurable computing," 2000.

[Comp00c] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," 2000.

[Comp99] K. Compton and S. Hauck, "Configurable computing: A survey of systems and software," 1999.

[De J89] Kenneth A. De Jong and William M. Spears, "Using genetic algorithm to solve NP-complete problems," In James D. Schaffer, editor, *Proc. of the Third Int. Conf. on Genetic Algorithms*, pp. 124–132, Morgan Kaufmann, San Mateo, CA, 1989.

[DeHo99] André DeHon and John Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," In *Proceedings of the 36th ACM/IEEE conference on Design automation conference*, pp. 610–615, ACM Press, 1999.

[Fidu82] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," In *Proceedings of the 19th conference on Design automation*, pp. 175–181, IEEE Press, 1982.

[Fidu88] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," In *Papers on Twenty-five years of electronic design automation*, pp. 241–247, ACM Press, 1988.

[Glov95] Glover, F. and Kelly, J. P. and Laguna, M., "Genetic algorithms and tabu search: Hybrids for optimization," *Computers Ops Research*, vol. 22, pp. 111–134, 1995.

[Grah95] Paul Graham and Brent Nelson, "A hardware genetic algorithm for the travelling salesman problem on SPLASH 2," In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications*, pp. 352–361, Springer-Verlag, Berlin, / 1995.

[Grah96] P. Graham and B. Nelson, "Genetic algorithms in software and in hardware - A performance analysis of workstations and custom computing machine implementations," In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 216–225, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Gurw03] Gurwant Koonar, *A Reconfigurable Hardware Implementation of Genetic Algorithmjs for VLSI CAD Design* Master's thesis, University of Guelph, 2003.

[Hama97] Y. Hamadi and D. Merceron, "Reconfigurable architectures: A new vision for optimization problems," 1997.

[Hauc98] S. Hauck, "The future of reconfigurable systems," 1998.

[Hayk99] Simon Haykin, *Neural Networks: A Comprehensive Foundation, Second Edition*, Prentice Hall PTR, 1999.

[Kang03] Sung-Mo Kang and Yusuf Leblebici, *CMOS Digital Integrated Circuits: Analysis and Design (Third Edition)*, pp. 244, McGraw-Hill, New York, 2003.

[Kern70] B. Kernighan and S. Lin, "An effective heuristic procedure for partitioning graphs," *The Bell Systems Technical Journal*, pp. 291–308, 1970.

[Kirk83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[Koza97] John R. Koza, Forrest H Bennett III, Jeffrey L. Hutchings, Stephen L. Bade, Martin A. Keane, and David Andre, "Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays," In Tetsuya Higuchi, editor, *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, pp. 27–32, Nagoya, 1997.

[Loo02] S M Loo, B Earl Wells, N Freije, and J Kulick, "Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time Systems," In *Southeastern Symposium on System Theory*, 2002.

[Luo99] Zhen Luo, Margaret Martonosi, and Pranav Ashar, "An edge-endpoint-based configurable hardware architecture for VLSI CAD layout design rule checking," In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 158–167, IEEE Computer Society Press, Los Alamitos, CA, 1999.

[Mall88] Sivanarayana Mallela and Lov K. Grover, "Clustering based simulated annealing for standard cell placement," In *Proceedings of the 25th ACM/IEEE conference on Design automation*, pp. 312–317, IEEE Computer Society Press, 1988.

[Mart01] Peter Martin, "A hardware implementation of a genetic programming system using FPGAs and Handel-C," *Genetic Programming and Evolvable Machines*, vol. 2, No. 4, pp. 317–343, 2001.

[Mart02a] Peter Martin, "A Pipelined Hardware implementation of Genetic Programming using FPGAs and Handel-C," Technical Report, Department of Computer Science, University of Essex, 2002.

[Mart02b] Peter Martin, "An analysis of random number generators for a hardware implementation of genetic programming using fpgas and handel-c," 2002.

[Mazu99] Pinaki Mazumder and Elizabeth M. Rudnick, *Genetic algorithms for VLSI design, layout & test automation*, Prentice Hall PTR, 1999.

[MCNC90] MCNC, "1990 MCNC Layout Benchmark Set," 1990.

[Megs98] Megson and Bland, "Synthesis of a systolic array genetic algorithm," In *IPPS: 11th International Parallel Processing Symposium*, IEEE Computer Society Press, 1998.

[Mich94] Zbigniew Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs (Second Edition)*, Springer-Verlag, New York, 1994.

[Mitc96] Melanie Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, Massachusetts, 1996.

[Moor65] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, No. 8, pp. , 1965.

[Nich03] Kristian Nichols, *A Reconfigurable Architecture for Artificial Neural Networks* Master's thesis, University of Guelph, 2003.

[Perk00] Perkins, S. and Porter, R. and Harvey, N.R., "Everything on the chip: A hardware-based self-contained spatially- structured genetic algorithm for signal processing," *Evolvable Systems: From Biology to Hardware: Proc. 3rd International Conference on Evolvable Systems (ICES 2000), Lecture Notes in Computer Science*, vol. 1801, pp. 165–174, 2000.

[Plat98] Marco Platzner and Giovanni De Micheli, "Acceleration of satisfiability algorithms by reconfigurable hardware," In Reiner W. Hartenstein and Andres Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pp. 69–78, Springer-Verlag, Berlin, / 1998.

[Pres92] W H Press, S A Teukolsky, W T Vetterling, and B P Flannery, *Numerical Recipes in C: The Art of Scientic Computing (Second Edition)*, Cambridge University Press, Cambridge, 1992.

[Rama] Pratap Ramamurthy and Jai Vasanth, "VLSI Implementation of Genetic Algorithm," under review by ACM conference.

[Reev02] Colin R. Reeves and Jonathan E. Rowe, *Genetic Algorithms: Principles and Perspectives: A Guide to GA Theory*, Kluwer Academic Publishers, 2002.

[Scot95] Stephen D. Scott, Ashok Samal, and Sharad C. Seth, "HGA: A hardware-based genetic algorithm," In *FPGA*, pp. 53–59, 1995.

[Shac01] Shackleford, Barry and Snider, Greg and Carter, Richard J. and Okushi, Etsuko and Yasuda, Mitsuhiro and Seo, Katsuhiko and Yasuura, Hiroto, "A high-performance, pipelined, fpga-based genetic algorithm machine," *Genetic Programming and Evolvable Machines*, vol. 2, No. 1, pp. 33–60, 2001.

[Sitk95] Nathan Sitkoff, Mike Wazlowski, Aaron Smith, and Harvey Silverman, "Implementing a genetic algorithm on a parallel custom computing machine," In *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines (FCCM'95)*, pp. 180–187, IEEE Press, 1995.

[Smit98] J E Smith and F Vavak, "Replacement strategies in steady state genetic algorithms: Static environments," In *Foundations of Genetic Algorithms 5*, Morgan Kaufmann, 1998.

[Supp01] Celoxica Customer Support, "Rc1000 hardware reference manual," 2001.

[Supp02] Celoxica Support, "http://www.celoxica.com/support/view_article.asp ?ArticleID=360 (Accessed: June 25, 2004)," 2002.

[Whit94] Darrell Whitley, "A genetic algorithm tutorial," *Statistics and Computing*, vol. 4, pp. 65–85, 1994.

[Wrig03] Michael G. Wrighton and Andr M. DeHon, "Hardware-assisted simulated annealing with application for fast fpga placement," In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pp. 33–42, ACM Press, 2003.

[Xili03] Xilinx, "Corporate backgrounder," 2003.

[Zhon98a] Peixin Zhong, Pranav Ashar, Sharad Malik, and Margaret Martonosi, "Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with boolean satisfiability," In *Design Automation Conference*, pp. 194–199, 1998.

[Zhon98b] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik, "Accelerating boolean satisfiability with configurable hardware," In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 186–195, IEEE Computer Society Press, Los Alamitos, CA, 1998.