THESIS

A LOW COST INDIVIDUAL-WELL ADAPTIVE BODY BIAS (IWABB) SCHEME FOR LEAKAGE POWER REDUCTION, PERFORMANCE ENHANCEMENT AND MANUFACTURING YIELD IMPROVEMENT IN THE PRESENCE OF INTRA-DIE VARIATIONS

Submitted by
Justin Gregg
Department of Electrical and Computer Engineering

In partial fulfillment of the requirements
for the Degree of Masters of Science in Electrical Engineering
Colorado State University
Fort Collins, Colorado
Summer 2003

Copyright © Justin Gregg 2003
All Rights Reserved

COLORADO STATE UNIVERSITY

—

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY JUSTIN GREGG ENTITLED A LOW COST INDIVIDUAL-WELL ADAPTIVE BODY BIAS (IWABB) SCHEME FOR LEAKAGE POWER REDUCTION, PERFORMANCE ENHANCEMENT AND MANUFACTURING YIELD IMPROVEMENT IN THE PRESENCE OF INTRA-DIE VARIATIONS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTERS OF SCIENCE IN ELECTRICAL ENGINEERING.

Committee on Graduate Work

____

____

____
Adviser

____
Department Head

ABSTRACT OF THESIS

A LOW COST INDIVIDUAL-WELL ADAPTIVE BODY BIAS (IWABB) SCHEME FOR LEAKAGE POWER REDUCTION, PERFORMANCE ENHANCEMENT AND MANUFACTURING YIELD IMPROVEMENT IN THE PRESENCE OF INTRA-DIE VARIATIONS

As VLSI technologies scale smaller, process variations continue to increase, causing larger variations from design targets in circuit operating frequencies and power. These wider variations reduce manufacturing yields and therefore profits. This thesis presents a new method of adapting body biasing on a chip during post-fabrication testing in order to mitigate the effects of process variations. Individual well biasing voltages can be connected either to a chip wide well bias or to a different bias voltage through a self-regulating mechanism, allowing bias voltage adjustments on a per well basis. The scheme requires at most one bias voltage distribution network and very small silicon area, but allows for back biasing adjustments to more effectively mitigate die-to-die and within-die process variations. This thesis investigates using a single-objective evolutionary algorithm, a gradient and local random walk algorithm, and a multiple-objective evolutionary algorithm to determine the bias setting for each well. Our experimental results show binning yields as low as 12% can be improved to nearly 80% after using the proposed IWABB method.

Justin Gregg
Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, Colorado 80523
Summer 2003

ACKNOWLEDGMENTS

This work is partially funded and supported by the Hewlett-Packard Company (HP). Thanks to all the people in ETL for their patience while the simulations for this work took over the compute cluster.

The soIWABB algorithm is based on code originally developed in conjunction with David Bolme and Jeff Boody for a project in Dr. Adele Howe's artificial intelligence course. Development of the code was continued under the supervision of Dr. Tom Chen in his advanced VLSI course.

Some circuit translators and circuit object code were graciously provided Dustin Perkins.

DEDICATION

To all the people along the way

TABLE OF CONTENTS

1 Introduction **14**

2 Existing Work **16**

3 Background **18**

3.1 Process variations 18

3.1.1 Sources of process variations 18

3.1.2 Effects of process variations 22

3.1.3 Mitigating effects of process variation 23

3.1.4 Modeling process variations 25

3.2 Search algorithms 26

3.2.1 Random walks, hill climbing and simulated annealing algorithms 27

3.2.2 Gradient based search algorithms 29

3.2.3 Evolutionary algorithms 30

3.2.3.1 Single-objective evolutionary optimization 31

3.2.3.2 Multiple-objective evolutionary optimization 33

3.2.4 Heuristics in search algorithms 36

4 Proposed Method **37**

4.1 Circuit structure 37

4.1.1 Simulated well capacitance 37

4.1.2 Well voltage control structures 41

4.1.3 Balancing pullup and voltage divider sizing, activity factor and well polarity 41

4.1.4 Test circuits . . . 46
4.2 IWABB algorithms . . . 48
4.2.1 Pseudo-weighted aggregate SOEA IWABB (soIWABB) . . . 50
4.2.2 Gradient-local random walk IWABB (gIWABB) . . . 54
4.2.3 MOEA IWABB (moIWABB) . . . 57

5 Experimental Results **59**
5.1 Initial investigations . . . 61
5.1.1 Fully floating and voltage divider controlled wells . . . 61
5.1.2 Required biasing resolution . . . 61
5.1.3 Use of V_{b_n} biasing . . . 62
5.2 soIWABB results . . . 64
5.2.1 Convergence . . . 64
5.2.2 Test chip results . . . 64
5.2.3 *adder32* results . . . 65
5.2.4 *adder32* solutions summary . . . 67
5.3 gIWABB results . . . 70
5.3.1 Convergence . . . 70
5.3.2 Test chip results . . . 71
5.3.3 *adder32* results . . . 71
5.3.4 *L64buP* results . . . 71
5.4 moIWABB results . . . 75
5.4.1 Convergence . . . 75
5.4.2 Test chip results . . . 76
5.4.3 *adder32* results . . . 78

6 Discussion and Conclusion **82**
6.1 Analysis . . . 82
6.1.1 Algorithm effectiveness . . . 83
6.1.2 Algorithm efficiency . . . 84
6.1.3 Mode effectiveness . . . 84
6.1.3.1 Yield improvement metric . . . 85
6.1.3.2 Objective-space slope metric . . . 86
6.1.4 IWABB value . . . 88

6.2 Conclusion 89
6.3 Future Work 90

REFERENCES **91**

A gIWABB source code **142**

B moIWABB source code **159**

C soIWABB and gIWABB analysis tools source code **182**

D moIWABB analysis tools source code **184**

LIST OF FIGURES

1.1 Trends of process variations based on 1997 NTRS and IBM process information. . . 15

3.1 Basic top view and cut away schematic representation of a MOS transistor showing basic dimensions. 19

3.2 Exaggerated example of possible process variations caused by inconsistencies in photoresist development. 20

3.3 Diagram of a simple aberration-free ($OPD = 0$) reducing projector. 21

3.4 (a) Random dopant atom placement from a three-dimensional Monte Carlo simulation. (b) IV characteristics from 100 random dopant placements.[22] 21

3.5 Example of P_{op} and f_{op} for 100 32-bit static CMOS ripple adders on a 100nm process technology. 24

3.6 Flow chart of a random walk algorithm. 28

3.7 Flow chart of a local hill climbing search algorithm. 29

3.8 Flow chart of a basic evolutionary algorithm. 31

3.9 Example of uniform crossover for a 10 bit chromosome. Colored bits represent bits chosen to build the child chromosome. 32

3.10 (a) Illustration of a SOEA population after many generations in a two dimensional objective space (minimization of both objectives). (b) Dominance and pareto optimality in a two dimensional objective space (minimization of all objectives). 33

3.11 (a) Illustration of (a) SPEA2 and (b) NSGA2 fitness calculations. 34

4.1 Schematic of inverter chain test circuit to investigate body voltage bounce when nwell parasitic resistance and capacitance are included. 38

4.2 Body voltage bounce in the inverter chain shown in Figure 4.1 with the nwell biased to 0.8V by an ideal source, $W_{p1} = W_{p2} = 1\mu$m and the nfets appropriately sized to balance the inverters. 38
4.3 Comparison of body voltage bounce V_{pb1} in the inverter chain shown in Figure 4.1 with the nwell (a) floated and (b) biased to 0.8V for varying W_{p1} and W_{p2}. 40
4.4 Change in total inverter chain (Figure 4.1) delay from 0.8V biased nwell to floated nwell $((d_{floated} - d_{biased})/\text{mean}(d_{floated}))$ for varying W_{p1} and W_{p2}. 41
4.5 Well voltage control structures. 42
4.6 Comparison of V_{b_p} bounce for the circuit shown in 4.1 with (a) an ideal 0.8V source well contact and (b) a 1.0μm wide, minimum length pullup pfet. 43
4.7 Floating pfet body voltage at inv1 of Figure 4.1 with a 1.0μm wide pullup pfet. The length of the pullup varies from one to three times the minimum and the nwell bias voltage varies from 1.08V to 1.18V. 44
4.8 P_{op} versus f_{op} for the *adder32* test circuit with activity factors of about 7% and 20%. The nwell bias voltage varies from $V_{dd} - 0.4$V to $V_{dd} + 0.2$V. Power and frequency are normalized to the high activity circuit with $V_{b_p} = V_{dd}$. 44
4.9 Circuit delay, P_{op}, power-delay product and energy-delay product for the *adder32* test circuit with varying V_{b_p}. All quantities are normalized to their values at $V_{b_p} = V_{dd}$. 45
4.10 Voltage divider sizing tests based on (a) steady-state V_{b_p} and (b) voltage-divider static current with 16μm switching inverter in the nwell for varying voltage divider component lengths. 46
4.11 Graphical representation of the structure of the correlation matrices used in (a) *adder32* and (b) *L64buP* Monte Carlo manufacturing variability simulation. 47
4.12 Pre-IWABB power and frequency characteristics of 100 (a) *adder32* and (b) *L64buP* chips generated by the Monte Carlo simulation shown with the target power-frequency pairs (f_t, P_t) used in IWABB testing. 48
4.13 Correlation of P_{op} and f_{op} to minimum, mean and maximum L_{eff} deviations from the mean shown with their repsective r values. 49
4.14 Vector addition in objective space of two NWF configurations for an example chip from *adder32*. 56
4.15 Initial comparison between SPEA2 and NSGA2 on an *adder32* test chip. 57

5.1 L_{eff} deviation from the mean in σ for test chips 101–104. 60

5.2 Comparison of yield improvement for NWF with floating and voltage-divider controlled nwells using gIWABB. 62
5.3 Comparison of yield improvement for 30mV and 100mV bias resolutions using gIWABB. 63
5.4 Yield improvement from gIWABB on *adder32* using V_{b_n} biasing. 63
5.5 Convergence of NWF fitness with soIWABB. 64
5.6 Comparison of solutions from NWF convergence testing with soIWABB using 10, 50, 100, 200, 400 and 800 generations with a population size of 30. 65
5.7 soIWABB NWF results for *adder32* test chips 101–104. 66
5.8 soIWABB *adder32* yield improvement plot. 66
5.9 Summary of the best configurations found by soIWABB with NWF+NWB+VDD. . 68
5.10 Summary of the best configurations found by soIWABB with (a) NWF+VDD and (b) NWF+NWB. 68
5.11 Summary of the best configurations found by soIWABB with (a) NWF and (b) VDD. 69
5.12 Summary of the best configurations found by soIWABB with (a) NWB+VDD and (b) DWB. 69
5.13 Convergence of NWB+VDD fitness with gIWABB. 70
5.14 Comparison of solutions from NWB+VDD convergence testing with gIWABB using 10, 50, 100, 200, 400 and 800 generations with a population size of 30. 71
5.15 gIWABB NWF results for *adder32* test chips 101–104. Average evaluations do not include test evaluations. 72
5.16 gIWABB *adder32* yield improvement plot. 72
5.17 gIWABB *L64buP* yield improvement plot. 73
5.18 (a) Evolution of the pareto front from NWF convergence testing with moIWABB. (b) Convergence of NWF simulated fitness from moIWABB. 75
5.19 Comparison of solutions from NWF convergence testing with moIWABB using 1, 2, 5, 10, 20, 40 and 50 generations with a population size of 30. 76
5.20 moIWABB NWF results for *adder32* test chips 101–104. 77
5.21 Illustration of falsely low yield from moIWABB. 78
5.22 moIWABB *adder32* results with interpolated best delay points for a give P_{max}. . . . 80
5.23 moIWABB *adder32* yield improvement plot showing minimum improvement with error bars to maximum improvement. 81

6.1 Illustration of possible objective-space slopes from moIWABB solutions. 87

6.2 Summary of objective-space slopes of all chips initially below P_{max} for each mode from the moIWABB algorithm. 88
6.3 Summary of objective-space slopes of all chips initially above P_{max} for each mode from the moIWABB algorithm. 88
6.4 Example of initial and final chip distributions in objective space for *adder32*. 89

LIST OF TABLES

4.1 Pseudo code for the soIWABB algorithm. 51
4.2 Pseudo code for the **evaluate** function used in soIWABB, gIWABB and moIWABB . 53
4.3 Pseudo code for the **crossover** function in soIWABB. 54
4.4 Pseudo code for the **mutate** function used in soIWABB. 55
4.5 Pseudo code for gIWABB algorithm. 55
4.6 Pseudo code for moIWABB algorithm. 58

5.1 soIWABB *adder32* yield improvement results. 67
5.2 gIWABB *adder32* yield improvement results. 73
5.3 gIWABB *L64buP* yield improvement results. 74
5.4 moIWABB *adder32* minimum/maximum yield improvement results. 79

6.1 Evaluations made per chip averaged over all modes and yields for each algorithm. . . 84
6.2 Difference between final and initial yields averaged over all initial yields for all biasing modes. 86

Chapter 1

Introduction

During the past twenty years, the semiconductor industry has pursued and accomplished the seemingly unattainable goal of continuous exponential growth. The scaling of the MOSFET transistor dimensions has been key to the ever increasing transistor density in microelectronic chips. In its adherence to Moore's Law, the industry has pushed through many obstacles, road blocks and brick walls. One bump in the road facing the industry now is process variations.

Process variations are deviations from the designed specification for any parameter caused by errors and inconsistencies during the manufacturing process. Process variations have been around since the inception of semiconductor manufacturing, but process controls have not scaled as quickly as the critical dimensions. Figure 1.1 shows the trends of relative deviation from specifications for the effective gate length (L_{eff}), wire width (W), height (H), thickness (T), and resistivity (ρ), gate oxide thickness (t_{ox}), threshold voltage (V_{th}), and power supply voltage (V_{dd})[1]. The data in this plot is based on 3σ variation information published in the 1997 National Technology Roadmap for Semiconductions (NTRS) report as well as process information from IBM. The most dominant feature in this data is the rapid increase in variation of L_{eff}, moving up towards 50%. However, the International Technology Roadmap for Semiconductors (ITRS) paints a different picture for L_{eff} variation, having it remain constant at 10% of the critical dimension[2]. While this optimistic view would be nice, the truth probably lies between the ITRS and NTRS/IBM data. Most sources report L_{eff} variations increasing from 10% in 180nm technology to 20% or more in 50nm technology[3; 4].

Process variations can be divided into four classes: lot-to-lot, wafer-to-wafer, die-to-die, and within-die. In older process technologies, most variations were associated with the first three classes. These variations can be addressed at the chip level with some of the techniques discussed in Chapter 2. However, as technology scales, within-die variations account for a growing percentage of the overall variations. These fluctuations can only be addressed on a sub-chip (e.g. sub-circuit) level.

The effects of process variations will be detailed in Section 3.1.2, but generally they cause changes

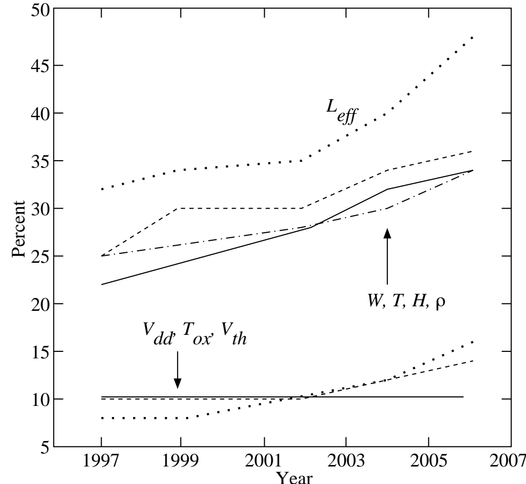


Figure 1.1: Trends of process variations based on 1997 NTRS and IBM process information.

a circuit's maximum operating frequency (f_{op}) and operating power consumption (P_{op})[3; 4; 5; 6]. Most applications have a power limit (P_{max}) dictated by either the power available from the supply or by a heat dissipation limit. Any chips with $P_{op} > P_{max}$ can not be sold and are considered yield loss. Chips that meet the power requirement must also meet a target minimum frequency (f_{min}). Many manufacturers bin their chips based on f_{op}, but there is still an f_{min} such that any chips with $f_{op} < f_{min}$ are discarded. Process variations tend to cause a widening of the distribution of f_{op} and P_{op} from a manufacturing process. With a wider distribution of circuit operating parameters, more chips fall outside of the acceptable region defined by f_{min} and P_{max}. In this way, process variations cause a reduction in manufacturing yields due to the operating parameter changes they cause. With increasing relative process variation magnitude, and thus larger changes in P_{op} and f_{op}, yields from manufacturing processes will decrease as technology scales down. The incredible burden of improving production yields needs to be addressed at every step from design, to fabrication, to post-fabrication test.

This thesis presents a new approach to adaptive body biasing in order to mitigate the effects of process variations and improve manufacturing yields. It addresses process variations on a per well basis, allowing it to address within-die variations as well as larger scale variations. Individual wells can have a predetermined, locally generated bias or a chip-wide adjustable bias. Also included in the scheme is adjustable power supply voltages. The approach requires very little overhead in chip silicon area, signal routing and post-fabrication test time. This discussion mainly focuses on the method as it applies to microprocessors but can theoretically be applied to any circuit. Based on small circuit simulations, the method is able to improve an initial yield of 12% to nearly 80%.

Chapter 2

Existing Work

Post-fabrication adjustment of supply voltages and bias voltages has shown to be effective at reducing the effects of process variations by reducing the spread of operating parameters. This tightening of the P_{op} and f_{op} distributions directly improves manufacturing yields.

One post-fabrication adjustment technique is the use of supply voltage scaling to change the operating parameters of microprocessors. Adjusting the supply voltage for a circuit trades power consumption for performance. This technique is used to maximize performance for a given power limit. The simplest method of power supply scaling is to statically use two supply voltages. A 47% reduction in P_{op} was realized in a media processor using this technique in [7]. [8] uses two power supply voltages in an MPEG4 decoder to reduce P_{op}. A higher supply voltage is applied to critical paths so throughput is not affected, while non-critical paths use a lower supply voltage in order to reduce P_{op}. A method of optimally grouping gates into voltage groups is presented in [9].

Since most current designs are power limited, it is desirable to dynamically adjust the supply voltage based on the work load present. This dynamic voltage scaling (DVS) can be an effective way of reducing dynamic power consumption with only slight impact on performance and throughput. It can be used to minimize P_{op} over an extended time based on the activity of the circuit. Dual dynamic supply voltages are used to reduce P_{op} of an MPEG4 codec chip in [10]. [11] and [12] show significant power savings by applying DVS to microprocessor systems, and similar results are shown for smaller circuits in [13; 8].

Another method of post-fabrication adjustment is using body effect to change the threshold voltage (V_{th}) of transistors. As CMOS processes scale smaller, V_{th} of devices is also reduced. Shorter gates and lower V_{th} produces an exponential increase in static power. It is shown in section 3.1.3 that DVS is less effective at reducing static power than dynamic power. Instead, the transistor body bias (V_b) can be adjusted to change f_{op} and P_{op}. Just like DVS, an adaptive body bias (ABB) can be used to change the operating parameters of a circuit over time. The idea was discussed in

[14] and [15] as a way of adjusting circuit performance. ABB has been applied in microprocessors to reduce standby power consumption in [16], while [17] and [18] used ABB to improve yield by addressing P_{op} and f_{op}. [19] also presents an ABB scheme which addresses manufacturing yield. It uses a externally generated V_{b_n} and a locally generated V_{b_p} based on P_{op} and single critical path f_{op} measurements.

Since both DVS and ABB can adjust the operating parameters of a chip during runtime, their adjustments are based on actual operating parameters of an individual chip. This allows them to lessen the effect process variations have on these operating parameters. However, since DVS and ABB must be applied uniformly to an entire chip, they can only address die-to-die (D2D) and lot-to-lot (L2L) variations. Unfortunately, within-die (WID) variations are becoming more significant than D2D and L2L which makes DVS and ABB less effective[3].

In order to address WID variations, the ideas of voltage scaling and body biasing need to be applied to smaller, more localized areas in a chip. [19] presents a method of applying ABB to small circuit blocks within a chip based on a local determination of the required biasing. While the results of this scheme are impressive, there are several major draw backs to the method. To be truly effective, the method requires a very expensive triple well process so that both nfet body bias (V_{b_n}) and pfet body biases (V_{b_p}) can be locally adjusted. It also requires a replica critical path, phase detector, and counter at each area of interest across the die. A central PVT-dependent reference voltage is required to enable local optimal bias generation at each area. Each local bias generation requires a R-2R ladder D/A converter which controls an op-amp at each area of interest. All these additions present an enormous increase in chip area. However, the additions allow for dynamic adjustments of biases based on locally dynamic conditions such as V_{dd} and temperature. The spatial granularity of WID variations, including the locally dynamic conditions, addressed by this method is limited by the size of all this additional circuitry. That is, variations that occur within the true critical path but not in the replica critical path cannot be corrected. Also, the location of the replica critical paths across the die would strongly influence the schemes effectiveness. Optimal placement might be hindered by layout blockages such as IP blocks or continuous cache regions. Overall, the WID ABB scheme from [19] is prohibitively expensive and difficult to implement on a microprocessor. The scheme presented in this thesis is fundamentally based on [19], but addresses variations on the well level. It also requires much less overhead in terms of both silicon area and routing resources.

Chapter 3

Background

3.1 Process variations

Process variations in semiconductor manufacturing is when process parameters deviate from their ideal designed values. They can be classified into four basic tiers: lot-to-lot variations, wafer-to-wafer variations, die-to-die (D2D) variations, and within-die (WID) variations. Part of each tier of variations can be attributed to systematic and random variations. All of these variations change the operating parameters P_{op} and f_{op} of the final circuit. The following sections show some sources of process variations, their effects on circuits, methods of mitigating these effects and a technique for modeling variations.

3.1.1 Sources of process variations

There are several design parameters specified for the creation of a single transistor, wire or any other component. This research focuses mainly on variations to transistor design parameters. Unavoidable inconsistencies in the manufacturing process make every design parameter a random variable. For the creation of a single transistor, the designer must specify a gate width (W_d) and length (L_d). These are the basic parameters defining the transistor. Figure 3.1 shows some of these design parameters. The process used to manufacture the circuit defines several other parameters including gate oxide thickness (t_{ox}), channel doping density (N_A), and junction depths. This is still just a small subset of all the parameters used to specify a single transistor. The designer also must specify drain and source areas and shapes, drain-source separations, and special implantations such as lightly doped drain or HALO structures. In more advanced processes, the designer can also specify to use a high or low V_{th} device. These two devices are usually accomplished by varying N_A. Each geometry specified requires a lithographic process to manufacture. Any inconsistency in any step of the lithographic process causes deviation from the specifications of the design.

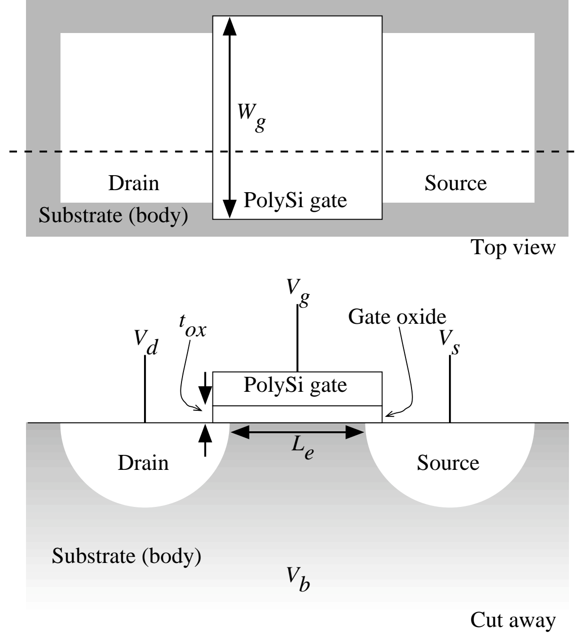


Figure 3.1: Basic top view and cut away schematic representation of a MOS transistor showing basic dimensions.

One source of random variations is the lithographic process step that creates patterns using a photoresist. In this step, the entire wafer is coated with photoresist, exposed to the desired pattern, and then developed. The development cycle washes away exposed photoresist leaving behind the pattern of the exposure mask (for positive-type photoresist). The precision of this entire process is crucial since it forms nearly every dimension in the circuit. Slight changes in the development cycle can lead to random fluctuations from wafer to wafer. These changes can be caused by inconsistencies in the photoresist, slight changes in the developer solution concentration, and differences in post-exposure bake temperatures. As shown in Figure 3.2, under or incomplete development can lead to smaller geometries than designed, while over development can cause larger or even inadvertently joined geometries. WID variations can occur due to micro-depletion of the developer caused by dense areas of exposed resist. Also, the critical dimension control can be randomly affected by edge roughness on the developed lines since the molecular size of the resist is on the order of several nanometers[2].

An example of systematic and random process variations comes from lenses used in the lithographic process. In sub-wavelength lithography, the smallest theoretical resolution in an imaging system is the diffraction limit defined by the Rayleigh scaling equation

$$W_{min} = k_1 \times \lambda / NA \tag{3.1}$$

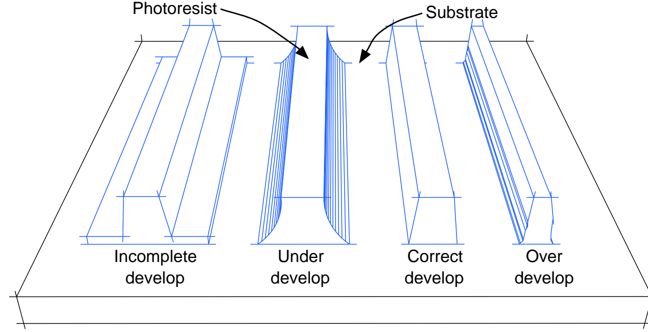


Figure 3.2: Exaggerated example of possible process variations caused by inconsistencies in photoresist development.

where W_{min} is the minimum resolved line width, λ is the wavelength of the light source, NA is the numerical aperture of the optics in the projection system, and k_1 is the dimensionless Rayleigh constant of order unity. As process generations advance, reduction of W_{min} has focused on reducing k_1 of projection systems, while incrementally reducing λ by changing the light source. Advanced resolution enhancement techniques like phase-shift masks, optical proximity correction, off-axis illumination and sub-resolution assist features have shifted the limiting component in reducing k_1 from the mask to the lens. Lens aberrations, both due to lens design and, more significantly, manufacturing, are quickly becoming an important factor in k_1[20].

A simple reducing projector is shown in Figure 3.3 with several light rays. The optical path length of a ray is the distance along the ray times the local index of refraction. In a simple system with ideal lenses, the optical path length is the same for each ray. One can say the optical path difference (OPD) for any ray is zero. The OPD is defined as the difference in optical path length of a ray from the optical path length of a reference ray passing through the center of the aperture stop. Any variation, imperfection or distortion in the lenses can cause the OPD for the effected area to be non-zero. This is modeled as surface at the aperture stop, and usually has a general bowl shape with areas further from the center experiencing a larger OPD[1; 20; 21].

Problems with the lenses have many sources, but in general OPD can be caused by lens heating, auto-focus errors, leveling errors, wafer non-flatness, etc. The effects of all the aberrations are pattern-independent shifts of the image, degradation of the image, orientation dependent shift of focus, image asymmetry, pattern-dependent image shifts, image anomalies, and pattern-dependent focus shifts[20]. All of these effects result in small variations from the specified design parameters for each geometry.

Random variations also come from dopant concentration fluctuations. In sub-100nm technologies,

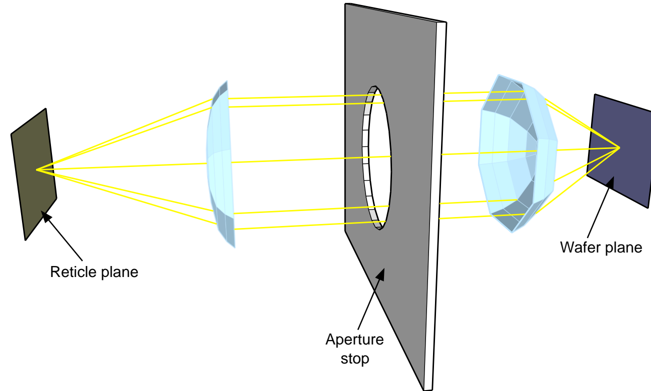
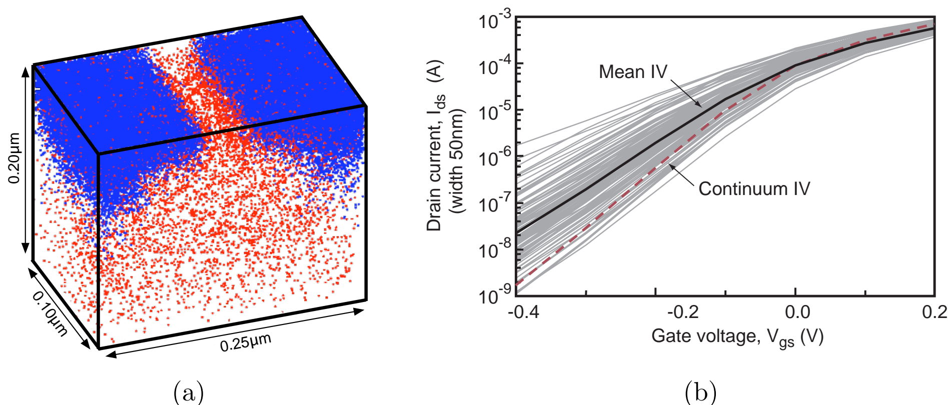


Figure 3.3: Diagram of a simple aberration-free ($OPD = 0$) reducing projector.



(a) (b)

Figure 3.4: (a) Random dopant atom placement from a three-dimensional Monte Carlo simulation. (b) IV characteristics from 100 random dopant placements.[22]

channel dopant concentrations can be on the order of several hundred atoms, and is scaling roughly as $L_{\mathit{eff}}^{1.5}$[22]. The wafer-wide dopant concentration is very tightly controlled in the ion implantation process. However, due to this atomic quantization, the placement of these dopants in the fet channels can have an effect on the V_{th} of a fet. This phenomenon has been extensively studied with two- and three-dimensional Monte Carlo simulations and field solvers[22; 23]. Figure 3.4a shows an example of the dopant placements from one of these simulations. Figure 3.4b shows the IV curves from 100 random dopant placements. This variation in V_{th} can be substantial in current processes, and is getting worse as technologies scale below 100nm.

There are several other sources of process variation, each of which adds a slight change to every parameter. Although each source of variation causes only small changes, the over all critical dimension (usually associated with L_{eff}) can be expected to vary ±10–20% in a mature manufacturing process[3; 4]. As processes continue to shrink the critical dimension, the relative magnitude of the

process variation is getting worse.

3.1.2 Effects of process variations

In normal operation, transistors in digital circuits, such as microprocessors, are usually off or in saturation. Therefore, one major factor in determining the performance of a circuit is the drain current in saturation[24]. Since the saturation current is responsible for charging the interconnect and gate capacitance of the next nodes, it is clear that performance is proportional to drain current (I_{ds}). One model for the drain current is given by:

$$I_{ds} = \frac{\mu C_{ox}}{2} \frac{W}{L_{eff}} (V_{gs} - V_{th})^2 [1 + \lambda (V_{dg} - V_{th})] \tag{3.2}$$

where μC_{ox} is the conductance parameter of the device, V_{gs} is the gate to source voltage, λ is the channel length modulation parameter which is inversely proportional to L_{eff}, and V_{dg} is the drain to gate voltage. Equation 3.2 is only accurate for long channel devices ($L_{eff} > 1\mu$m) but it is commonly used as a reference for general transistor operation. It is easy to see that I_{ds} is inversely proportional to L_{eff}, and thus, circuit delay is proportional to L_{eff}. Variation in L_{eff} impacts circuit performance by changing I_{ds}. Similarly, variations in V_{th} changes I_{ds} and thus effects circuit performance. V_{th} fluctuations are caused by variations in the substrate dopant concentrations, as well as variations in L_{eff} which cause V_{th} roll-off due to short-channel effects. In sub-100nm technologies, V_{th} variations are dominated by the latter[25; 26].

A second model of circuit performance is given by the alpha-power model which gives an estimate of the delay of a basic CMOS inverter[27]. The delay of complex gates is proportional to the delay of a basic inverter, and thus circuit delay can be modeled using

$$t = \frac{d_L k}{(V_{dd} - V_{th})^\alpha} \tag{3.3}$$

where d_L is the logic depth of the path, k is a process dependent constant and α is a measure of the velocity saturation. As stated before, short-channel effects make V_{th} proportional to L_{eff}. Thus, circuit delay is proportional to L_{eff}.

Total circuit power consumption, consisting of static (leakage) power and dynamic (switching) power, must also be considered. First, static power is caused by

- leakage current through the gate via direct tunneling (I_{gate})
- sub-threshold conduction between drain and source (I_{off})
- drain and source to body junction leakage ($I_{db/sb}$)

Process variations mainly effect I_{off} since it depends strongly on L_{eff} and V_{th} as shown by

$$I_{off} = \frac{W}{L_{eff}} I_S \left[1 - e^{\frac{-V_{dd}}{V_T}}\right] e^{\frac{-(V_{th}+V_{off})}{nV_T}} \tag{3.4}$$

where I_S, n and V_{off} are empirically derived process constants and V_T is the thermal voltage [28]. Clearly, I_{off} is strongly dependent on L_{eff} and V_{th}, which is dependent on L_{eff} itself. Therefore, process variations that change L_{eff} have a large impact on the static power consumption of the circuit due to this I_{off} component.

The other two components of static power are less dependent on L_{eff}, but can be affected by other process variations. For example, I_{gate} is exponential with respect to t_{ox}, and $I_{db/sb}$ is affected by N_A. However, these variations are usually less dramatic than L_{eff} and V_{th}[2].

Dynamic power consumption has almost no direct dependence on L_{eff} and V_{th} process variations. It is modeled by

$$P_{dyn} = C \cdot V_{dd}^2 \cdot f_{op} \cdot \alpha \tag{3.5}$$

where C is the switched capacitance and α is the activity factor. Since process variations change f_{op}, the dynamic power consumption is also reduced. This can be viewed as part of the power performance trade-off, and it can be a valuable tool in adjusting overall power consumption as the next section shows. Variations in L_{eff} also change the gate oxide capacitance by changing the area of the gate. This increase is linear with respect to L_{eff} whereas f_{op} is super-linear.

Overall, process variations cause chips to exhibit a distribution of P_{op} and f_{op}. Figure 3.5 shows an example of such a distribution from a 32-bit static CMOS ripple adder caused by 5nm 3σ L_{eff} varaitions on a 100nm commercial process technology. There is about a $\pm 7\%$ spread in f_{op} from the average, while P_{op} varies from 17% above to 7% below the average. This wide distribution causes manufacturing yields to depend on the P_{max} and f_{min} chosen. If both target operating parameters are chosen to be the average values, the yield would be only 12%. 42% of the chips are above the average P_{op}. This means that a product binning scheme which only threw out high power chips would still have a 58% yield loss.

3.1.3 Mitigating effects of process variation

As stated in Chapter 2, adjusting the supply voltage and transistor body voltage can be used to reduce the effects of process variations. Changing the operating parameters of a circuit by adjusting the supply voltage and body bias can compress the distribution of P_{op} and f_{op} shown in Figure 3.5.

Equation 3.2 gives a good idea of how V_{dd} scaling methods affect circuit performance. For nfets, reducing V_{dd} will reduce V_{gs} and thus quadratically reduce I_{ds} and the circuit performance. V_{dg} and

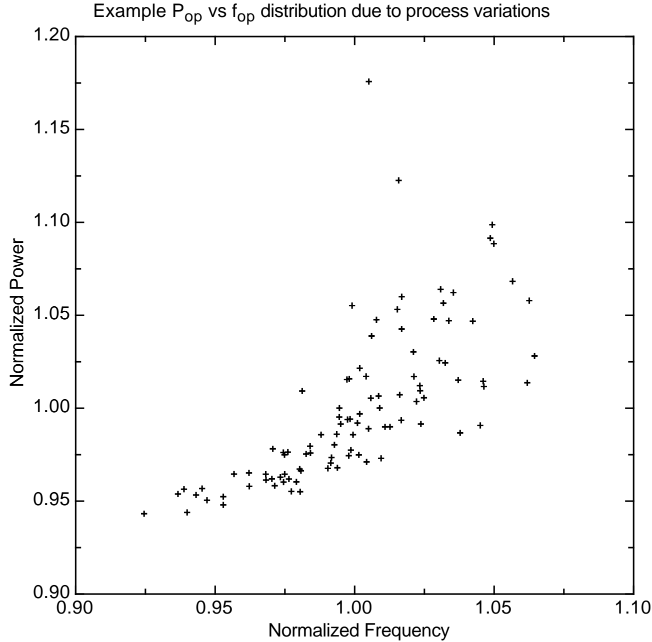


Figure 3.5: Example of P_{op} and f_{op} for 100 32-bit static CMOS ripple adders on a 100nm process technology.

λ also depend on V_{dd}. Overall, scaling V_{dd} has almost a cubic relationship to circuit performance[11]. Scaling V_{dd} can be an effective tool in changing the power consumption of a circuit as well. Equation 3.5 shows that reducing V_{dd} quadratically reduces P_{dyn}. This is compounded by the reduction in f_{op} caused by reducing V_{dd}. All three components of static power are reduced by lowering V_{dd}. I_{gate} and $I_{db/sb}$ are reduced since they are proportional to the voltage across the gate and the drain/source to body voltages, respectively. I_{off} is exponentially dependent on V_{dd} as shown in equation 3.4.

Adaptive body biasing relies on body-effect's ability to change transistor V_{th} in order to adjust circuit operating parameters. The equation

$$V_{th} = V_{th0} + \gamma_n(\sqrt{V_{sb} + 2\phi_F} - \sqrt{2\phi_F}) \tag{3.6}$$

where V_{sb} is the source to body voltage, V_{th0} is the threshold voltage when $V_{sb} = 0\text{V}$, γ_n is the nfet body effect parameter, and $2\phi_F$ is surface potential, shows how V_{th} is affected by the body bias for nfets[25]. Combining equation 3.6 and equation 3.2 shows how I_{ds}, and thus circuit performance, is proportional to the body bias. A positive V_{sb} is referred to as a forward body bias, and a negative V_{sb} is a reverse body bias.[1] The same trend can be found with equation 3.3. Body bias has little

[1]Since the source of nfets is usually tied to ground, V_{sb} can be referred to as V_{b_n}. Similarly, the body voltage of pfets can be referred to V_{b_p}.

effect on dynamic power besides the indirect relationship caused by changing f_{op}. For static power, body biasing can dramatically change I_{off}. Equation 3.4 combined with equation 3.6 shows the exponential relationship between I_{off} and body bias.

Adaptively adjusting both the supply voltage and the body bias can dramatically change the operating parameters of circuits. If done in a way that reduces the spread of the operating parameter distributions, manufacturing yields can be improved substantially.

3.1.4 Modeling process variations

In a manufacturing process, every dimension and parameter defined in the design process becomes a random variable. To

simplify, only the variability of transistor gate length was modeled since it has the most pronounced effect on circuit operating properties. Due to the short-channel effect, variations on channel length will also cause V_{th} to fluctuate as shown in Section 3.1.2. The overall distribution of transistor lengths upon variation should be a p-variate normal distribution $L' = N_p(\mu = L, \Sigma)$ with a mean $\mu = L \in \Re^p$ and a covariance $\Sigma > 0 \in \Re^{p \times p}$, where p is the number of transistors and L is the designed transistor length. Variations should also be correlated geographically across a chip. That is, transistors located near one another should have highly correlated variations. This correlation can be represented in a correlation matrix $\rho \in \Re^{p \times p}$, where each entry $0 \leq \rho_{i,j} \leq 1$ is the correlation between transistor i and j. Since the variance of all the transistors was assumed to be equal, ρ has an equal structure to the covariance matrix Σ . By partitioning the normal along this structure and applying a linear transformation to each part, this distribution can be rewritten as:

$$L' = N_p(\mu, \Sigma) = L + \sigma \cdot N_p(\vec{0}, \rho) \tag{3.7}$$

where σ is the standard deviation of all the transistors. The probability density function of this new distribution can be expressed as:

$$pdf(N_p(\vec{0}, \rho)) = |2\pi^p \rho|^{-1/2} \exp\left\{-\frac{1}{2} x^\top \rho^{-1} x\right\} \tag{3.8}$$

where $x \in \Re^p$ [29].

Take note that the inverse of ρ must be computed to produce such a distribution. In order to guarantee ρ^{-1} exists, ρ must be positive definite. Unfortunately, there is no known way to produce an arbitrarily sized matrix that is positive definite by construction, so the values in ρ must be chosen by trial and error.

3.2 Search algorithms

In most modern engineering optimization tasks, multiple parameters contribute to the optimization objective values. Generally, there are also constraints on the parameters. A general form for an optimization problem is

$$\begin{aligned} &\textit{minimize:} && g(\vec{o}(\vec{x})) \\ &\textit{subject to:} && \vec{x} \geq \vec{0} \end{aligned}$$

where g is a weighting or objective function for the objective vector $\vec{o}$, and $\vec{x}$ is the parameter vector.[2] The length of $\vec{o}$ is the dimension of the objective space, and the length of $\vec{x}$ is the dimension of the parameter space (n). Assuming nonlinear interaction between parameters (epistasis), the size of the search space is m^n where m is the number of different values each parameter can take. If n and m are small, the search space of the optimization problem is small. In this case optimization can be done by enumerating every possible combination of parameters. In fact, one would expect to find the solution after only $m^n/2$ evaluations. The probability of finding the optimal solution after k evaluations is k/m^n. However, since there is no way of knowing when the optimal solution is found, one must evaluate all possibilities to prove optimality. This quickly becomes a massive problem as n or m increases, and adding constraints can make traversing the search space more difficult.

For example, consider the problem where $\vec{x}$ represents 10 binary values ($m = 2$, $n = 10$) where there are no other parameter constraints. If the objective function takes one second to compute at each point, an exhaustive search of this space would take about 17 minutes. However, if we triple the number of parameters, the search space grows exponentially to 2^{30}. It would take 34 years to exhaustively search this space. Unfortunately, a search space of 2^{30} is rather small in modern engineering problems, and product cycles are significantly shorter than 34 years. This presents a dilemma.

There are two obvious solutions to the dilemma:

1. Do not exhaustively search every possible combination of parameters.

2. Settle for a good solution instead of the absolute optimum.

Taken together, these two points can significantly reduce the amount of time it takes to find a solution to an optimization problem. However, the points bring up two problems. First, how does one traverse

[2] To ease discussion, it is assumed that optimizations are always of the minimization type. Any maximization problem can be restated as a minimization problem.

the search space in order to find a good solution? Second, what is a good solution? To address the first problem, there are several alternative methods to searching large spaces, but this thesis only covers a few. Section 3.2.1 and 3.2.2 address random searches (e.g. random walk), simulated annealing and gradient based searches (e.g. steepest descent). Section 3.2.3 covers evolutionary algorithms. The second problem can be a bit more difficult. The easy answer is that a good solution is one that meets a predefined criteria for the problem. This requires a priori information about the objective space and an executive decision defining the criteria for an acceptable solution. However, many times this a priori knowledge is hard to come by, and the a priori decisions can be biased and poorly made. The field of multiple objective optimization brings the solution by removing the objective function g and finding a set of equally good solutions. Section 3.2.3.2 gives a brief introduction to multiple objective optimization with evolutionary algorithms, specifically the SPEA2 and NSGA2 algorithms. Finally, a note on the use of heuristics in search algorithms is in section 3.2.4.

3.2.1 Random walks, hill climbing and simulated annealing algorithms

A random walk is the simplest way to traverse a large search space. It is even simpler than an exhaustive search since no effort needs to be made to organize the search in order to reach every possible point. However, it will generally take more evaluations to find the optimal solution. For parameter spaces in which it is difficult to organize an exhaustive search, this decrease in search speed might be acceptable since it might be rather expensive (either computationally or functionally) to compute the next step for an exhaustive search.

Figure 3.6 shows the basic flow of a random walk algorithm. The search works in steps where $\vec{x}^{(i)}$ is the parameter vector at step i. The algorithm is usually initialized with a random parameter vector $\vec{x}^{(1)}$. After initialization, each step performs a random change to one or more of the parameters giving $\vec{x}^{(i)} \rightarrow \vec{x}^{(i+1)}$. If $g(\vec{o}(\vec{x}^{(i)})) \leq g(\vec{o}(\vec{x}^{(best)}))$, the step is saved as the best found configuration $\vec{x}^{(best)}$. The search then continues from $\vec{x}^{(i)}$. This simple algorithm can be very effective at walking to a good solution. The probability of finding the optimum after k evaluations is $1 - \left(\frac{m^n-1}{m^n}\right)^k$ as the set of all steps $\{\vec{x}^{(1)}, \vec{x}^{(2)}, ..., \vec{x}^k\}$ represents a random sample of size k. As k increases, the probability that an optimal solution has been reached increases.

In order to improve the speed of search, a more localized or neighborhood search can be implemented. In the random walk algorithm, each step creates a new parameter vector randomly. This new step may not be near the previous step in either the parameter or objective space. It is often the case that there is a smoothness of the search space meaning that good parameter vectors will

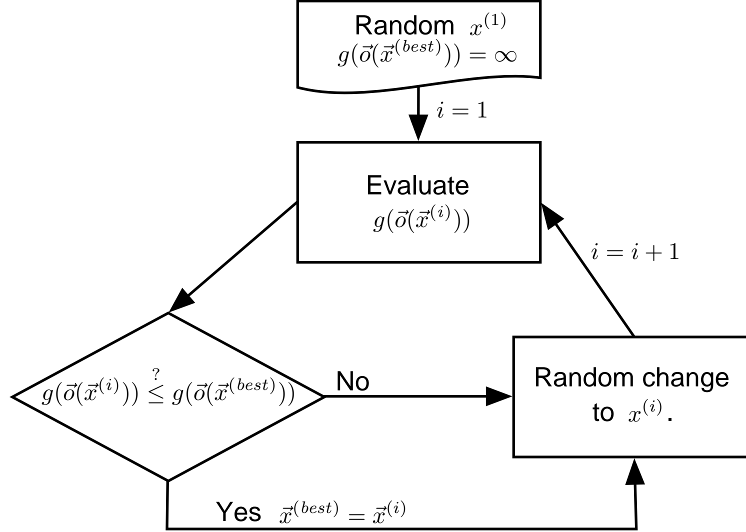


Figure 3.6: Flow chart of a random walk algorithm.

tend to be near other good parameter vectors. By exploiting this smoothness, an algorithm can make a small change in a parameter vector and expect only a small change in the objective vector. By modifying the random walk search to only make a small random change to the parameter vector for each step, the search walks to a neighbor in the parameter space. This is termed a local random walk. This type of search can be even more useful if the search is also allowed to step back. If the objective value at step $i+1$ is worse than that at step i, the algorithm goes back to $x^{(i)}$ and iterates again. This type of algorithm is called a local hill climbing search[30], and its flow chart is in Figure 3.7.

One problem with the local hill climbing algorithm is that it can get stuck in local minima or plateaus in the objective space. This is due to the fact that the optimal solution is not necessarily in the neighborhood of the initial random parameter vector, $\vec{x}^{(0)}$. One way around this problem is to restart the algorithm with a random parameter vector when it converges or stops improving the objective[31]. The restart allows the algorithm to explore a different neighborhood in order to possibly find a better solution. This multi-start local hill climb algorithm, if allowed enough restarts, will always eventually find the optimal solution since it reduces to a exhaustive random search[32].

Another way to allow the algorithm to explore other neighborhoods is called simulated annealing[33]. Basically, the algorithm has a probability to keep a step that is worse than the previous step. The probability of accepting step i is usually:

$$P(accept\ \ i) = \begin{cases} 1 & \text{if } g(\vec{o}(\vec{x}^{(i)})) \leq g(\vec{o}(\vec{x}^{(i-1)})) \\ e^{T(i)\cdot\left(g(\vec{o}(\vec{x}^{(i-1)}))-g(\vec{o}(\vec{x}^{(i)}))\right)} & \text{if } g(\vec{o}(\vec{x}^{(i)})) > g(\vec{o}(\vec{x}^{(i-1)})) \end{cases} \tag{3.9}$$

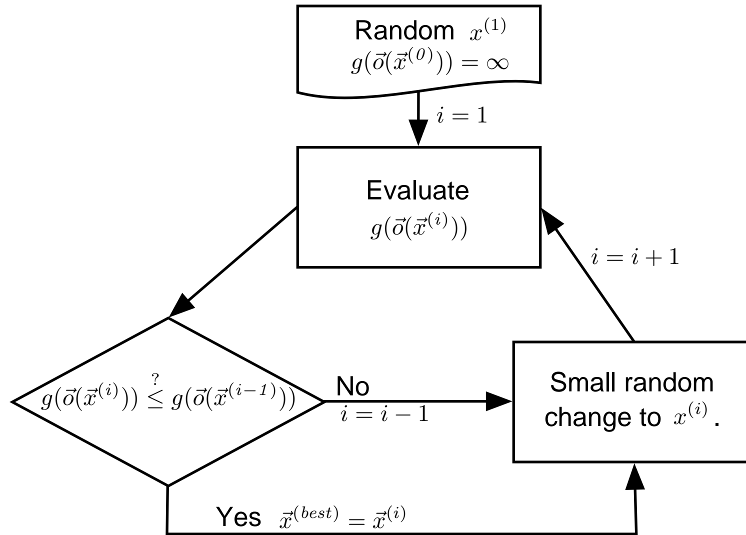


Figure 3.7: Flow chart of a local hill climbing search algorithm.

where $T(i)$ is usually referred to as the temperature of the annealing and is a function of the step number i. $T(0)$ is normally high, so that there is a good chance that worse solutions are accepted at the beginning of the search. As the search progresses, $T(i)$ decays making it less likely that a bad solution will be taken.

All of these algorithms only require a smoothness in the parameter space. They do not require any knowledge of the objective function or its derivatives. This is very advantageous since this information is not always available. However, they are generally slow to converge if $\vec{x}^{(0)}$ is not close to the optimal solution. To make an algorithm approach minima more rapidly, more information about the objective space is needed.

3.2.2 Gradient based search algorithms

Utilizing gradient information in a search can provide an enormous speed improvement since the algorithm knows which way to go in order to improve fitness. Some gradient search algorithms can use both the first and second derivatives in order to speed convergence. Using the first and second derivatives allows an algorithm to know what direction and approximately how far to go. The simplest gradient search is the steepest descent or gradient descent algorithm. Like the random walk and the local hill climber, steepest descent walks to the optimal point in steps. Each parameter-space step is based on

$$\vec{x}^{(i+1)} = \vec{x}^{(i)} - \delta(i)\hat{s}^{(i)}$$
$$\vec{s}^{(i)} = \nabla g(\vec{o}(\vec{x}^{(i)}))$$

where i is the iteration number, $\delta(i)$ is an iteration dependent step size parameter, and $\hat{s}^{(i)}$ is the unit vector of $\vec{s}^{(i)}$ which is the gradient of the objective function. $\hat{s}^{(i)}$ points in the direction of the steepest increase of the objective function, so with each iteration the algorithm steps in the $-\vec{s}^{(i)}$ direction. This is much more efficient than the random walk and the local hill climber because the steepest descent algorithm can always move in the direction that improves the fitness the most. The choice of $\delta(i)$ is vital since the algorithm can easily overstep the optimal point even though it moved in the right direction, and there are several methods to determining $\delta(i)$.

Obtaining the gradient information may be difficult or impossible. If the objective function is not known or is not differentiable, its gradient needs to be estimated at each iteration by evaluating two points for each parameter and calculating the slope for each. This can be very expensive, so the benefit of the gradient search needs to be very high to make it worthwhile.

In general, gradient based algorithms can converge much faster than local search or random walks. The algorithms can still get stuck in local minima and plateaus. They also require more memory than the other walking algorithms in order to store the gradient information for each parameter. Conforming to the "No free lunch" theorm[34], gradient algorithms can be very powerful on the right problems, and powerful time wasters when applied to the wrong problems.

3.2.3 Evolutionary algorithms

Evolutionary algorithms (EA) are based on a computational model of biological evolution[35]. The optimization parameters are encoded in a chromosome-like structure which can be recombined and mutated. This is usually referred to as an individual or chromosome. Each individual has a fitness which is related to the problem's objective space.[3] EAs evolve a population of individuals by selecting objectively-better individuals to produce the next generation. The flow of a basic EA is shown in Figure 3.8. Usually the EA is initialized with a population of random individuals. Individuals are selected from that population to produce a parent population. Then individuals in the parent population go through the process of crossover and mutation which is modeled after biological mating. This produces a child population from which new individuals are selected to be inserted in the general population. This process is repeated for each generation and slowly evolves the population towards an optimal solution. Specifics of how the individuals are assigned a fitness, and how the processes of selection, crossover, mutation and reinsertion are performed varies dramatically. The next two sections will detail the basis of single and multiple objective EAs used in Chapter 4.

---

[3]The normal biological evolution model favors increasing fitness. For consistency with the previous discussion, fitness will be minimized by the EAs.

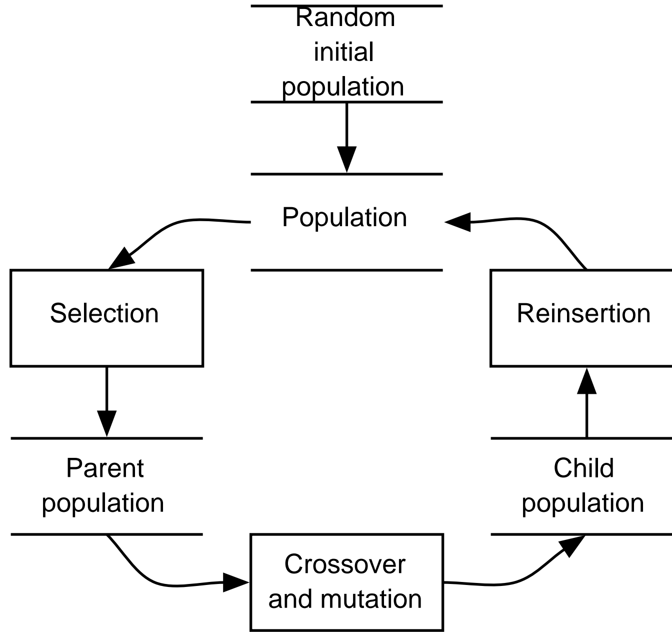


Figure 3.8: Flow chart of a basic evolutionary algorithm.

3.2.3.1 Single-objective evolutionary optimization

The primary feature of a single objective EA (SOEA) is the use of a single fitness value to compare individuals. The fitness value is used to rank individuals in the selection process and to choose which children get reinserted into the general population. If the problem has a multidimensional objective space, it must be reduced to a single dimension. A simple weighting function can be used to do this:

$$\mathit{fitness} = g(\vec{o}(\vec{x})) = \sum_{j=1}^{n} w_i f_i$$

where $\vec{w}$ is a vector of weights such that $\sum_{j=1}^{i} w_j = 1$, n is the dimensionality of the objective space and $\vec{o}$ is the objective function which maps the parameter space $\vec{x}$ onto the objective space. This objective weighting is sometimes referred to as vector aggregation or weighted sum SOEA. The draw back of this method is its requirement of a priori knowledge of the correct weighting to use. This can introduce bias into the optimization and lead to suboptimal solutions in the multiple dimension object space.

Parent selection can be based on this single fitness value. One common selection method is tournament selection[36; 37] because of its simplicity and speed. Tournament selection works by randomly choosing two individuals from the general population. A tournament is held in which the winner is the individual with the smaller fitness. The winner of the tournament is added to the parent population, and will reproduce via crossover. From a biological stand point, this is analogous

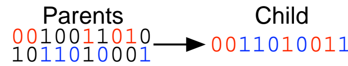


Figure 3.9: Example of uniform crossover for a 10 bit chromosome. Colored bits represent bits chosen to build the child chromosome.

to Darwinistic selection in which more fit individuals have a higher chance of reproducing. The tournament selection is repeated until the parent population reaches a predetermined size.

The size of the parent population can play an important role in the convergence of the SOEA based on the different crossover techniques used. The Genitor model SOEA[38] is popular when the objective function is computationally expensive since it uses only two parents to create a single child for each generation. [38] shows how this method is still able to apply sufficient evolutionary pressure on the population. Other methods use larger parent populations which produce larger child populations.

The crossover mechanism is what drives the SOEA. It is responsible for producing new individuals which, presumably, are at least as good as the parents. Crossover methods vary based on the form of the parameter space. For binary parameters, the simplest crossover is uniform crossover[39]. Figure 3.9 shows an example of uniform crossover on a 10 bit binary chromosome. Uniform crossover is performed by randomly selecting a bit from the same position in one of the two parents for each position in the binary string. This system is appealing because each bit is inherited by the child independently of any other bit. It is also unbiased with respect to the representation of the chromosome (e.g. the bit ordering or the starting and ending bits).

For real valued parameters, a very effective method of crossover is intermediate crossover[40]. This method assigns the child a random parameter value somewhere between the parent parameters. A slight modification to this is allowing the child to take values just outside the range of the parents.

Mutation is key to good search space exploration and preventing premature convergence in an EA. It basically makes small random changes to children before they are added to the population. This is easily done for both binary and real coded parameters. Mutation helps to maintain genetic diversity in the population by adding characteristics to individuals that have been lost from, or were never present in the population. A mutation rate, the probability that an individual is mutated, of $1/n$ is very common, where n is the population size[40; 41; 42].

Finally, the children need to be reinserted into the general population. One simple way to reinsert the children is to replace the parents by two of their children. However, this can be detrimental to the population since the parents were originally selected because they were relatively strong, and

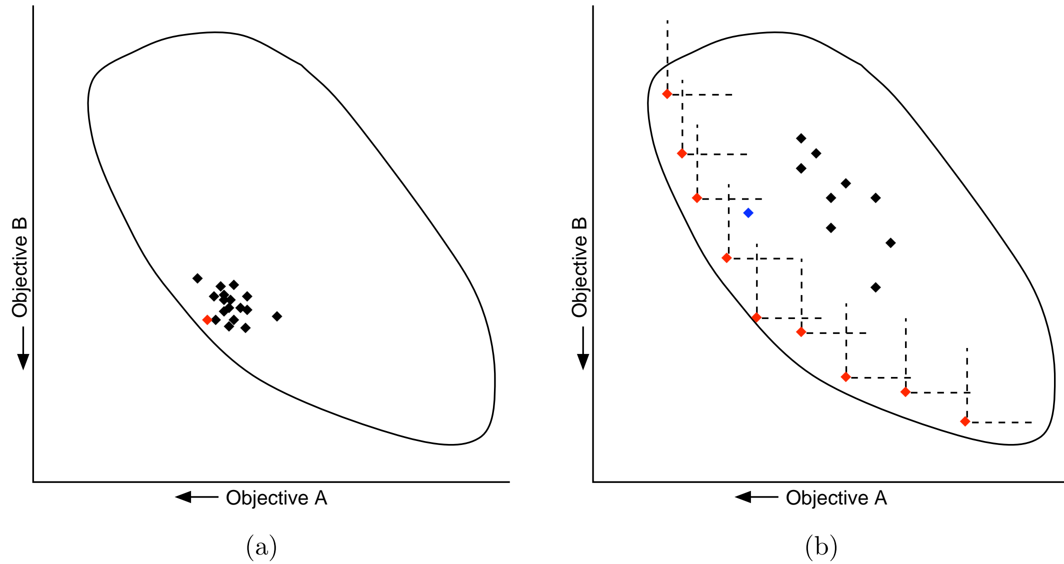


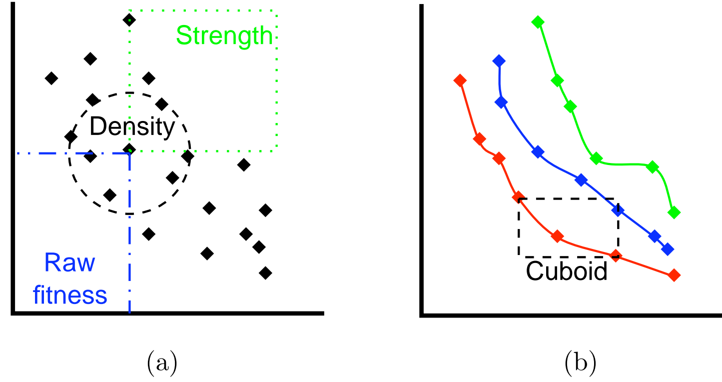
Figure 3.10: (a) Illustration of a SOEA population after many generations in a two dimensional objective space (minimization of both objectives). (b) Dominance and pareto optimality in a two dimensional objective space (minimization of all objectives).

there is no guarantee that the children are better than their parents. From this stand point, a better approach is to replace the weakest individuals in the population with the children. This is also closer to the biological basis of the EA in which better fit individuals reproduce while weaker individuals die off.

The solution provided by a SOEA is simply the individual with the smallest fitness. This best individual's fitness is dependent on the weighting vector used and may be suboptimal in the problem's objective space if the weighting vector was poorly chosen. Also, as more generations pass, the population tends to cluster. Figure 3.10a shows this clustering in a two dimensional objective space. The region bounded by the solid line represents the feasible region of the problem. The red point is the best solution found, and the black points are other members of the population. The clustering of the black points near the red can be detrimental to the overall quality of the solution since the search space is not aggressively searched due to lack of diversity in the population.

3.2.3.2 Multiple-objective evolutionary optimization

In contrast to the SOEA, multiple objective EAs (MOEA) do not directly reduce the dimensionality of the objective space in order to assign fitness by use of a weighting function. MOEAs maintain the essence of the objective space dimensionality by use of the concept of pareto optimality and dominance. An individual dominates another if it is better in at least one objective dimension but not worse in any other. The pareto optimality concept yields that there is no best individual,



(a) (b)

Figure 3.11: (a) Illustration of (a) SPEA2 and (b) NSGA2 fitness calculations.

but some individuals are better than others. An individual is said to be non-dominated or pareto optimal if there is no known solution that dominates it. Figure 3.10b illustrates dominance and pareto optimality in a two dimensional objective space. Again, the region bounded by the solid line represents the feasible region of the problem. All red points are pareto optimal with their regions of dominance shown by the dashed lines. As a group, the red points form a pareto optimal front which approximates the edge of the feasible region. This front represents the edge of the explored search space that contains equally good solutions. The blue point is dominated by only one point, and all black points are dominated by multiple other points.

Even though no weighting function is used to reduce the objective dimensionality, MOEAs do need to be able to compare individuals. This is usually done with a fitness value based on the dominance of an individual. In order to stem the clustering of solutions found in SOEA, the fitness value also includes a measure of the population crowding. Two of the more popular MOEAs are the updated Strength Pareto EA (SPEA2)[43; 44] and the updated Non-dominated Sorting Genetic Algorithm (NSGA2)[45].

SPEA2 uses both dominance and a measure of solution crowding to calculate an individual's fitness. This is done by assigning each individual a strength according to the number of individuals in the population it dominates. A raw fitness for each individual is then calculated as the sum of the strengths of the individual's dominators. In this way, individuals on the pareto front (ones with no dominators) will have a raw fitness of zero. In order to take solution crowding into account, a density estimate equal to $1/(d_k + 2)$ is used, where d_k is the distance in objective space to the kth nearest neighbor. The final fitness of an individual is the sum of the raw fitness and density estimate. This makes individuals on or near the pareto front that are not near many other points have the lowest fitness value. Figure 3.11a illustrates this fitness calculation.

Selection in SPEA2 is done by an environmental selector that first picks individuals in the pareto front and adds them to the population. If the population has too many individuals, it is truncated to the correct size by removing individuals with the highest overall fitness. Since all the individuals in the current population are in the pareto front, their raw fitness is zero, and their overall fitness is exactly the density estimate. After truncation, the population is the least crowded individuals on the pareto front. If the population is too small after the initial pareto front selection, individuals with the lowest fitness are added until it is full. This adds individuals nearest to the pareto front that are not near others so the search can be directed in that area.

The NSGA2 algorithm does not use a concept of fitness directly. However, from its ranking operators a fitness can be derived. Basically, it uses a ranked dominance concept. Individuals are grouped into successive pareto fronts by finding the first front, removing it from the population, and repeating. Each front has a successively higher fitness. In order to reduce solution crowding, NSGA2 uses a density estimation for each individual. It is based on the average side length of the largest cuboid containing only that individual. The cuboid is found by finding the nearest neighbor in both directions for each objective. The average Manhattan distance to each of these neighbors is crowding measure. Individuals with a larger average cuboid side length are more fit since they are less crowded. Figure 3.11b illustrates this fitness calculation.

Based on this derived fitness, NSGA2's selection is not entirely different from SPEA2. It first adds successive fronts to the population until it is full or over full. If the population is over full, the most crowded individuals are truncated from it. This selection function maintains the pareto front, and drives the search towards less crowded areas.

These fitness computations are much more complex than the simple weighted sum used in a SOEA, but their advantages are great. First, there is no need to choose a weighting vector, and thus no introduction of bias into the optimization algorithm. The solution from a MOEA is an approximation to the true pareto front given by the final population. This way the optimal solution for different possible weightings of objectives can be seen without running another optimization. Furthermore, the need for a priori knowledge about the weighting trade-offs is eliminated. Each solution from a MOEA can be considered a trade-off solution for different objective weightings. Finally, by encouraging less dense solutions in the population, there is less clustering, and therefore more diversity in the population even after many generations. This allows for better exploration of the search space near the pareto front. The red points in Figure 3.10 give a comparison between the results of a (a) SOEA and a (b) MOEA.

3.2.4 Heuristics in search algorithms

Since there is not one good search algorithm for all problems, specializing algorithms to work well on single problems is a must. Using information about a problem's structure to facilitate a search algorithm is called a heuristic search. Even using information about the smoothness of the parameter space can be considered heuristic or meta-heuristic. Solving very complex optimization problems would not be practical without drawing on information from the problem to help formulate a search algorithm. In the algorithms presented in this thesis, the use of heuristic information plays a pivotal role. Without this information, these algorithms would break down to simple random searches.

Chapter 4

Proposed Method

The individual well adaptive body bias presented in this thesis has two major divisions in methodology. One method uses floating nwells and the other uses a voltage divider to set the well voltage. The term *floated* well refers to the two possible methods in which a well is either truly floating or connected to the voltage divider. Each major division also has subdivisions called bias modes. Each bias mode is a different combination of the possible biasing methods of V_{dd}, V_{b_n}, V_{b_p}. For example, one mode of IWABB might use nwell floating with the voltage divider, with V_{b_p} and V_{dd} biasing. The following sections present the circuit structures needed to accomplish IWABB with both floated wells and voltage divider controlled wells for all biasing modes.

Along with the methods of controlling the biasing, this chapter presents three algorithms for choosing the best configuration of biasing. Two of the algorithms are based on evolutionary algorithms, and one is based on a gradient-local random walk. These algorithms are designed to explore the search space in very few objective function evaluations which require time-consuming circuit simulations. The algorithms can be easily adapted for use on post-manufacturing test equipment. Since determination of operating parameters in the real world is significantly faster than circuit simulations, more aggressive algorithms may yield superior results.

4.1 Circuit structure

4.1.1 Simulated well capacitance

Although nwell parasitic resistance (R) and capacitance (C) are usually ignored until final layout verifications, these properties can play an important role in circuit operation. Most of the time it is assumed that pfet bodies are tied directly to an ideal voltage source. This makes for very consistent transistor operation since the body voltage does not bounce when the transistor or one of its neighbors switches. However, the actual body voltage contact is not ideal and is located a

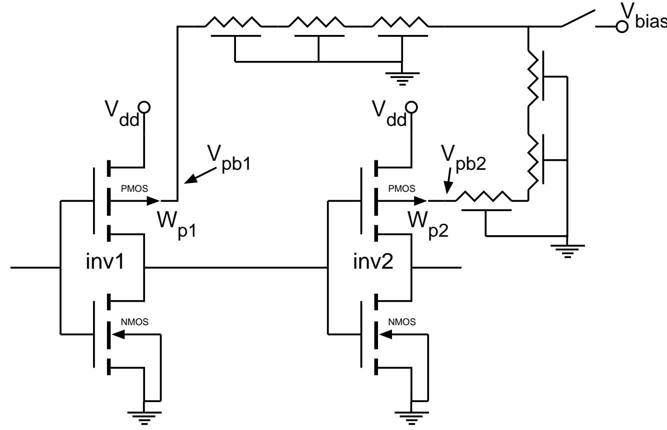


Figure 4.1: Schematic of inverter chain test circuit to investigate body voltage bounce when nwell parasitic resistance and capacitance are included.

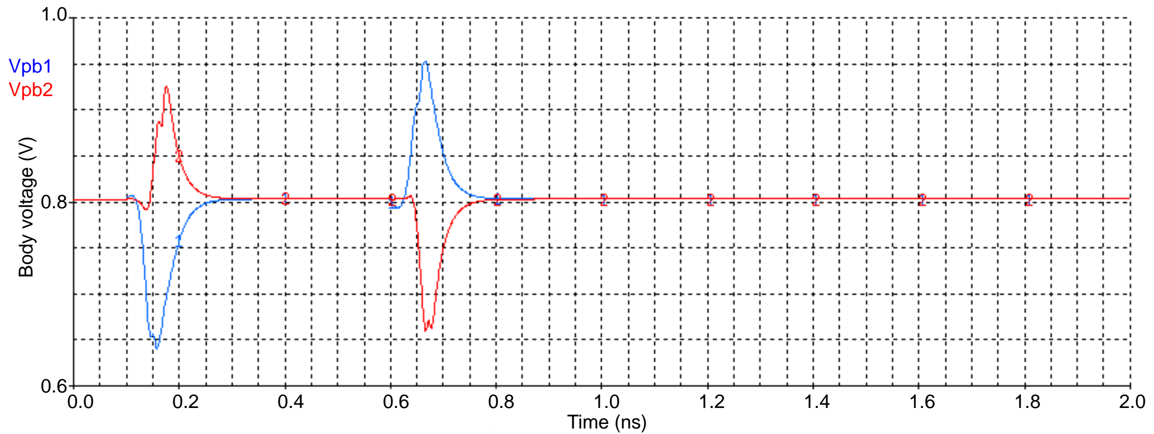


Figure 4.2: Body voltage bounce in the inverter chain shown in Figure 4.1 with the nwell biased to 0.8V by an ideal source, $W_{p1} = W_{p2} = 1\mu$m and the nfets appropriately sized to balance the inverters.

finite distance from the transistor. The connection to the voltage supply is shielded by the parasitic R and C of the nwell. On top of the shielding from the voltage source, each transistor is coupled to all other transistors in the well. This coupling and shielding allows transistor body voltages to bounce significantly when a transition occurs. To exemplify this, a simple inverter chain (Figure 4.1) is simulated through two transitions in a commercial 100nm CMOS process. The parasitic nwell R and C is modeled with a distributed RC equivalent. The well contact is located half the maximum allowed distance from the transistor bodies. Figure 4.2 shows the voltage at the pfet bodies (V_{b_p}) as the transitions occur. The well contact is tied to an ideal 0.8V supply in order to model forward body biasing to a voltage nearly equivalent to floating the well. To model an nwell with several gates switching nearly simultaneously, both pfets are 1μm wide.

There is about 150mV bounce on V_{b_p} as each gate switches. The recovery time from such bounces is simply the time it takes for current from the body bias contact to charge the parasitic well capacitance through the parasitic well resistance. If there is another transition before the body voltage is fully recovered, the body effect causes a shift in V_{th} of the transistor. For pfets, this shift is characterized by the equation

$$V_{th} = V_{th0} - \gamma_p(\sqrt{V_{bs} + 2\phi_F} - \sqrt{2\phi_F}) \tag{4.1}$$

where V_{bs} is the body to source voltage, V_{th0} is the threshold voltage when V_{bs} =0V, γ_p is the pfet body effect parameter, and $2\phi_F$ is surface potential[25]. This bounce induced body effect tends to preload the transistor so that the threshold voltage favors a change in state.

For example, consider an inverter whose output has just switched to 0. This switching event pulls the body of the pfet lower than its source ($V_{bs} \leq 0$, a forward body bias), thus making V_{th} of the pfet less negative. Now, for the next switching event, the pfet must return to saturation, meaning the source to gate voltage (V_{sg}) must become greater than $-V_{th}$ ($V_{sg} \geq -V_{th}$). With V_{th} less negative, the pfet will reach saturation sooner as the inverter input goes from 1 to 0 than it would have if the body voltage did not bounce. Thus, the inverter output will switch up faster. This up transition on the output causes the pfet's body voltage to bounce up ($V_{bs} \geq 0$, a reverse body bias). The higher V_{bs} causes a more negative V_{th}, allowing the pfet to come out of saturation more quickly as the inverter input rises from 0 to 1.

This effect is exactly what is referred to in silicon on insulator (SOI) technology as floating body transient V_{bs} effects[1]. In general, these effects account for a significant part of the speed advantages SOI processes have over bulk CMOS[46]. This begs the question: Why not just float the bodies like in SOI? By taking advantage of the self biasing nature of the nwell caused by the capacitive coupling between transistors, the body bias can be dynamically adjusted. Floating the nwell is slightly different from SOI in that the well voltage is controlled by the current and previous states of *all* the transistors in a well instead of just a single transistor. This fact actually gives added control over the body bias since the ratio of up switching to down switching transistors can be adjusted. The circuit in Figure 4.1 is used again in order to investigate the proper ratio of up to down switching transistors in a well and the performance advantage of floating a well over biasing it. Figure 4.3 compares the bounce in V_{b_p} at inv2 with a floated and biased nwell as the widths of pfets in the inverters varies. The nfets in the inverters are scaled with the pfets to keep each gate

[1]These effects are similar to body voltage hysteresis, but are usually associated more with transients caused by rapid, repeated switching events.

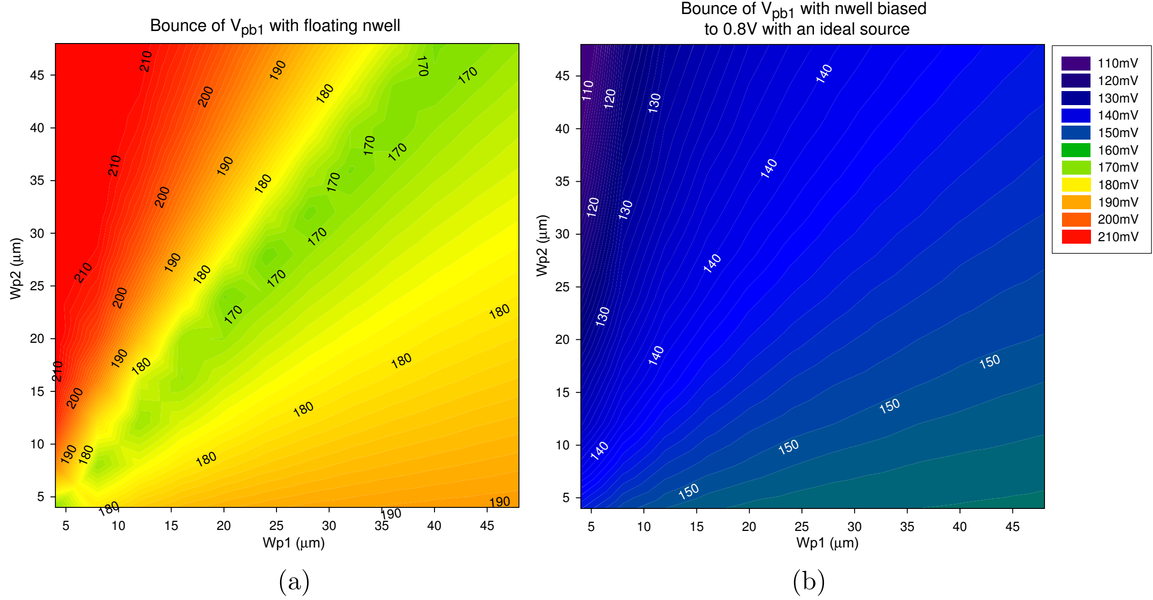


Figure 4.3: Comparison of body voltage bounce V_{pb1} in the inverter chain shown in Figure 4.1 with the nwell (a) floated and (b) biased to 0.8V for varying W_{p1} and W_{p2}.

balanced. Again, each pfet body is connected to the well contact through a distributed RC model of the well parasitics. The pfets are half the maximum allowed distance from the well contact. It is important to note that as the pfet widths are varied, the parasitic well capacitance is increased and the parasitic well resistance decreased proportionally. In the biased case, the nwell is connected to an ideal 0.8V source which is approximately the steady state V_{b_p} voltage in the floated case. It can be seen that the floated nwell case has more bounce in V_{b_p}.

Figure 4.4 shows the inverter chain delay benefits from the additional bounce on V_{b_p}. The floated nwell (SOI-like) configuration yields about a 2% decrease in delay due to the floating body transient effects over a similarly forward-biased nwell configuration. These floating body transient effects can only be realized when the body of fets bounce due to switching events. This requires well parasitics to be considered during simulation. Without a model of the well parasitics, these effects cannot be accurately realized.

Combining the results from Figures 4.3 and 4.4, the smallest benefit to delay comes when the ratio of up to down switch pfets in the well is close to 1. The floating body transient effects are greatest when the bounce in V_{b_p} is greatest, which occurs when all the pfets in the well switch at the same time. Having all the pfets in a well switch at the same time is not feasible, though. For random logic averaged over time, the switching ratio will be near one. However, grouping critical path logic in such a way to maximize coincident switching during crucial events may be a way to

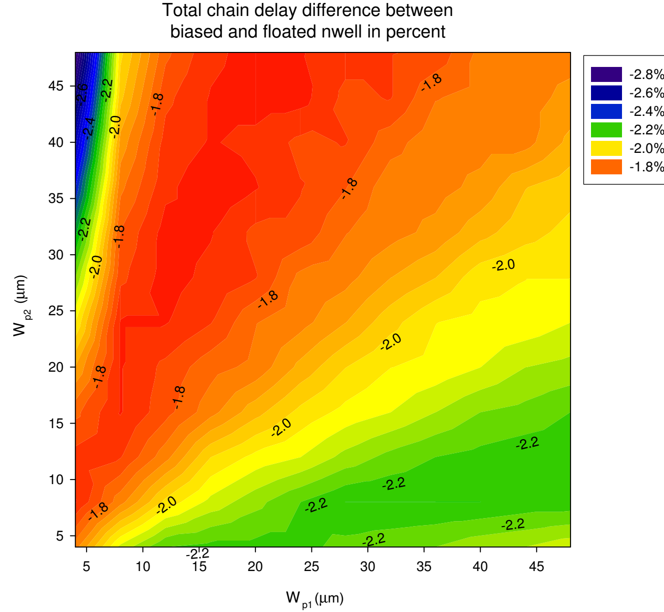


Figure 4.4: Change in total inverter chain (Figure 4.1) delay from 0.8V biased nwell to floated nwell $((d_{floated} - d_{biased})/\mathrm{mean}(d_{floated}))$ for varying W_{p1} and W_{p2}.

close timing on trouble paths. One could also group critical path pfets with large dummy pfets that switch at the same time in order to realize the transient speed effects. The designer would need to consider how much V_{b_p} bounce is acceptable for the design, though.

4.1.2 Well voltage control structures

In order to implement IWABB for all the different biasing modes, a simple well voltage control circuit needs to be added to each nwell. Since these circuits are so small in area, they can be added to every nwell on a chip without much difficulty. If even a small increase in area is too much, the well control structure can be added to only nwells on the critical paths.

There are two basic structures in the well control circuit: a pullup fet and a voltage divider. The voltage divider is not used in the floating well method of IWABB, but the pullup fet remains the same. Both of these structures are constructed with pfets in the nwell they control. Both structures are illustrated schematically in Figure 4.5.

4.1.3 Balancing pullup and voltage divider sizing, activity factor and well polarity

The sizing of the pullup transistor is crucial since it controls the nwell bias of connected wells. It needs to be wide enough to effectively tie the well voltage to the applied bias even when transistors in the well switch, causing a bounce on the well voltage. However, each transistor is shielded by the

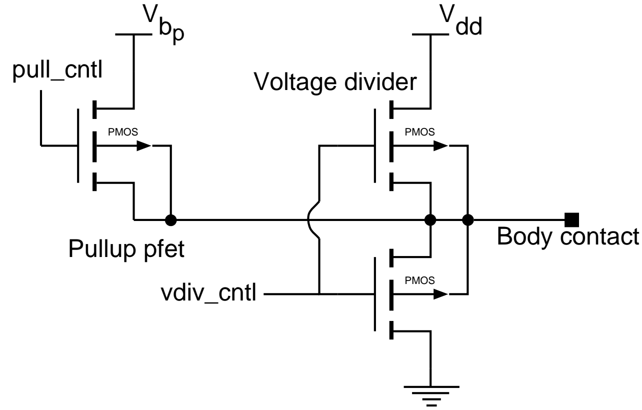


Figure 4.5: Well voltage control structures.

parasitic R and C of the well between the location of the pullup and the location of the transistor. Simulations of Figure 4.1 with varying inverter size are shown in Figure 4.6. As before, the parasitic R and C of the well is scaled with the pfets, and each pfet is half the maximum allowed distance to the well contact. In Figure 4.6a, the nwell is connected to an ideal 0.8V source. The V_{b_p} bounce is only strapped to within several hundred millivolts. Figure 4.6b shows a 1.0μm wide, minimum length pullup transistor is able to accomplish almost the same strapping.

Another consideration is the leakage through the pullup when it is turned off to allow the well to float or be controlled by the voltage divider. This leakage current is primarily determined by the pullup transistor length, the pullup transistor width, the nwell bias voltage applied to the source of the pullup, and the well voltage at the contact. If the pullup transistor is too short, the leakage through it can actually charge the floating or voltage divider controlled well. The leakage current can be substantial since increasing V_s increases V_{sd} and V_{sg} making the transistor more conductive. As the nwell bias voltage increases, current leaking through the off pullup will increase the well voltage. This increase causes a reduction in the static power consumption of the transistors in the well. However, the increase in pullup leakage power can be more than the static power decrease in the well, causing a net increase in power consumption. The solution to this leakage is to reduce the pullup transistor's I_{off} by increasing its L_{eff}. Figure 4.7 shows the floating pfet body voltage at inv1 of Figure 4.1 with a 1.0μm wide pullup pfet. The length of the pullup varies from one to three times the minimum and the nwell bias voltage varies from 1.08V to 1.18V. When the pullup is minimum length and the nwell bias is 1.18V, the floating body voltage is almost 70mV above the target of 850mV. This takes the well voltage out of the *EDP*-optimal biasing range discussed later. Increasing the pullup length reduces this leakage-induced V_{b_p} increase and only marginally reduces the pullup's

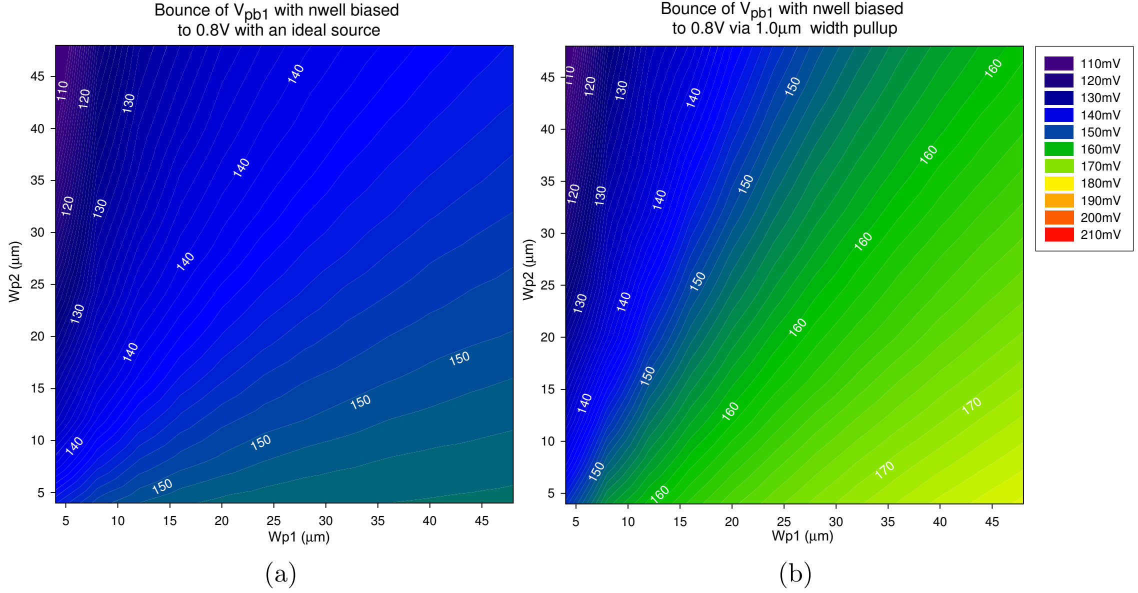


Figure 4.6: Comparison of V_{b_p} bounce for the circuit shown in 4.1 with (a) an ideal 0.8V source well contact and (b) a 1.0μm wide, minimum length pullup pfet.

voltage strapping effectiveness. To compensate, the width of the pullup can be increased, but this was determined to be unnecessary. The pullup transistors in the test circuits shown in Section 4.1.4 are 1.0μm wide and 0.3μm in length.

The static power change in the well caused by a change in the nwell bias voltage is determined by several factors including the overall size of the well, the voltage at the body of the transistors in the well, and the activity factor in the well (α). The final factor in this is very important since it can govern whether individual well biasing is beneficial. As discussed in Section 3.1.3, body biasing mainly changes the static power of transistors. If α of a well is too low, that is, there are too few switching transistors in the well for a given time period, then forward biasing the nwell will cause a large increase in the static power which accounts for a relatively large percentage of the total power. However, since I_{off} is exponentially decaying with respect to the nwell body bias, increasing the nwell bias causes a progressively smaller decrease in the static power. These trends can be seen in the low activity curve in Figure 4.8. With a larger α, the benefit nwell forward bias can be insignificant with respect to power dissipation. The test circuits in this thesis use an activity factor of approximately 20%.

One of the most important parameters of IWABB is the floating or voltage-divider-controlled well voltage. The optimal well voltage is process and circuit dependent and can be determined by examining the total power and delay of the circuit with varying well voltages. Figure 4.9 shows the circuit delay, P_{op}, power-delay product (PDP) and energy-delay product (EDP) as a function

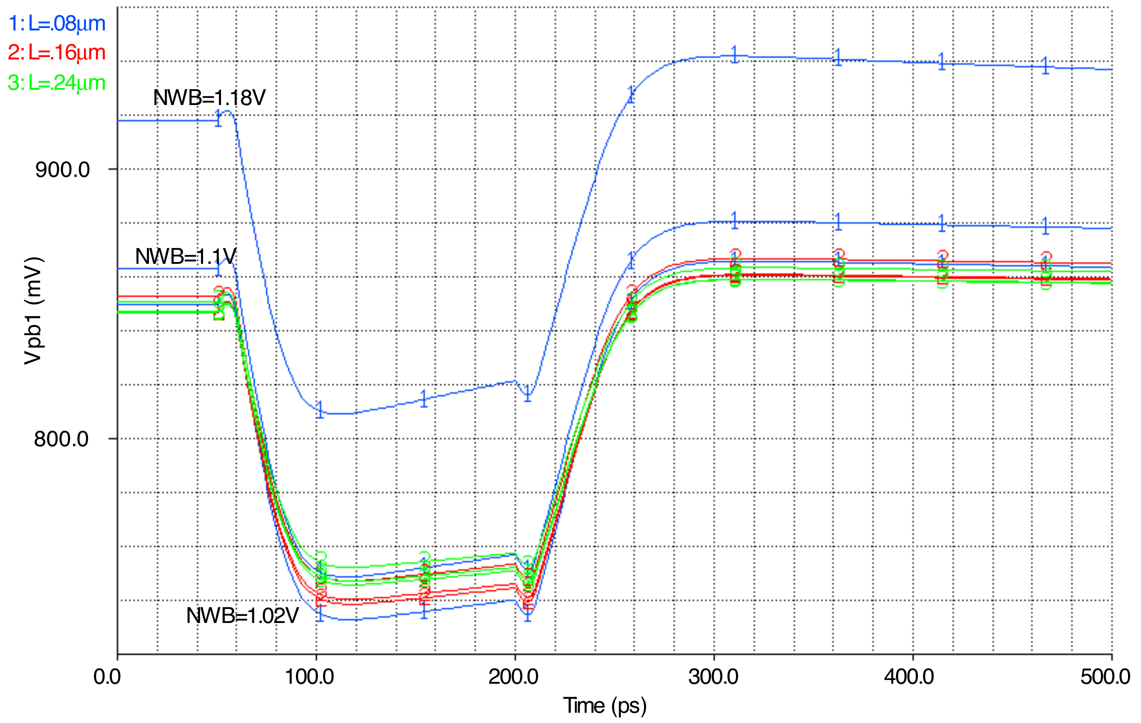


Figure 4.7: Floating pfet body voltage at inv1 of Figure 4.1 with a 1.0μm wide pullup pfet. The length of the pullup varies from one to three times the minimum and the nwell bias voltage varies from 1.08V to 1.18V.

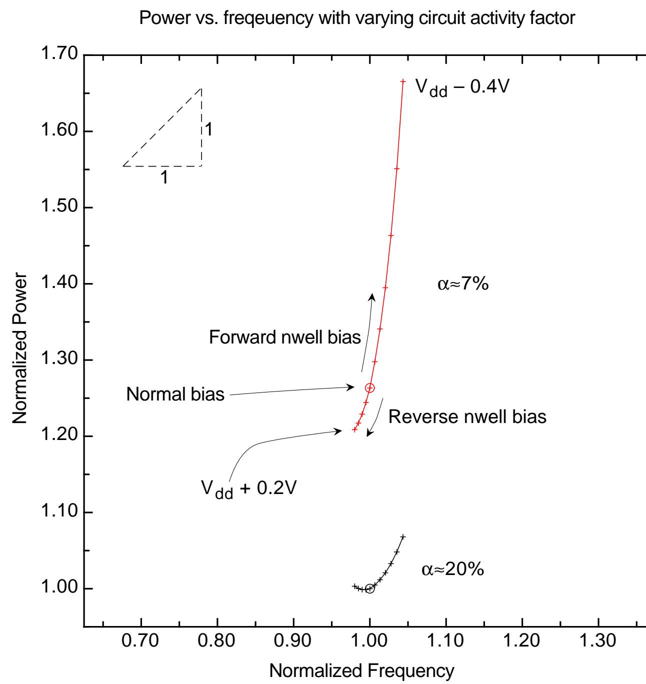


Figure 4.8: P_{op} versus f_{op} for the *adder32* test circuit with activity factors of about 7% and 20%. The nwell bias voltage varies from $V_{dd} - 0.4$V to $V_{dd} + 0.2$V. Power and frequency are normalized to the high activity circuit with $V_{b_p} = V_{dd}$.

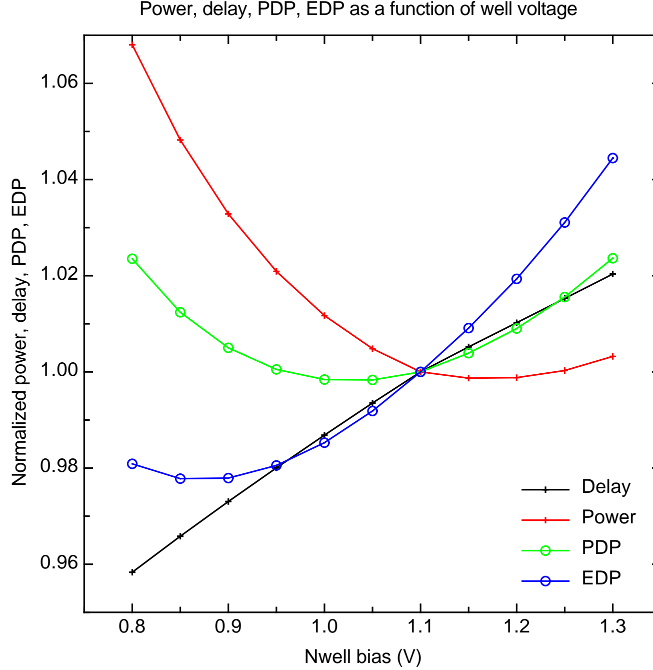


Figure 4.9: Circuit delay, P_{op}, power-delay product and energy-delay product for the *adder32* test circuit with varying V_{b_p}. All quantities are normalized to their values at $V_{b_p} = V_{dd}$.

of the nwell bias for *adder32*. The minimum in the *EDP* curve at about 850mV represents the optimal nwell biasing point for this circuit and process[47]. This biasing point gives the optimal trade-off between performance and power. Biasing at any other point would diminish the usefulness of IWABB.

For the voltage-divider controlled well method, the well voltage is determined by the ratio of the lengths of the transistors in the voltage divider. This makes it possible to tune the well voltage to the optimal point without having to carefully group the pfets based on switch polarity. The total length of the transistors in the voltage divider determines the static short-circuit current when the voltage divider is on. This power would need to be budgeted for the chip and is not included in the power measurements in this paper. In general, the voltage divider power dissipation can be on the order of 20μW, and not be a concern in overall chip power. For a large chip with 10^7 pfets divided into 10^4 nwells this amounts to about 0.2W. Given the power budget of modern microprocessors beyond 100W, this is only 0.2% of the overall power budget. Figure 4.10 shows the selection of the voltage-divider lengths based on the well voltage and voltage-divider leakage current. The voltage dividers implemented in this thesis's test circuits use minimum width pfets with lengths of 2.0μm and 1.8μm for the top and bottom pfets, respectively. This sizing gives a steady-state well voltage of about 850mV with about 2μA voltage divider leakage.

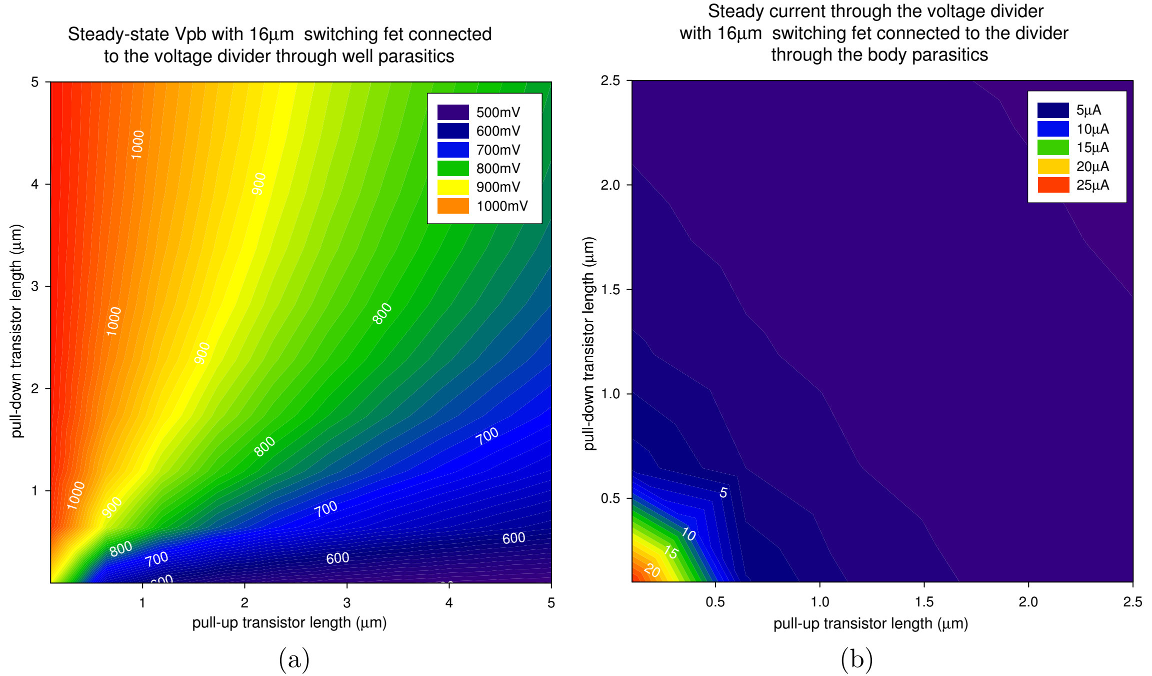


Figure 4.10: Voltage divider sizing tests based on (a) steady-state V_{b_p} and (b) voltage-divider static current with 16μm switching inverter in the nwell for varying voltage divider component lengths.

For the floating well method, the floating well voltage is determined primarily by the polarity of the transistors in the well. If a majority of the pfets in a well are conducting (weighted by their widths), the well will tend to float at higher voltage than a well with more non-conducting pfet width. In order to achieve optimal floating well voltage, one could group pfets into nwells such that the most common well polarity gave a voltage near the optimal point on the *EDP* curve in Figure 4.9.

4.1.4 Test circuits

Two test circuits are used in this thesis. The test circuits have different characteristics and basic structures in order to test IWABB on a variety of circuit topologies. Each test circuit is not intended to represent the characteristics of an entire microprocessor. Instead they model a single critical path that would occupy a distributed area across a die. The first circuit, *adder32*, was chosen mainly for its simplicity. It is a basic 32-bit static carry-ripple adder containing 896 transistors. The second circuit, *L64buP*, is a critical path from a commercial 64-bit microprocessor consisting of mostly dynamic logic as well as block to block drivers with 4279 transistors overall. The pfets in both circuits are grouped into artificial nwells. *adder32*'s nwells are based on each individual adder bit slice, making 32 nwells. *L64buP* is broken down into its major functional blocks and grouped according to proximity in a functional flow chart. Thirty-three groups are formed from 75 function

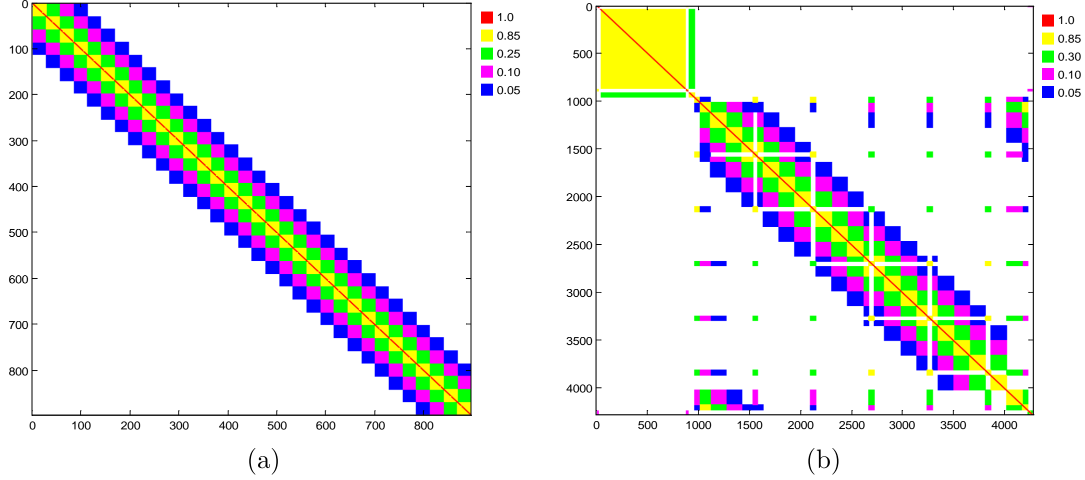


Figure 4.11: Graphical representation of the structure of the correlation matrices used in (a) *adder32* and (b) *L64buP* Monte Carlo manufacturing variability simulation.

blocks and one group is used for odd transistors (e.g. transistors representing load blocks). Both circuits have simulated well parasitics added to each pfet as described in Section 4.1.1.

The correlation matrices of the L variations for each circuit are qualitatively determined based on the geography of a reasonable layout. That is, consecutive logic blocks in a functional flow chart are assumed to be geographic neighbors. The actual correlation values are determined by trial and error for the reasons described in Section 3.1.4. For *adder32*, transistors in the same adder bit are set to have very highly correlated variations, while variations of widely separated adder bits are not correlated. Successively neighboring adder bits have correlation coefficients of 0.85, 0.25, 0.10, 0.05 and 0.00. Transistors in bits further than three bits apart have only random correlation in their variation. These correlations are used to construct a 896×896 four-diagonal 28×28 block matrix ρ as illustrated in Figure 4.11a.

The structure of ρ is more complex for *L64buP* than for *adder32* since its functional flow chart is more complex. Successive logic blocks were given correlation coefficients of 0.85, 0.30, 0.10, 0.05 and 0.00. This produced a 4279×4279 matrix with the structure represented in Figure 4.11b.

The standard deviation σ used to generate the new lengths is 5nm. This is consistent with the ~18% variation at 3σ cited in [3]. Figure 4.12 shows the pre-IWABB characteristics of one hundred chips generated by this simulation for *adder32* and *L64buP*. The *adder32* circuit shows a distribution ranging about 15% in f_{op} and about 25% in P_{op}. The dynamic *L64buP* circuit shows a wider distribution than *adder32* with a range of about 30% in f_{op} and 55% in P_{op}.

The correlation between both f_{op} and P_{op} with respect to the minimum, mean and maximum

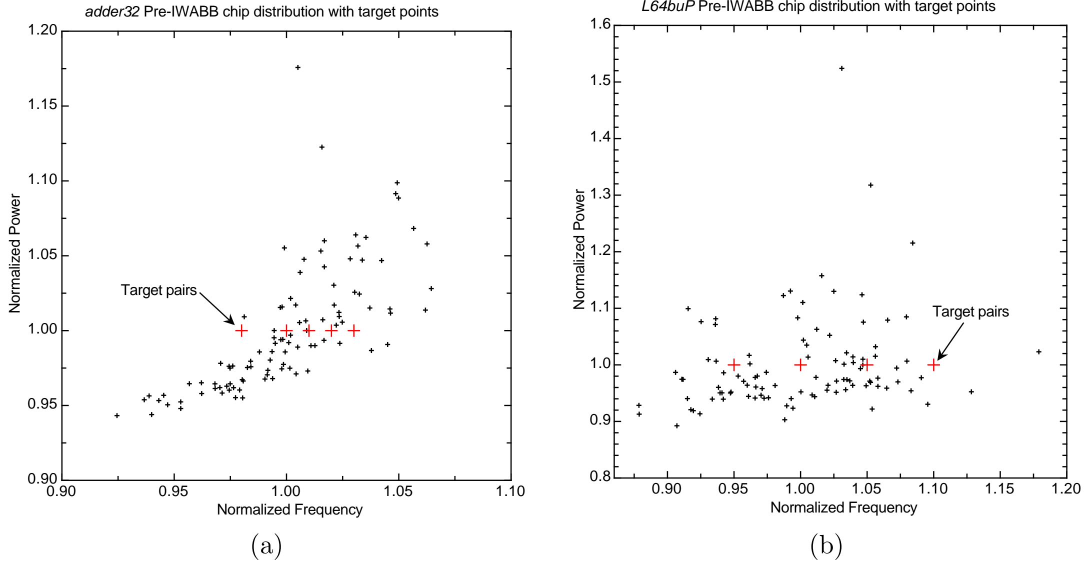


Figure 4.12: Pre-IWABB power and frequency characteristics of 100 (a) *adder32* and (b) *L64buP* chips generated by the Monte Carlo simulation shown with the target power-frequency pairs (f_t, P_t) used in IWABB testing.

L_{eff} deviations from the mean for each *adder32* chip is shown in Figure 4.13. The minimum deviation in L_{eff} gives a representation of the shortest L_{eff} on each chip, and the maximum is the longest L_{eff} on each chip. It can be seen that f_{op} is most strongly correlated to the mean deviations on an entire chip. This is consistent with logic, since in this static ripple adder every transistor is part of the critical path. Therefore, the mean L_{eff} of every gate on the chip determines the delay of the circuit. The minimum and maximum L_{eff} deviations are only inherently correlated with f_{op} since they contribute to the mean. However, P_{op} is shown to be most highly dependent on the shortest L_{eff} on the chip. This is consistent with theory because the shortest transistors will have the highest leakage, causing P_{op} to be higher due to increased static power. There is also a strong correlation between P_{op} and the mean L_{eff} since it determines the frequency, and the dynamic power is proportional to the frequency.

4.2 IWABB algorithms

In order to thoroughly investigate the effectiveness of the IWABB principle, three different algorithms are tested. Different algorithms might return solutions of different quality, so it is useful to compare the results. Also, different algorithms have different efficiencies. When implemented in silicon, IWABB would be performed during post-fabrication test. Each minute of test time adds cost to the product, so algorithms which can quickly achieve a good solution may be better. To make the use

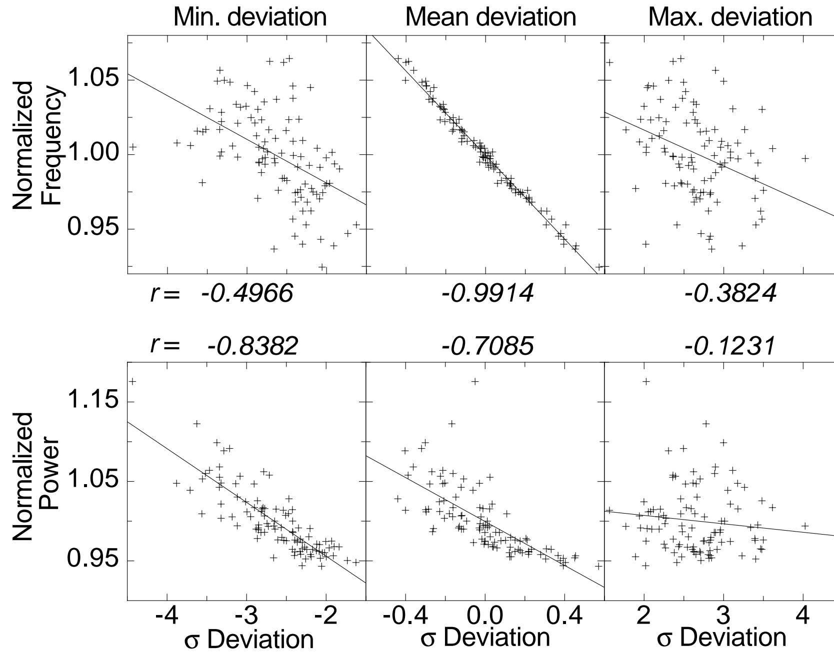


Figure 4.13: Correlation of P_{op} and f_{op} to minimum, mean and maximum L_{eff} deviations from the mean shown with their repsective r values.

of IWABB worthwhile, the improvement in overall yield needs to more than offset the added cost in test time.

Each of the algorithms can handle all of the different biasing modes. Each mode is a combination of possible biasings $(V_{dd}, V_{b_p}, V_{b_n})$ and nwell floating. However, microprocessors (and most other digital circuits) contain circuit components that are sensitive to adjusting V_{b_n}. For example, most digital circuits use static random access memories (SRAMs). Modern microprocessors have vast amounts of SRAM used for caches. The read-write threshold of these SRAMs is affected by V_{b_n}. Therefore, changing V_{b_n} can affect the reliability of the overall circuit. This make adjusting V_{b_n} via the substrate voltage impossible, so an expensive triple well process would be needed to do V_{b_n} biasing. Section 5.1 investigates modes using V_{b_n} further. Since other research in this area has used V_{b_n} biasing[28; 19; 15], particularly in a form similar to DWB, that mode is included. The abbreviations for the different modes are:

NWF+NWB+VDD	Floating nwells with V_{b_p} and V_{dd} biasing
NWF+VDD	Floating nwells with V_{dd} biasing
NWF+NWB	Floating nwells with V_{b_p} biasing
NWF	Floating nwells
NWB+VDD	V_{b_p} and V_{dd} biasing
DWB	V_{b_n} and V_{b_p} biasing, for comparison to other research
VDD	V_{dd} biasing, for comparison to other research
NWF+DWB	Floating nwells with V_{b_n} and V_{b_p} biasing
NWF+PWB	Floating nwells with V_{b_n} biasing
NWF+PWB+VDD	Floating nwells with V_{b_n} and V_{dd} biasing
NWF+DWB+VDD	Floating nwells with V_{b_n}, V_{b_p} and V_{dd} biasing

The three different algorithms are a pseudo-weighted aggregate SOEA (soIWABB), a gradient-local random walk (gIWABB), and a MOEA (moIWABB). All of the algorithms are setup to optimize both P_{op} and f_{op}. soIWAB and gIWABB use a modified weighted aggregate objective function to reduce this two dimensional objective space to a single fitness value. In order to accomplish this reduction, a target objective vector $\vec{o}^{(T)} = (f_{min}, P_{max})$ must be supplied. moIWABB actually works with an objective space comprised of $\vec{o} = (P_{op}, 1/f_{op})$ and no dimensional reduction is needed.

All the algorithms use the same parameter space encoding regardless of the biasing mode used. The parameter space is encoded in a chromosome consisting of three real numbers representing each of the biases and a binary sub-chromosome representing the floating well configuration. The value of each bit in the binary sub-chromosome determines if the corresponding well is connected or floated. If any part of the chromosome is not available in the biasing mode, it is forced to its default value and remains unchanged by the algorithm.

4.2.1 Pseudo-weighted aggregate SOEA IWABB (soIWABB)

The soIWABB algorithm is based on a simple genetic algorithm using single tournament selection and a pseudo-weighted aggregate objective function. However, it is heavily hybridized as suggested by [48]. The hybridization uses heuristic information about the known trends in IWABB. For example, it is obvious that adding a floating well will always increase P_{op} and usually f_{op}. The magnitude of the changes depend on several things, but the trend is inescapable. By using this sort of information, the speed of the algorithm is greatly improved.

The soIWABB algorithm pseudo code is shown in Table 4.1. Its general form is the same as the standard EA in that it starts with a randomly generated population, selects parents, generates children by crossover and mutation, and reinserts the children into the population. The hybridization and heuristics lie in the specific functions used for each of these steps.

The generation of the initial population uses the most heuristics of any step. It attempts to generate a population that is near the objective targets, $\vec{o}^{(T)}$, by performing an estimated gradient

for each *chip*	
initialize individual *best_ind*	Clear the best individual
initialize *population*	
$popsize = 0$	
evaluate connected and half floated	Gives float gradient info
evaluate bias tests	Generates bias gradient info
while $popsize < maxpopsize$	Make random initial population
create new individual *ind*	
Randomly choose float or bias	Favor more effective method
If bias chosen	
estimate bias needed to meet req.	Using gradient info
If float chosen	
estimate floats needed to meet req.	Using gradient info
randomly generate needed floats	
evaluate *ind*	
add *ind* to *population*	
while $gen < maxgen$	Begin EA
select two parents	Tournament selection
crossover parents to produce *child*	
mutate *child*	
evaluate *child*	
reinsert *child*	Replace worst fit individual
record *best_ind*	
next *chip*	

Table 4.1: Pseudo code for the soIWABB algorithm.

search in the f_{op} objective. The fully connected, normally biased chip is evaluated and used as the baseline objective vector $\vec{o}(\vec{x}^{(norm)}) = (f_{op}^{(norm)}, P_{op}^{(norm)})$. If NWF is used in the current mode, an individual, $\vec{x}^{(float)}$, with about half of its nwells floating (chosen randomly) is evaluated, and the slopes for nwell floats are calculated as:

$$\frac{\partial f_{op}}{\partial NWF} \approx \frac{o_1(\vec{x}^{(float)}) - o_1(\vec{x}^{(norm)})}{n}$$
$$\frac{\partial P_{op}}{\partial NWF} \approx \frac{o_2(\vec{x}^{(float)}) - o_2(\vec{x}^{(norm)})}{n}$$

where n is the actual number of nwells chosen to float. This is not a completely accurate measure of the slopes since the specific floating wells plays an important role in the change in $\vec{o}(\vec{x})$. However, the ratio of these slopes gives a measure of the effectiveness of floating nwells. This effectiveness, given by

$$\xi_{NWF} = \frac{\left(\frac{\partial P_{op}}{\partial NWF}\right)}{\left(\frac{\partial f_{op}}{\partial NWF}\right)} \approx \left(\frac{\partial P_{op}}{\partial f_{op}}\right)_{NWF}, \tag{4.2}$$

approximates the objective space slope between $\vec{o}(\vec{x}^{(norm)})$ and $\vec{o}(\vec{x}^{(float)})$. For each bias allowed in the biasing mode, soIWABB makes another test evaluation and calculates a similar ξ. Again, these are just estimates of the slopes since the actual slopes depend on the values of the other parameters. The maximum ξ belongs to the best parameter, p, to reduce P_{op} since $(\partial P_{op}/\partial f_{op})_p$ is greatest.

That means changing p will effect P_{op} the most while effecting f_{op} the least. Likewise, the minimum ξ belongs to the best parameter to increase f_{op}.

These objective space slopes are used to calculate another useful piece of information. The algorithm estimates the change needed to be made in each parameter p in order to meet the target objective o_j by a simple linear estimator

$$\epsilon_p = \frac{o_j^{(t)} - o_j^{(norm)}}{\partial o_j / \partial p}. \tag{4.3}$$

In the biasing modes, ϵ_p would be a real number. For nwell floating, ϵ_{NWF} is an estimation of the number of floating nwells required to meet o_j.

Using this information, the initial population is created by generating individuals near the target objectives. Each individual is based on $\vec{x}^{(norm)}$, and parameter changes are added to it. If the chip is initially under the frequency target ($o_1(\vec{x}^{norm}) < f_{op}$), the choice of the parameter to change is made randomly, favoring the highest ξ. If a biasing method is chosen in a mode that also allows NWF, a fraction of the estimated floats is also added to increase the population diversity. Before the individual is evaluated and added to the population, the slopes $\partial P_{op}/\partial p$ are used to estimate the P_{op} of the new individual. If it is estimated that the new individual is above the power requirement, an effort is made to reduce the power by changing the biasing (if allowed in the mode). A similar method to above is used to determine the best bias to change. Finally, the individual is evaluated and added to the population.

The pseudo code for the evaluation function is shown in Table 4.2. The evaluation of individuals is the most time consuming process in all of these algorithms since it is making calls to SPICE. Therefore, one key feature to the evaluation function is the cache. The first thing the algorithm does when evaluating an individual is it checks to see if that individual is in the cache, meaning that it has already been evaluated. If the individual is in the cache, it immediately returns the results of the previous evaluation stored in the cache. For an individual not in the cache, the evaluation function writes an *ispice* formatted runinput circuit file and runs *ispice*. The *ispice* output is parsed by *hpspice* which returns f_{op} and P_{op} for the circuit configuration defined by the individual.

The evaluation function assigns a fitness to the individual based on these objective values. This objective function is loosely based on the weighted aggregate function where the fitness is the weighted sum of the objectives. If the individual violates only one target objective, the fitness is simply the violation magnitude, normalized to the target. In the weighted aggregate scheme this would be weights of 0 and 1 for the non-violating and violating objectives, respectively. If both

for each $cached_ind$ in cache	Check if configuration is in cache
if $cached_ind = ind$	
return $cached_ind$	
write circuit for $chip, ind$	Runinput format
call *ispice*	Simulates runinput
call *hpspice*	Reports results of simulation
assign $fitness$ to ind	Aggregate/distance based fitness
if ind is better than $best_ind$	
$best_ind = ind$	
if $best_ind$ meets req.	
record $best_ind$	
next $chip$	
add ind to cache	
return ind	

Table 4.2: Pseudo code for the **evaluate** function used in soIWABB, gIWABB and moIWABB

objectives violate the targets, the fitness is the distance to the targets in normalized objective space:

$$f(\vec{o}) = \sqrt{\left(\frac{\vec{o}_1^{(T)} - \vec{o}_1}{\vec{o}_1^{(T)}}\right)^2 + \left(\frac{\vec{o}_2^{(T)} - \vec{o}_2}{\vec{o}_2^{(T)}}\right)^2} \tag{4.4}$$

This is also similar to a weighted aggregate objective function. An equally weighted aggregate objective function would be equivalent to the Manhattan distance in objective space from the individual to the target. Equation 4.4 is simply the Euclidean distance in objective space between the same points. Finally, if both objectives meet or beat the targets, the individual is good, and the fitness is simply the negative of equation 4.4. The evaluation function keeps track of the best individual in $best_ind$. If the newly evaluated individual is better than $best_ind$, the individual replaces $best_ind$. If the individual is good, the algorithm records it, and moves on to the next chip. Finally, the evaluation function adds the newly evaluated individual to the cache and returns the evaluated individual to the algorithm.

Once the initial population is filled, the algorithm uses tournament selection to pick one pair of parent individuals for reproduction. Through crossover and mutation, these parents produce one child. This single offspring method is similar to reproduction model used in [38]. The pseudo code for the crossover function is in Table 4.3. In order to provide greater evolutionary pressure on the population in soIWABB, the crossover function favors the stronger parent. It also uses heuristic information when available in order to speed convergence.

The crossover function first determines the stronger parent by comparing the parental fitnesses. A child binary chromosome representing the floating nwell configuration is created with uniform crossover, favoring the stronger parent. In order to increase the amount of local search the algorithm performs, the crossover function limits the change in number of floating nwells. The child

chromosome is allowed only one more or one less float than the average of the two parents. The child's biasing is determined by small increases or decreases from the parental average based on heuristics and the allowed biasing in the mode.

Pseudo code	Comment
determine stronger parent	
for each *nwell_bit* in *child*	Uniform crossover
randomly choose parent	Favor stronger parent
child nwell_bit = chosen parent *nwell_bit*	
while child floats>average parent floats+1	Slowly evolve number of floats
randomly remove one float	
while child floats<average parent floats−1	
randomly add one float	
child bias = average parent *bias*	
if average parent $P_{op} > P_{max}$	
randomly choose:	Depending on biasing mode
increase reverse bias on nfets	Decrease V_{b_n}
increase reverse bias on pfets	Increase V_{b_p}
reduce V_{dd}	
if average parent $f_{op} < f_{min}$	
randomly choose:	Depending on biasing mode
increase forward bias on nfets	Increase V_{b_n}
increase forward bias on pfets	Decrease V_{b_p}
increase V_{dd}	
return *child*	

Table 4.3: Pseudo code for the **crossover** function in soIWABB.

The child produced by the crossover function is mutated in order to maintain population diversity and move the algorithm out of local minima. The mutation function is shown in Table 4.4. The mutation is simply the random changing of each of the parameters. It can randomly exchange the state of two nwell bits. This keeps the number of floated wells the same, but changes the configuration. The mutation can also randomly flip bits in the binary nwell chromosome which has the effect of connecting a floated well or floating a connected well. It can also randomly increase or decrease all the allowed biases. The rate of mutation is kept at approximately $1/n$ where n is the population size[41; 42].

4.2.2 Gradient-local random walk IWABB (gIWABB)

The gIWABB algorithm is more aggressive in its search, and sacrifices performance for improved quality of solution in comparison to soIWABB. It shares many strategies with soIWABB, and uses the same evaluation function. It also makes use of even more heuristics. Table 4.5 shows the pseudo code for gIWABB.

gIWABB starts similarly to soIWABB by finding gradient information. gIWABB does not use biasing gradients, though, and instead of generalizing the slopes for nwell floating, it finds the

randomly exchange two nwell states	
randomly float/connect an nwell	
randomly choose:	Depending on biasing mode
increase forward bias on nfets	Increase V_{b_n}
increase reverse bias on nfets	Decrease V_{b_n}
randomly choose:	Depending on biasing mode
increase forward bias on pfets	Decrease V_{b_p}
increase reverse bias on pfets	Increase V_{b_p}
randomly choose:	Depending on biasing mode
increase V_{dd}	
decrease V_{dd}	
return *child*	Mutation rate $\sim 1/popsize$

Table 4.4: Pseudo code for the **mutate** function used in soIWABB.

for each *chip*	
initialize individuals ind, $best_ind$	
evaluate *ind*	
for each $nwell_bit$ in *ind*	Build NWF gradient info
float $nwell_bit$	
evaluate *ind*	
save gradient info	
connect $nwell_bit$	
if $f_{op} < f_{min}$	
$f_{est} = f_{op}$	
while $f_{est} < f_{min}$	
add best float	Connected well with $\min(\partial P_{op}/\partial f_{op})$
update f_{est}	Use gradient info
evaluate *ind*	
while $step < maxstep$	Start random walk
if $f_{op} < f_{min}$	
randomly choose:	Depending on bias mode
increase forward bias on nfets	Increase V_{b_n}
increase forward bias on pfets	Decrease V_{b_p}
add best float	Connected well with $\min(\partial P_{op}/\partial f_{op})$
else if $P_{op} > P_{max}$	
randomly choose:	Depending on bias mode
increase reverse bias on nfets	Decrease V_{b_n}
increase reverse bias on pfets	Increase V_{b_p}
connect worst float	Floated well with $\max(\partial P_{op}/\partial f_{op})$
evaluate *ind*	
next *step*	
record $best_ind$	
next *chip*	

Table 4.5: Pseudo code for gIWABB algorithm.

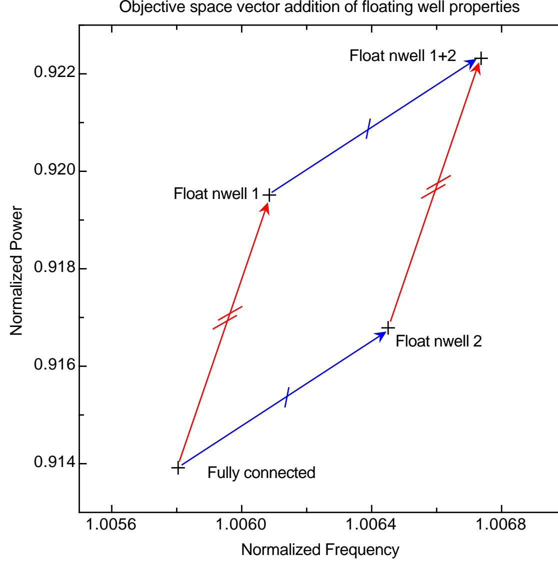


Figure 4.14: Vector addition in objective space of two NWF configurations for an example chip from *adder32*.

gradient for each nwell. The computational overhead of evaluating each well individually is similar to the initial population in soIWABB for small chips, but may become a concern in chips with a large number of wells. The gradient of each well is very useful since the benefit of floating a well is directly related to the variations in the well. A well with mostly positive variations ($L_{eff} > L_d$) is a good candidate to float since doing so will speed up these slower fets. By having larger L_{eff}, these fets are generally lower in leakage power, so floating the well does not increase P_{op} as much as floating a well with small gate lengths. By determining the ξ for each well, the algorithm can find the best wells to float immediately. The gradient information for individual wells is also useful because in the NWF mode, objective vectors can add. That is,

$$\vec{o}(\vec{x}^{(a)} + \vec{x}^{(b)}) \approx \vec{o}(\vec{x}^{(a)}) + \vec{o}(\vec{x}^{(b)}) \tag{4.5}$$

where $\vec{x}^{(a)}$ and $\vec{x}^{(b)}$ are two different binary NWF parameter configurations and $(\vec{x}^{(a)} + \vec{x}^{(b)})$ represents the binary OR of the two. This important property is shown in Figure 4.14 with an example chip from the *adder32* circuit. Objective parameters are shown for floating the first nwell, the second nwell, and both the first and second nwells. It is clear that the objective properties for the configurations add as vectors in the objective space.

The gradients and efficiencies are calculated in the same manner as soIWABB, where $\vec{o}(\vec{x}^{(norm)})$ is used as a baseline for all the calculations. Similar to the soIWABB population initialization, gIWABB attempts to move close to the optimal point. Wells are floated in order of increasing $\partial P_{op}/\partial f_{op}$. The algorithm estimates the change in f_{op} caused by floating each well i with $\partial f_{op}/\partial NWF_i$, and

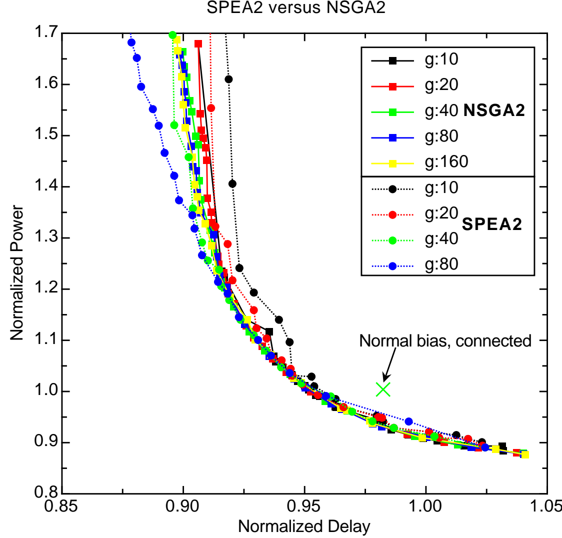


Figure 4.15: Initial comparison between SPEA2 and NSGA2 on an *adder32* test chip.

calculates an estimated frequency f_{est}. While $f_{est} < f_{min}$, gIWABB adds the best f_{op} increasing well. Once the estimated frequency reaches f_{min}, or all the wells are floated, the algorithm uses a local random walk search. Each step changes a random parameter by a small increment. Biases are changed by one bias resolution, and floating wells are added or removed in order of their respective ξ's. Heuristic information is used to guide the direction of the changes. In this way, the random walk acts like a hill climber since it is known that the walk will progress in a direction that improves an objective violation.

4.2.3 MOEA IWABB (moIWABB)

The MOEA based IWABB is based on the SPEA2[44] source code from PISA[49]. SPEA2 was chosen over NSGA2 since its pareto front progressed faster in initial tests (Figure 4.15). The NSGA2 algorithm is based on code from the primary author of [45]. This is by no means a thorough test of the capabilities of the two algorithms, but one algorithm needed to be chosen for closer analysis and modification.

moIWABB has two distinct behaviors. In modes without NWF, it performs the basic SPEA2 algorithm described previously. If NWF is used in the mode, it takes an iterative approach. This approach was adopted in order to limit the spread of the pareto front into regions of the objective space beyond where interesting solutions lay.

The pseudo code for the moIWABB algorithm in modes with NWF is shown in Table 4.6. moIWABB first uses SPEA2 to find a pareto front in the NWF mode. It uses a uniform binary crossover and a simple random mutator with a mutation rate of $1/n$ where n is the population size. The algo-

rithm then iteratively applies different biasing to the NWF pareto front, using a round of the SPEA2 environmental selection between each iteration. The biasing applied depends on those allowed by the biasing mode in use.

for each *chip*	
initialize archive population *arc*	Random individuals
while $gen < maxgen$	SPEA2 for nwell floating only
crossover *arc* to make new *children*	
mutate *children*	
for each $ind \in children$	
evaluate *ind*	
select new *arc* from $arc \bigcup children$	Environmental selection
while $step < maxstep$	Random walk for biases
for each *bias* in bias mode	
initialize population *children*	
for each $ind \in arc$	Produce two pareto fronts
increase *bias*	
evaluate *ind*	
add *ind* to *children*	
decrease *bias*	
evaluate *ind*	
add *ind* to *children*	
select new *arc* from $arc \bigcup children$	Environmental selection
record *arc*	
next *chip*	

Table 4.6: Pseudo code for moIWABB algorithm.

Chapter 5

Experimental Results

There are several different metrics available to judge the different algorithms. In the end, the effectiveness of IWABB is most important, regardless of the algorithmic implementation. Therefore, several tests are run in order to judge each algorithm. The first test is an investigation into the convergence rate of each algorithm. The algorithmic convergence rate relates directly to post-fabrication test time and thus chip cost. The rate of convergence is also important since it relates to the quality of the solution. In order to give each algorithm a chance of finding a good solution, the number of steps allowed is tuned with information from the convergence tests.

It is important for each algorithm to be able to find near optimal solutions, so the second test investigates each algorithm's ability to find good solutions. It is particularly important to find good solutions in the binary sub-chromosome since it represents the largest portion of the search space (2^{32} for *adder32* and 2^{33} for *L64buP*), and is, in some ways, the most difficult to search. To investigate each algorithm's ability to find a good well configuration, it is tested on four test chips based on the *adder32* circuit. The test chips have different patterns of variations, as shown in Figure 5.1. The test chips are referred to as chips 101–104. The optimal configuration of floating wells for these test chips is determined by the wells with longer lengths. Chip 101 should favor floating higher numbered wells. Chip 102 should favor floating wells 1–16. Chip 103 should have every fourth well floated. Chip 104 should have every fourth well floated without floating any other even numbered wells. All the algorithms should approach these optimal configurations. The target objectives for soIWABB and gIWABB are given such that they are just beyond the optimal solution. Therefore, even the optimal solution is not an acceptable chip. This forces the algorithms to search the space near the optimal solution until the maximum number of steps or generations is reached.

In the final test, the results from the three different algorithms are tested using the common metric of yield. Yield is the percentage of chips that meet both P_{max} and f_{min} target requirements. The yield metric is useful for comparing the overall effectiveness of each algorithm as well as the

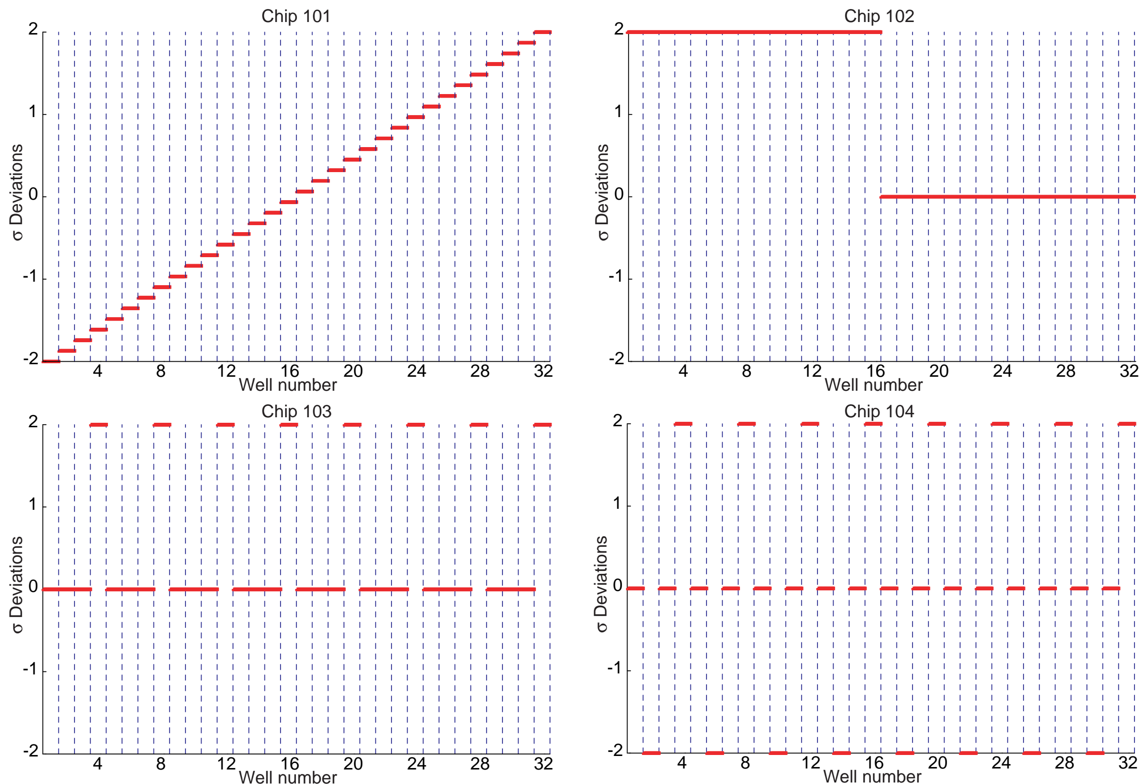


Figure 5.1: L_{eff} deviation from the mean in σ for test chips 101–104.

effectiveness of IWABB in general. Improving this metric is the overall objective to IWABB, even though the algorithms can only address one chip at a time. Yield is easily obtained from soIWABB and gIWABB since they report good and bad chips for each target requirement. The yield is the number of good chips out of the total (not including the test chips). The target requirements used in this test are shown in Figure 4.12. These points were chosen with a very restrictive P_{max} equal to the average P_{op} of all the normally biased, fully connected chips. This choice was made to show the effectiveness of IWABB in the extremely power limited designs of modern microprocessors. For *adder32*, the f_{min} points were chosen to give initial yields ranging from 2% to 32%. The f_{min} points are .98, 1.00, 1.01, 1.02 and 1.03 times the average f_{op} of the normally biased, fully connected chips. The *L64buP* circuit has a wider initial distribution in f_{op}, so the target requirement were chosen to be wider as well. The f_{min} values of 0.95, 1.0, 1.05 and 1.1times the average f_{op} of the normally biased, connected chips. These points give initial yields ranging from 1%–46%. moIWABB does not use target requirements within the algorithm, but instead returns an approximation to the pareto front in the form of a population. While it is interesting to look at the pareto front obtained, it needs to be reduced to a yield in order to directly compare the effectiveness of the algorithms. The

specifics of how this is done is discussed in Section 5.4. The other sections in this chapter summarize the results obtained from soIWABB and gIWABB.

5.1 Initial investigations

Due to the prohibitively long simulation time required for running all algorithms on all circuits in every possible mode, intermediate conclusions based on initial tests were used to reduce the number of overall investigations. Each investigation was performed with the algorithm most well suited[1] for the test.

5.1.1 Fully floating and voltage divider controlled wells

One investigation was into the effectiveness of fully floating nwells versus voltage-divider controlled "floating" nwells. The test was performed with the gIWABB algorithm in NWF mode on the *adder32* circuit. Figure 5.2 compares the yield improvement when using floating wells and voltage-divider controlled wells. The same objective targets are used for both circuit types, but the initial yield varies for the two different circuits. This is due to the added well capacitance from the voltage divider circuitry, and the increase in P_{op} from the voltage divider. It is clear that the floating nwells are not as effective as voltage-divider controlled wells. This is because the floating nwells are floating far from the optimal V_{b_p} voltage determined in section 4.1.3. The floating nwells have an average voltage of about 500mV, whereas the voltage-divider controlled nwells float at about 850mV. This is due to the less than optimal grouping of pfets in the nwells. The nwells in *adder32* are formed based on a bit slice grouping, which gives a perfect division of up switching and down switching pfets. The nwell switching polarity could be modified by adding dummy pfets (as section 4.1.3 explains) which would be able to raise V_{b_p}. However, for the purpose of this thesis, it was concluded that fully floating nwells was not an effective option and was not investigated any further.

5.1.2 Required biasing resolution

Another initial investigation looked into the required biasing resolution for the different biasing modes to be effective. Again using gIWABB, the yield improvement for NWF+NWB+VDD, aDWB, and VDD modes were compared for bias resolutions of 30mV and 100mV. The results are shown in Figure 5.3. The biasing resolution has an enormous effect on modes using V_{dd} biasing. A biasing resolution of 100mV is not sufficient to make any improvement on yield when V_{dd} is used alone.

[1] Mainly with respect to algorithmic speed.

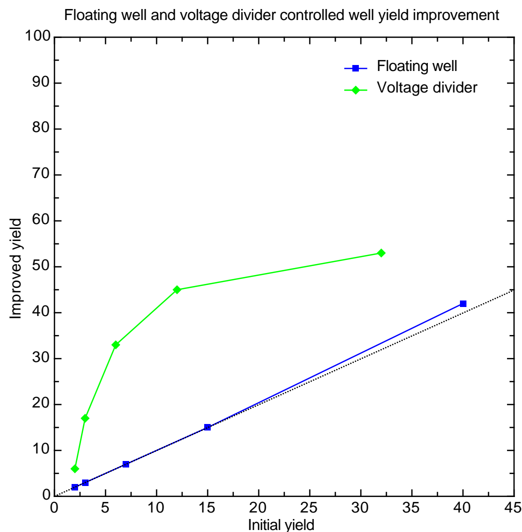


Figure 5.2: Comparison of yield improvement for NWF with floating and voltage-divider controlled nwells using gIWABB.

Although there is less of a difference between the resolutions in the aDWB mode, the 30mV resolution still outperforms the 100mV resolution. It was concluded that a 30mV bias resolution is strongly preferred, especially for V_{dd} biasing. This resolution was used in subsequent investigations.

5.1.3 Use of V_{b_n} biasing

Several different biasing modes using V_{b_n} biasing are listed in Section 4.2. As discussed in that section, these modes would require a triple well process to implement. For completeness, an initial investigation into the possible benefit of adding V_{b_n} biasing was conducted using the gIWABB algorithm on *adder32*. Figure 5.4 shows the yield improvement from this test. This plot shows the NWF+PWB+VDD is more effective at improving the yield than NWF+NWB+VDD at all initial yield points. More significantly, NWF+DWB+VDD mode soars all the other modes. Even at the worst initial yield of 2%, this mode produces a final yield of 75%. It is able to produce final yields at or above 90% on all the higher initial yields. These impressive results have a significant price tag, though. First, and most expensive, is the requirement for a triple well process. This alone makes using NWF+DWB+VDD intangible. On top of that, two adjustable power supplies would be needed, along with their distribution networks. Based on estimates from [19], this would require $\sim 1.7\text{mm}^2$ of silicon area. The use of NWF+DWB+VDD and other modes using V_{b_n} biasing just is not practical due to the expense of implementation. Therefore, these modes were not investigated further.

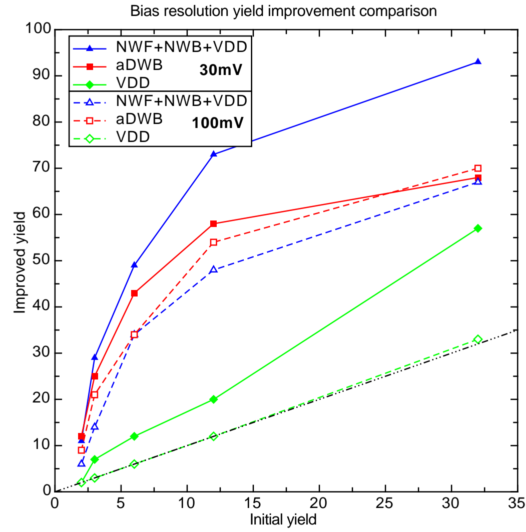


Figure 5.3: Comparison of yield improvement for 30mV and 100mV bias resolutions using gIWABB.

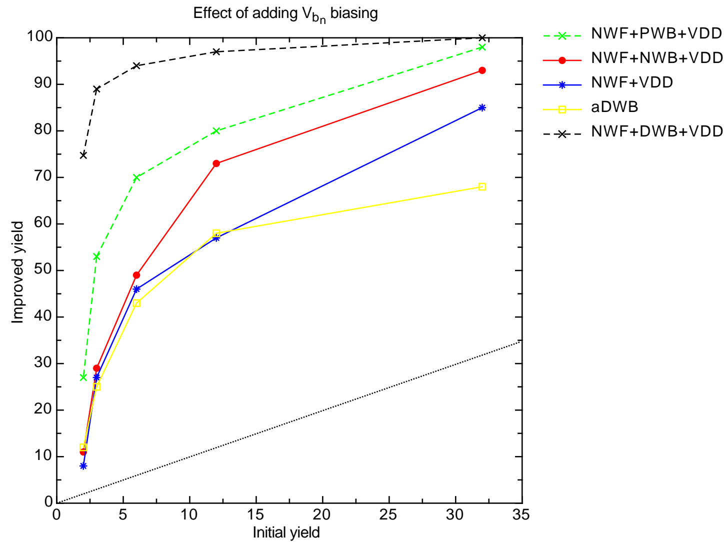


Figure 5.4: Yield improvement from gIWABB on *adder32* using V_{b_n} biasing.

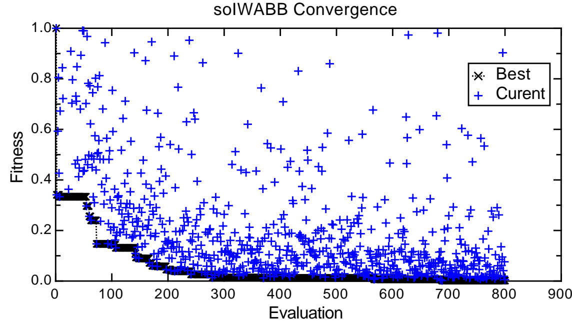


Figure 5.5: Convergence of NWF fitness with soIWABB.

5.2 soIWABB results

5.2.1 Convergence

Convergence testing for soIWABB was done using test chip 101 in NWF mode. This chip was used since results can be easily compared to the known optimum. The algorithm is given target objectives near the optimal solution with 16 floating wells, each with a slightly lower power requirement. This forced the algorithm to explore the parameter space near 16 floating wells. The convergence of the fitness during successive evaluations of the soIWABB algorithm is shown in Figure 5.5. This type of graph is usually associated with learning algorithms and is called a learning curve. It is clear from this learning curve that soIWABB approaches the best fitness in about 400 generations. Figure 5.6 expands on the learning curve by comparing the solutions for 10, 50, 100, 200, 400 and 800 generations with a population size of 30. The actual number of evaluations performed by each run varied due to the caching mechanism. Again, soIWABB is very near to the optimal solution (floating wells 18–32) in 400 generations. The 800 generation run continues to improve the solution slightly, but the added simulation time is not worth the small additional improvement. Based on this information, the soIWABB algorithm ran for 400 generations with a population of 30.

5.2.2 Test chip results

Using the algorithmic configuration determined from the previous investigations (400 generations with a population of 30), soIWABB was run on the four test chips in NWF mode. Even though test chip 101 was already simulated in this configuration, it was repeated. Figure 5.7 shows the results of these simulations. The result for chip 101 is similar to the one found in the convergence testing, but it is not identical due to the random nature of EAs. Notice that chip 101's best solution is very close to the optimal solution. Also, the average of all the evaluations is starting to look like the length deviation curve in Figure 5.1. This curve is desirable because it represents the searched

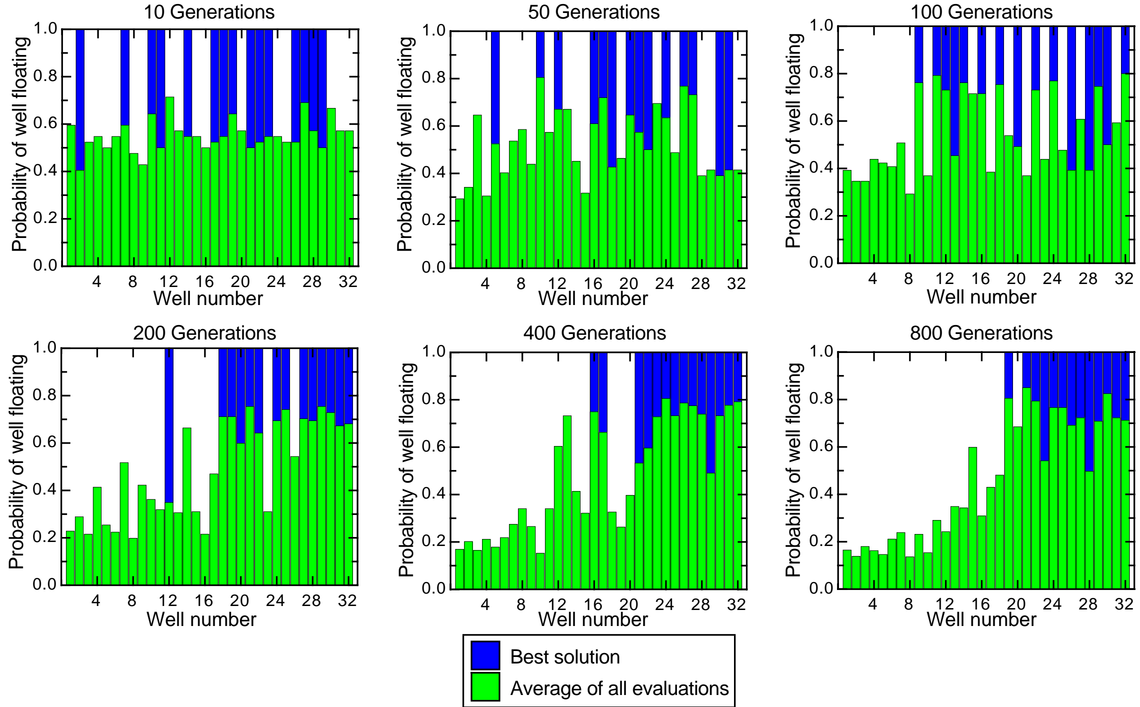


Figure 5.6: Comparison of solutions from NWF convergence testing with soIWABB using 10, 50, 100, 200, 400 and 800 generations with a population size of 30.

parameter space reflecting the problem space. The other three test chips show this as well. Chip 102's best solution is almost the optimal solution wherein the first 16 wells are floated. The best solution of chip 103 has every fourth well floated with a few extra. The best solution for chip 104 is very similar to chip 103 except that the solutions are more closely coupled to every fourth well, thus avoiding the other even numbered wells. Overall, the soIWABB algorithm performed very well in finding near optimal solutions in the NWF mode.

5.2.3 *adder32* results

Figure 5.8 shows the yield improvement results from the soIWABB algorithm. Table 5.1 presents the same data numerically. The NWF+NWB+VDD and NWF+VDD modes are basically even for all the yields, but NWF+NWB+VDD is slightly better. NWB+VDD performed very well in yield 1, improving the yield from 32% to 83%, though it performs worse than DWB in the other yields. NWF+NWB and NWF both give a moderate improvement for the better initial yield points. The VDD mode isn't able to improve any chip beyond the initial yield. In general, the performance of all the modes are similar on yield 5, but the more effective modes quickly improve with increasing initial yield.

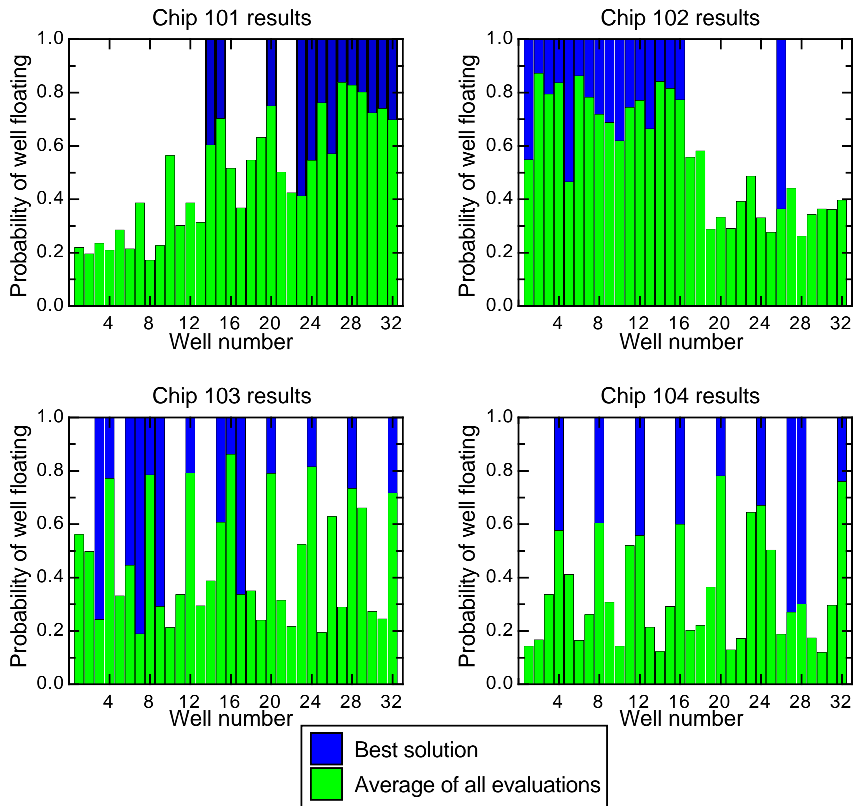


Figure 5.7: soIWABB NWF results for *adder32* test chips 101–104.

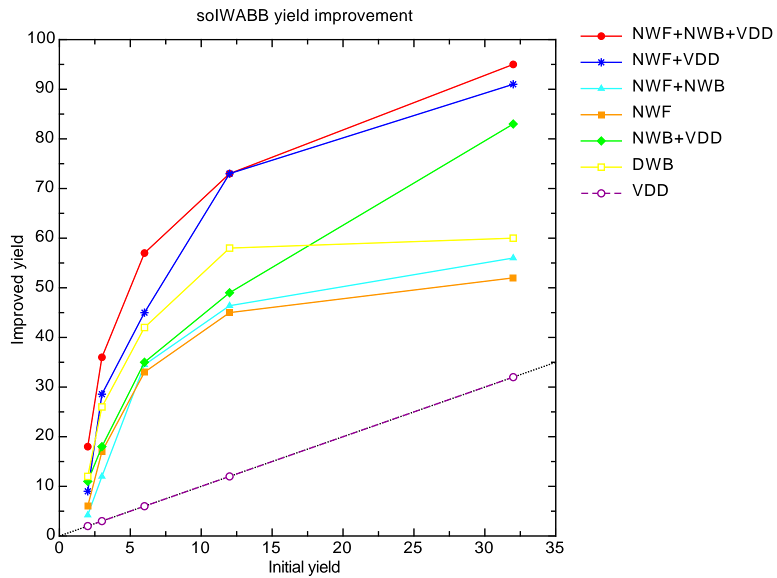


Figure 5.8: soIWABB *adder32* yield improvement plot.

	Yield 1	Yield 2	Yield 3	Yield 4	Yield 5
Normalized f_{min}	0.98	1.00	1.01	1.02	1.03
Normalized P_{max}	1.00	1.00	1.00	1.00	1.00
Initial yield	32.0%	12.0%	6.0%	3.0%	2.0%
NWF+NWB+VDD	95.0%	73.0%	57.0%	36.0%	18.0%
NWF+VDD	91.0%	73.0%	45.0%	28.6%	9.0%
NWF+NWB	56.0%	45.0%	27.0%	12.0%	4.0%
NWF	53.0%	45.0%	33.0%	17.0%	6.0%
NWB+VDD	83.0%	49.0%	35.0%	18.0%	11.0%
DWB	60.0%	58.0%	42.0%	26.0%	12.0%
VDD	32.0%	12.0%	6.0%	3.0%	2.0%

Table 5.1: soIWABB *adder32* yield improvement results.

5.2.4 *adder32* solutions summary

The solutions found by soIWABB for all the biasing modes are summarized in Figures 5.9–5.12. These plots show histograms for the each biasing of the best configurations found for all 100 chips. The acceptable configurations are shown in green. Also, for modes with more than one biasing, two dimensional parameter space plots are shown. Information on how each biasing is used in each mode can be seen in these figures. In each set of graphs, there are 55 chips that are acceptable in their fully connected, normally biased configurations. In the NWF+NWB+VDD results, it can be see that acceptable chips have anywhere from 0–32 floating wells. All acceptable configurations have V_{dd} below the normal, and most have their pfets forward biased. In the NWF+VDD mode, acceptable floating well configurations again range from 0–32 floats with V_{dd} below the normal. With NWF+NWB, the acceptable floating configurations mostly had less than half the wells floated with forward biased pfets. The NWF mode maintains the full range of floats being acceptable. The acceptable chips from the VDD mode only have the initial normal bias. The same trends of lowered V_{dd} and forward biased pfets are seen with NWB+VDD. The DWB mode generally finds acceptable chips by forward biasing nfets and pfets.

The overall trends from all the modes show lowered V_{dd}, forward biased pfets and a wide range of floating wells. The best solutions from the various algorithms are very similar to these results, so these plot are not repeated for each algorithm.

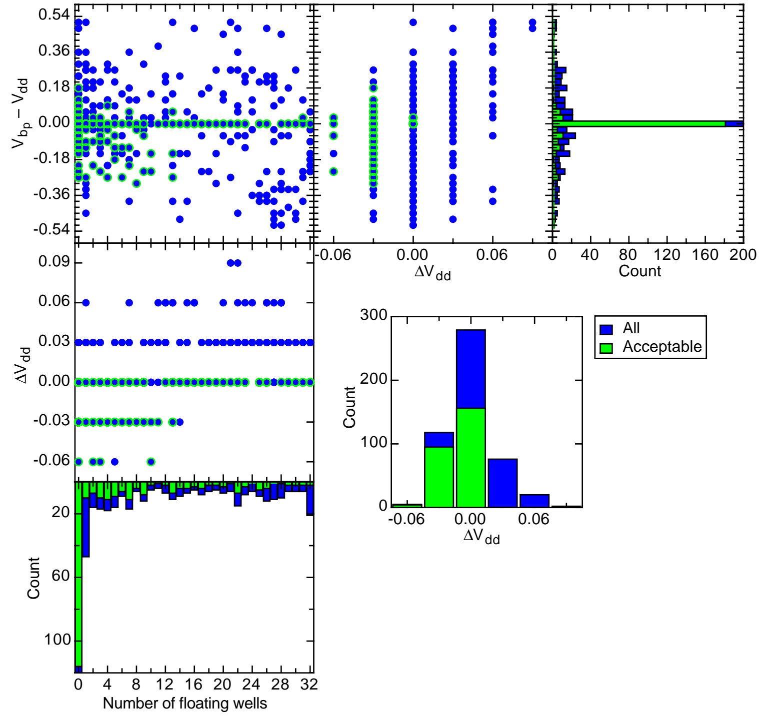


Figure 5.9: Summary of the best configurations found by soIWABB with NWF+NWB+VDD.

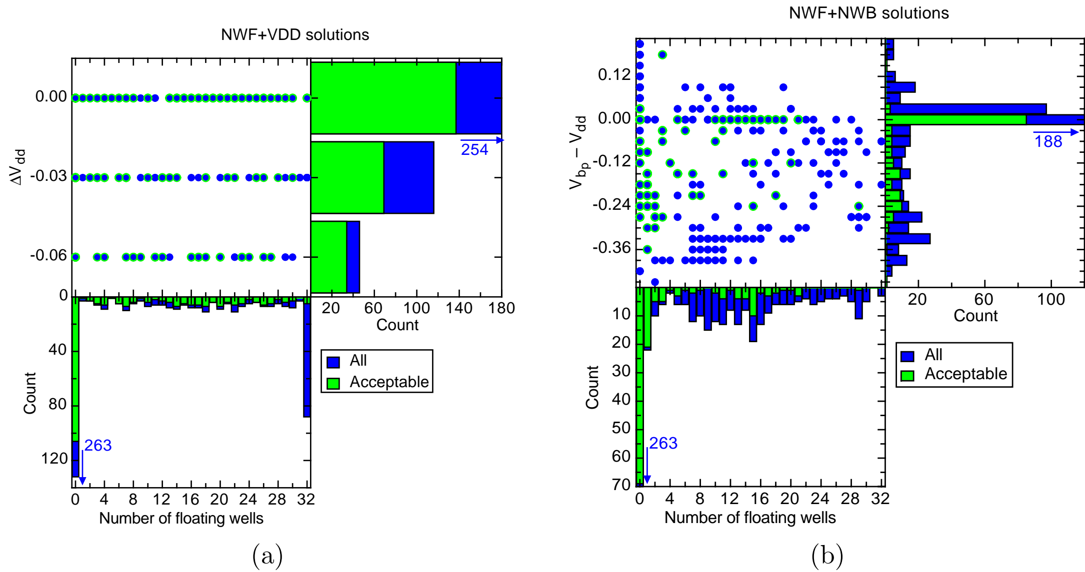


Figure 5.10: Summary of the best configurations found by soIWABB with (a) NWF+VDD and (b) NWF+NWB.

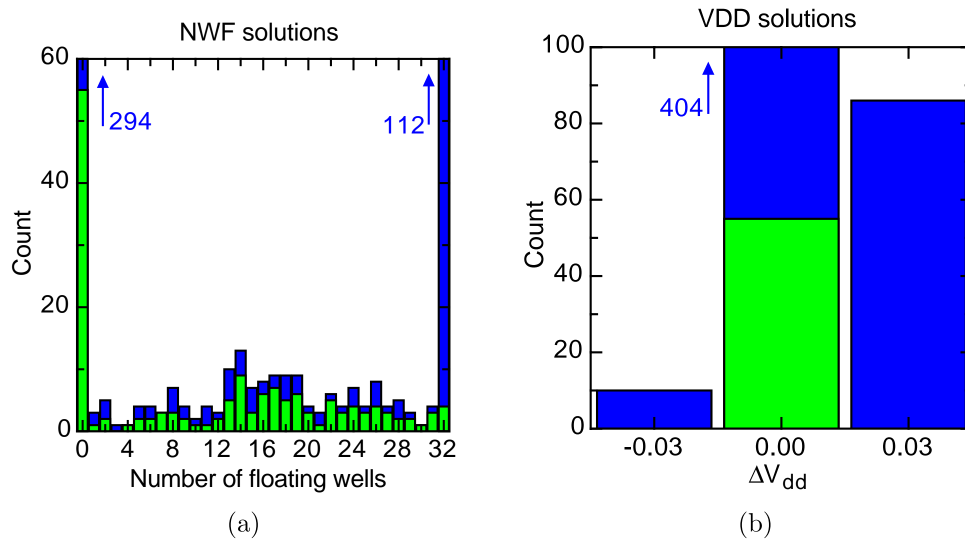


Figure 5.11: Summary of the best configurations found by soIWABB with (a) NWF and (b) VDD.

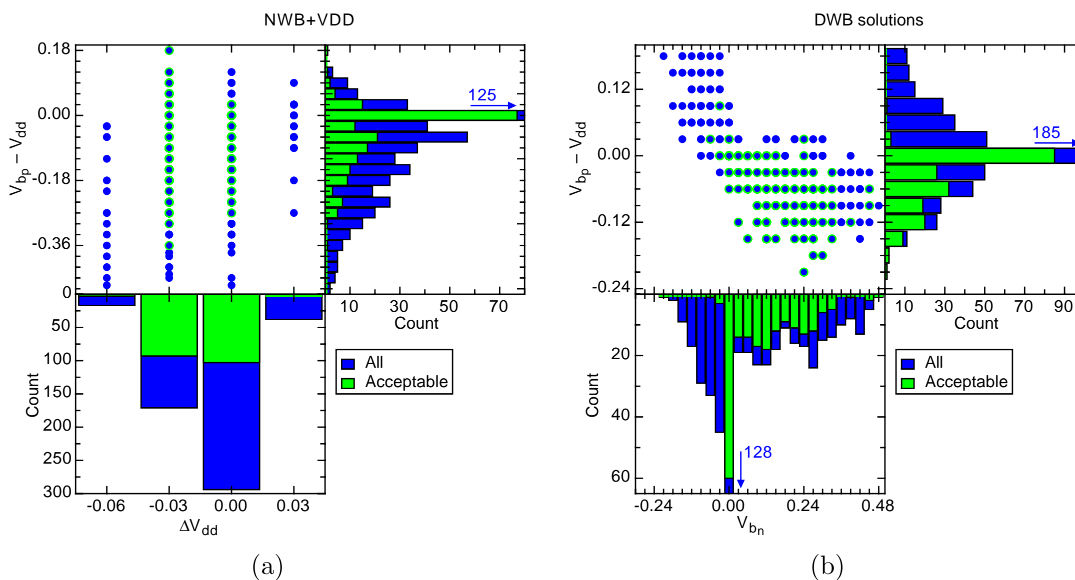


Figure 5.12: Summary of the best configurations found by soIWABB with (a) NWB+VDD and (b) DWB.

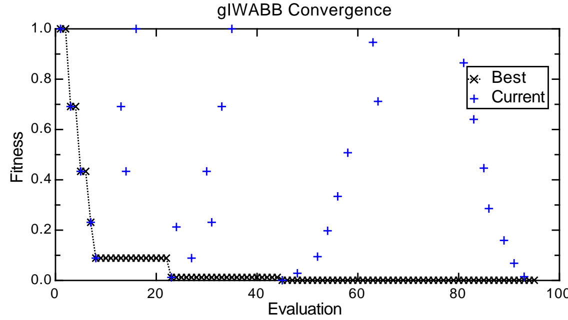


Figure 5.13: Convergence of NWB+VDD fitness with gIWABB.

5.3 gIWABB results

5.3.1 Convergence

The convergence tests for gIWABB were performed in a similar manner as soIWABB. Chip 101 was run in NWF mode for a varying number of allowed steps with the same objective targets. gIWABB's objective-space gradient-estimation of well floating is so effective that it finds the optimal solution immediately after the test evaluations. After the test evaluations, it only evaluates one other worse configuration, if any. Therefore the convergence test for gIWABB needed to be performed in a mode without NWF in order to determine the number of steps required to find an optimal bias parameter configuration. The NWB+VDD mode was chosen for the test. The learning curve of this test is shown in Figure 5.13. The graph shows fitness versus evaluation, not step. Each step in the search counts, but only unique evaluations are shown in the graph in order to facilitate comparison between the algorithms. Since this search is done with a simple local random walk, the fitness wanders around the optimal solution's neighborhood resulting in the current step's fitness wandering away from the best found solution. The best solution is found after about 50 evaluations. This data is taken from the simulation with 800 steps maximum.

As Figure 5.14 shows, the random local search can get lucky and find the solution in as few as 10 steps. This figure shows all evaluations made in both parameter and objective space for varying maximum steps. Even when run with a 10 step maximum, gIWABB was able to find the same solution as the 800 step run found. It is interesting to see how the 400 step run searched a very different neighborhood, but still found the same solution as the other runs. It is also important to point out how important the evaluation cache is in this algorithm. The 800 step run made only 98 unique evaluations, and the 400 and 200 step runs made only slightly fewer. Based on all this information and the runtime versus solution quality trade-offs, 200 steps maximum was the best option for gIWABB.

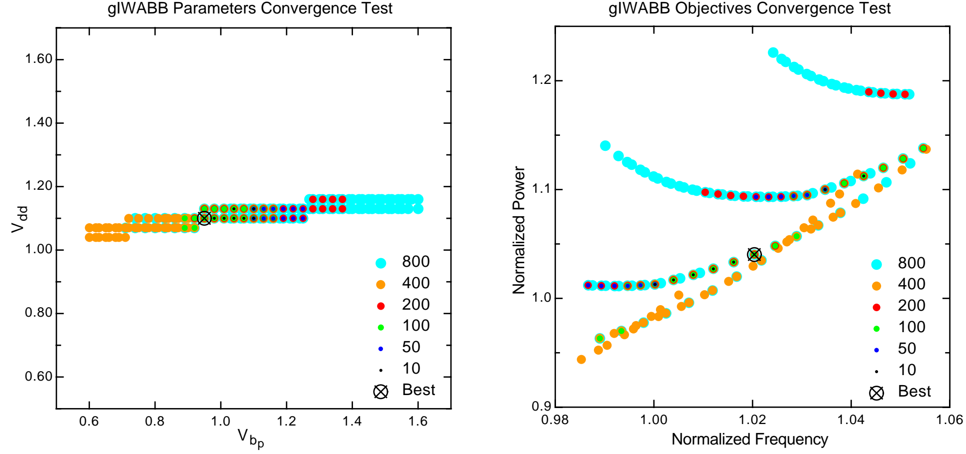


Figure 5.14: Comparison of solutions from NWB+VDD convergence testing with gIWABB using 10, 50, 100, 200, 400 and 800 generations with a population size of 30.

5.3.2 Test chip results

In order to compare the quality of solutions found between the different algorithms, gIWABB was run in NWF mode for the test chips. The solutions are shown in Figure 5.15 where the average is calculated without including the test evaluations used in gradient determination. gIWABB quickly reaches the optimal solution for all test chips, and rarely deviates from it. Chip 103 is the furthest from the optimal due to the the target frequency f_{min} being higher than that of the optimal solution. The algorithm floats wells in sequence until f_{min} is met, making the fitness only the P_{max} violation distance.

5.3.3 *adder32* results

The yield improvement results from gIWABB are shown in Figure 5.16 and summarized in Table 5.2. The NWF+NWB+VDD mode outperformed all other modes, improving the 12% initial yield to 73%. NWF+VDD, NWB+VDD, DWB all performed about the same in yield 2, but DWB performance flattened out in yield 1. NWF+NWB and NWF are about even in third place, with VDD performing the worst. In yields 3 and 4, the algorithms stay in approximately the same order. While in the most difficult yield (yield 5), NWB+VDD, DWB and NWF+NWB+VDD take the top spots with the other modes close behind.

5.3.4 *L64buP* results

The *L64buP* circuit is much more complex and has nearly five times to number of fets as *adder32*. These reasons combined with its heavy use of dynamic logic, make it take significantly longer to

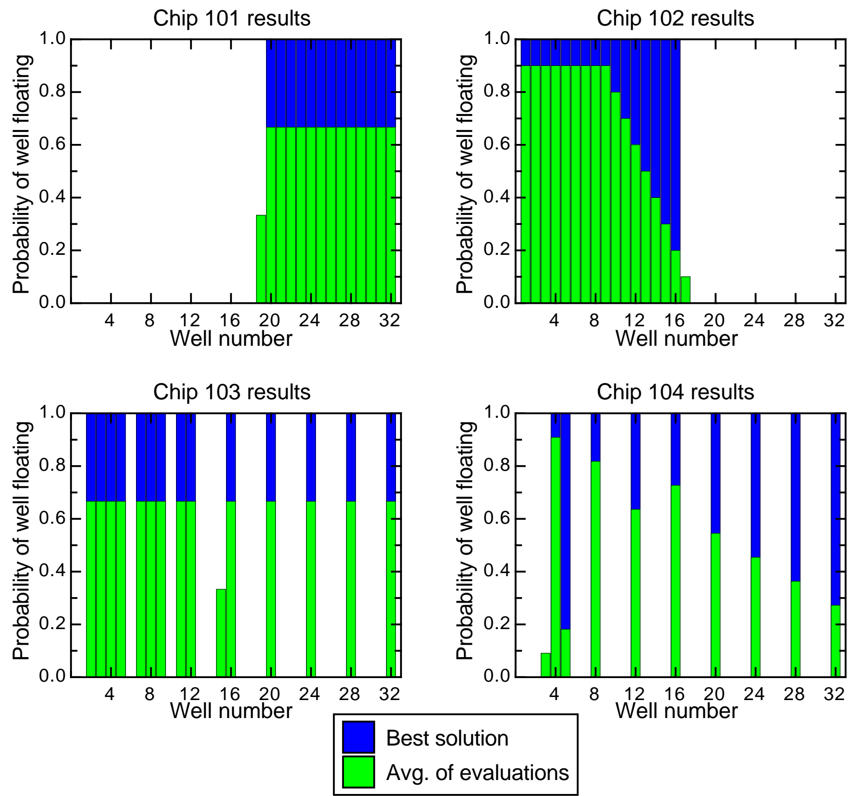


Figure 5.15: gIWABB NWF results for *adder32* test chips 101–104. Average evaluations do not include test evaluations.

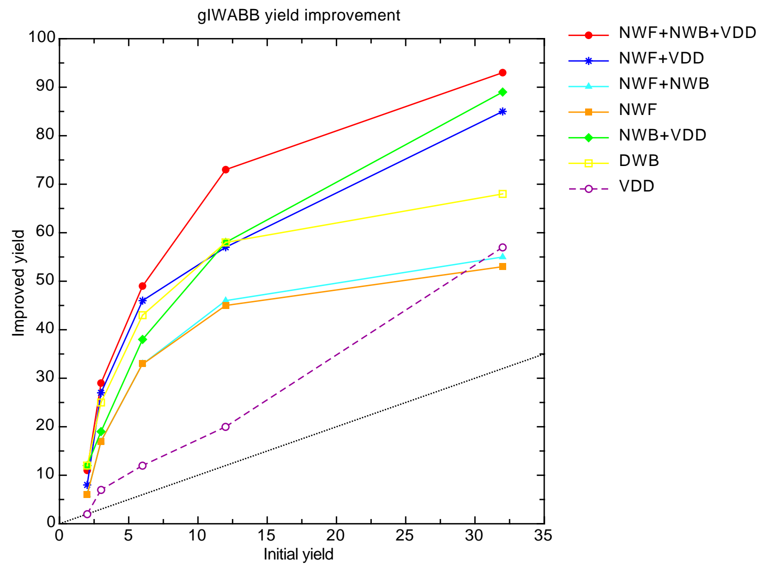


Figure 5.16: gIWABB *adder32* yield improvement plot.

	Yield 1	Yield 2	Yield 3	Yield 4	Yield 5
Normalized f_{min}	0.98	1.00	1.01	1.02	1.03
Normalized P_{max}	1.00	1.00	1.00	1.00	1.00
Initial yield	32.0%	12.0%	6.0%	3.0%	2.0%
NWF+NWB+VDD	93.0%	73.0%	49.0%	29.0%	11.0%
NWF+VDD	85.0%	57.0%	46.0%	27.0%	8.0%
NWF+NWB	55.0%	46.0%	33.0%	17.0%	6.0%
NWF	53.0%	45.0%	33.0%	17.0%	6.0%
NWB+VDD	89.0%	58.0%	38.0%	19.0%	12.0%
DWB	68.0%	58.0%	43.0%	25.0%	12.0%
VDD	57.0%	20.0%	12.0%	7.0%	2.0%

Table 5.2: gIWABB *adder32* yield improvement results.

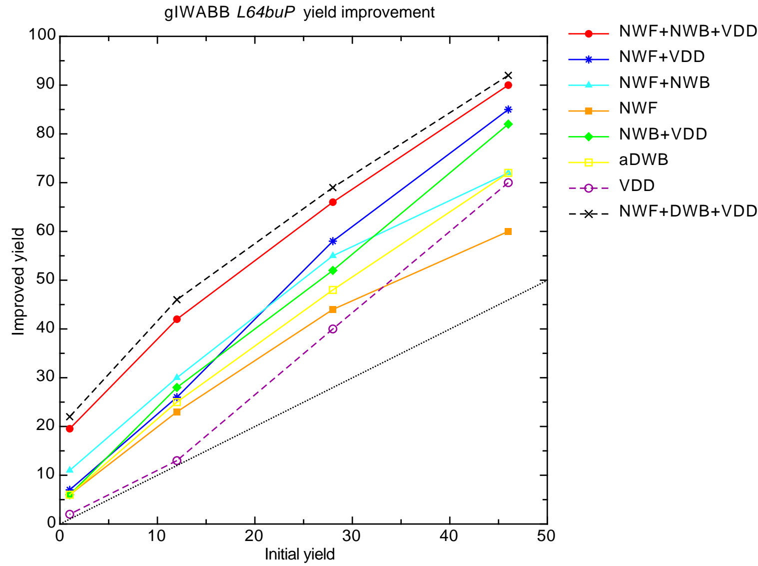


Figure 5.17: gIWABB *L64buP* yield improvement plot.

simulate. Due to time constraints *L64buP* was only run with gIWABB. The results for the other algorithms on this circuit are speculated in Section 6.1.4.

The yield improvement results are shown in Figure 5.17 and Table 5.3. The mostly dynamic structures used in *L64buP* mean most of the circuit is made of nfets. This can make V_{b_p} biasing significantly less effective at changing the operating parameters of the circuit. The NWF+DWB+VDD mode is included to show the effect V_{b_n} biasing can have on this dynamic circuit structure.

Even without using V_{b_n} biasing, gIWABB is able to improve the yield of *L64buP*. The general ordering of the different modes is similar to the *adder32* results. Even though the yield improvements seem significantly lower than those achieved on *adder32*, they are still quite significant. The NWF+NWB+VDD mode is able to nearly double the 46% initial yield, and is able to increase the 12% initial yield to 42%.

	Yield 1	Yield 2	Yield 3	Yield 4
Normalized f_{min}	0.95	1.00	1.05	1.10
Normalized P_{max}	1.00	1.00	1.00	1.00
Initial yield	46.0%	28.0%	12.0%	1.0%
NWF+NWB+VDD	90.0%	66.0%	42.0%	19.5%
NWF+VDD	85.0%	58.0%	26.0%	7.0%
NWF+NWB	72.0%	55.0%	30.0%	11.0%
NWF	60.0%	44.0%	23.0%	6.0%
NWB+VDD	82.0%	52.0%	28.0%	6.0%
DWB	72.0%	48.0%	25.0%	6.0%
VDD	70.0%	40.0%	13.0%	2.0%
NWF+DWB+VDD	92.0%	69.0%	46.0%	22.0%

Table 5.3: gIWABB *L64buP* yield improvement results.

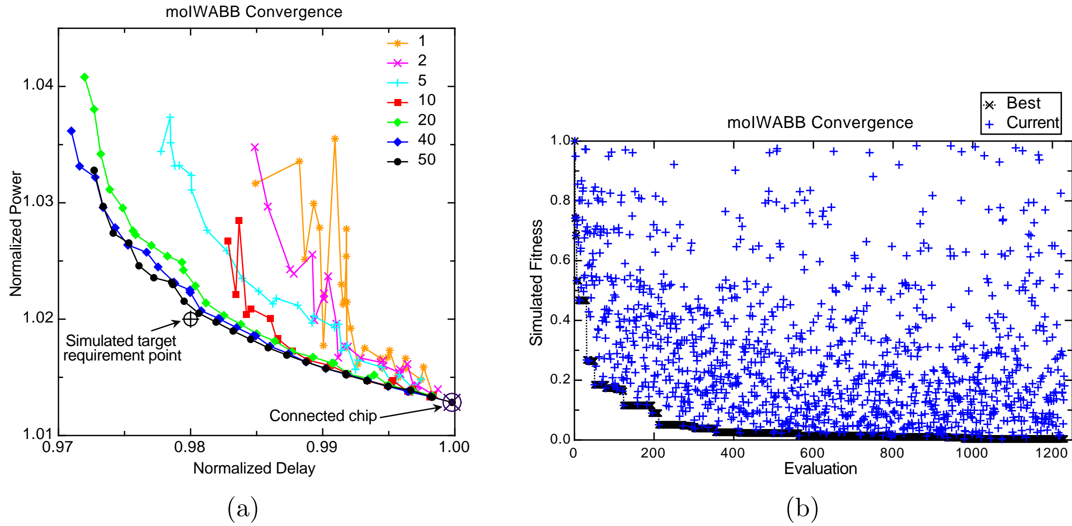


Figure 5.18: (a) Evolution of the pareto front from NWF convergence testing with moIWABB. (b) Convergence of NWF simulated fitness from moIWABB.

5.4 moIWABB results

5.4.1 Convergence

The convergence of moIWABB is more difficult to measure than for soIWABB and gIWABB because it converges to the pareto front instead a single optimal solution. The progression of 30 member pareto fronts from moIWABB in NWF mode on chip 101 with increasing maximum generations is shown in Figure 5.18a. When the algorithm is only allowed one generation after the initial random start, the pareto front is very rough. It quickly smoothes out with more generations, though. After about 40 generations, a very good pareto front is established. With 50 generations, slight progress is made in smoothing the front. The smoothness of the front is important for analysis later in this section.

In order to make a similar learning curve for the convergence of moIWABB, a single simulated fitness value was computed after the simulation. It is based on an unreachable objective target reuirement point shown in Figure 5.18a. The fitness for each individual was the Euclidean distance to this unreachable point. Figure 5.18b shows the learning curve with this simulated fitness, revealing that only small improvements are made after about 800 evaluations. This number of evaluations is usually reached in about 25 generations with a population of 30.

A comparison of the solutions found by moIWABB for each maximum generation is shown in Figure 5.19. Again, a very good solution is found between 20 and 40 generations. The average of all the evaluations is nearing convergence with the best solution at 40 generations and is very good at 50 generations. Based on these plots, 40 generations should be enough to find good solutions.

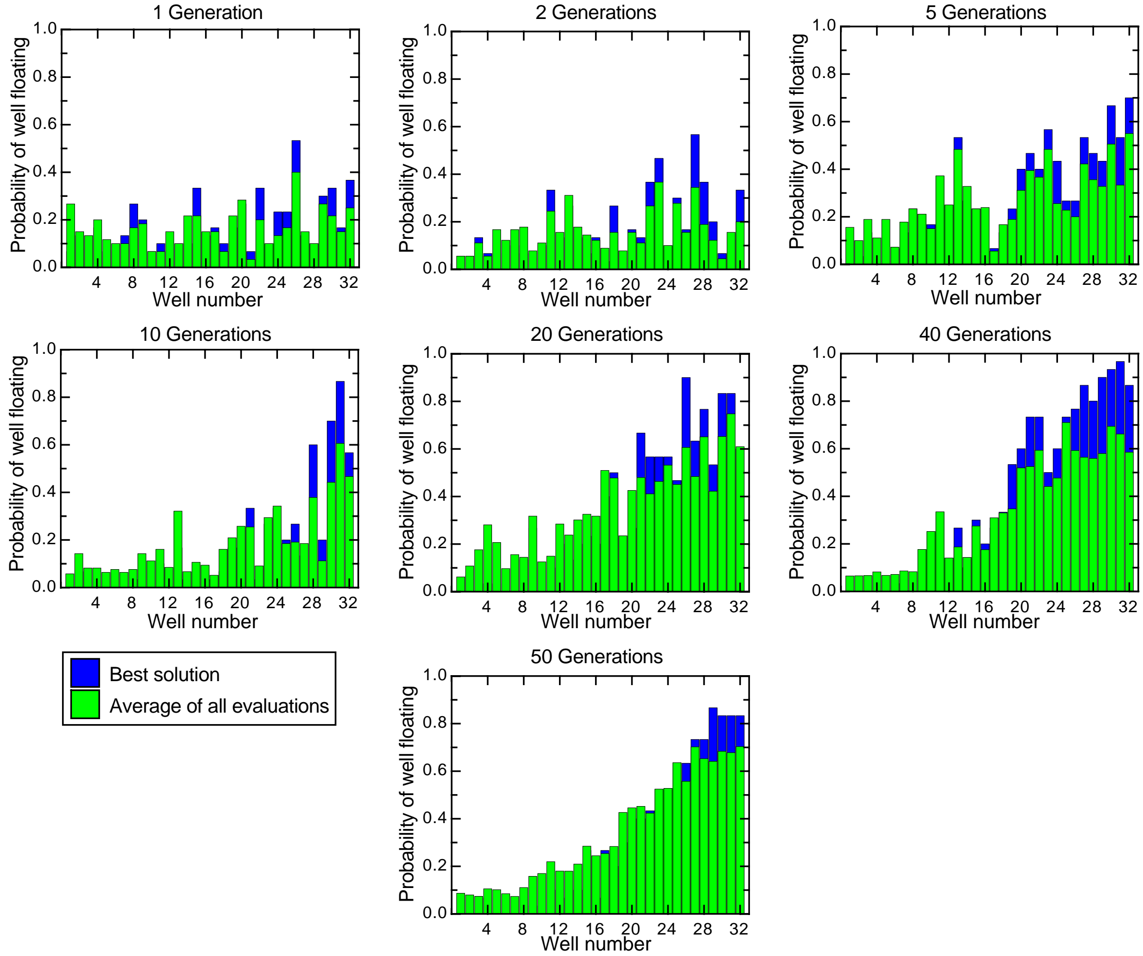


Figure 5.19: Comparison of solutions from NWF convergence testing with moIWABB using 1, 2, 5, 10, 20, 40 and 50 generations with a population size of 30.

However, as stated before, problems in later analysis can be caused by non-smoothness of the pareto front. For this reason, 50 generations with a population size of 30 were used in moIWABB testing.

5.4.2 Test chip results

The solutions for test chips 101–104 from the NWF mode of moIWABB are shown in Figure 5.20. As with the other algorithms, very good solutions were found. The advantage of the pareto front solution is shown here by the distribution of floating wells in the best solutions approximating the length deviations of the test chips. The average of all evaluations also converge to very near the final pareto population for all chips.

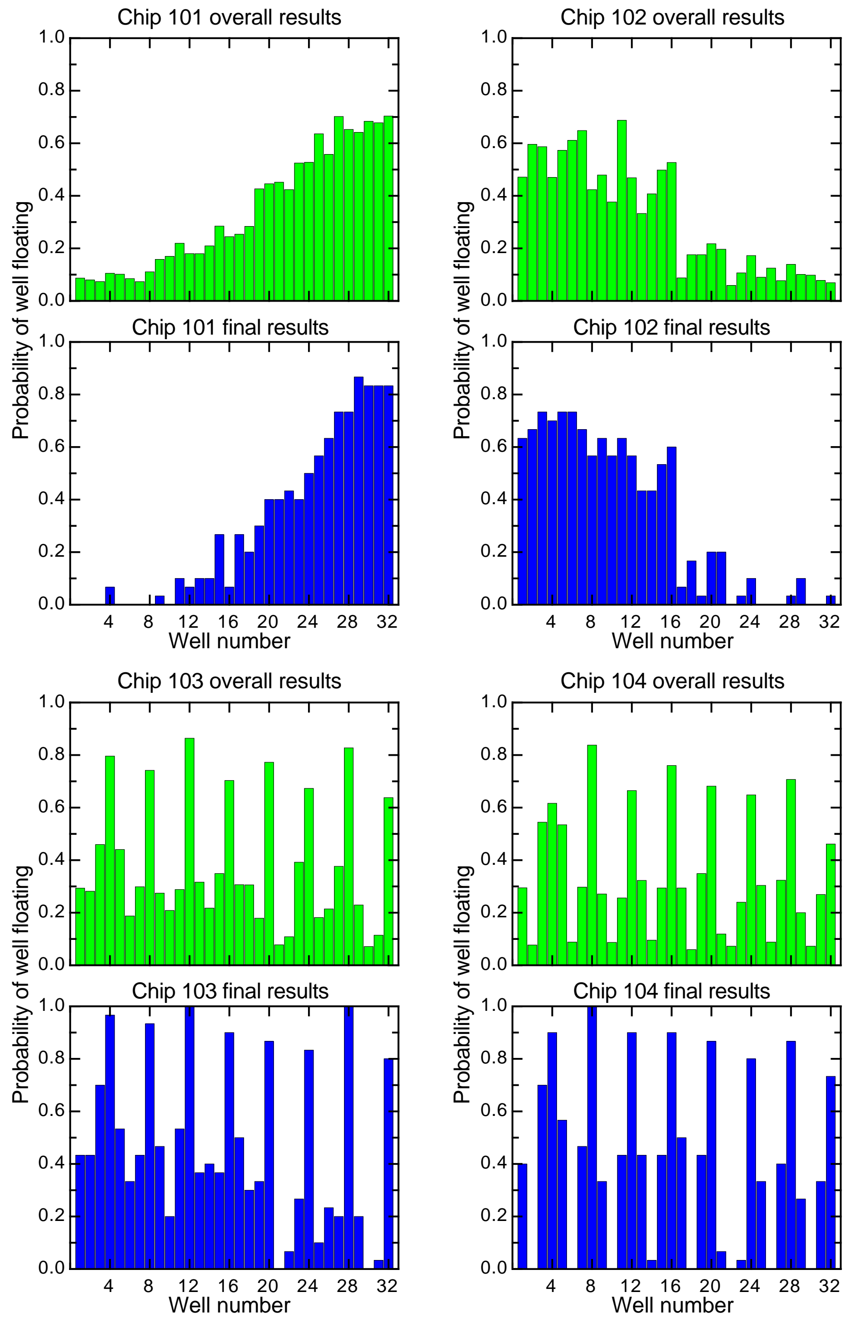


Figure 5.20: moIWABB NWF results for *adder32* test chips 101–104.

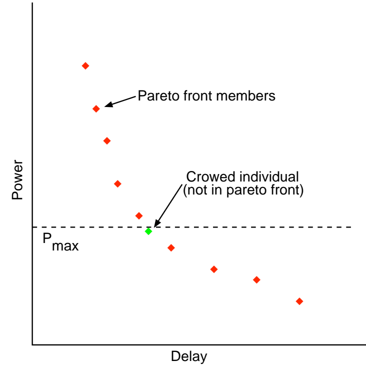


Figure 5.21: Illustration of falsely low yield from moIWABB.

5.4.3 *adder32* results

The moIWABB algorithm presents an analysis challenge in determining a yield from the pareto front. Given a set of objective points, the yield can be calculated by finding the individual in the pareto front with the smallest delay while still being under P_{max}. If this individual's delay is less than $1/f_{min}$, then the chip is good and increments the yield. If the delay is not less than $1/f_{min}$ or there is no individual in the pareto front below the power limit, then the chip is bad. This yield calculation is very straight forward. The yield it returns is guaranteed to be possible since all the good chip configurations are present in the pareto front. This is similar to the yield calculations in soIWABB and gIWABB in that the yield is guaranteed since the chip configurations that produce the yield are reported as solutions from the algorithm.

One feature in moIWABB is the crowding reduction in objective space. This feature tries to move individuals away from each other, and can cause the final yield calculated in this manner to be lower than what is actually possible. Figure 5.21 illustrates how this is possible. The red points are the pareto front in objective space resulting from a moIWABB run. The green point is an individual that was not included in the pareto front due to crowding. The green individual could be one that was evaluated and explicitly not included, or one that was never evaluated because the search was never driven in that direction due to the near-by pareto front member. Whatever the case, the difference in delay from the green point to the first red point below P_{max} can be substantial. This delay difference can cause the yield calculated based on members of the pareto front to be lower than what might be possible in a more directed search.

In order to get an estimate for the best yield possible from the pareto front, simple interpolation and extrapolation based on the front's members can be used. This requires a very smooth pareto front since it relies on the secant method for interpolation. Figure 5.22 shows the pareto front

	Yield 1	Yield 2	Yield 3	Yield 4	Yield 5
Normalized f_{min}	0.98	1.00	1.01	1.02	1.03
Normalized P_{max}	1.00	1.00	1.00	1.00	1.00
Initial yield	32.0%	12.0%	6.0%	3.0%	2.0%
NWF+NWB+VDD	96%/98%	77%/90%	54%/72%	34%/48%	19%/26%
NWF+VDD	99%/99%	79%/91%	52%/71%	36%/47%	18%/27%
NWF+NWB	62%/69%	54%/58%	43%/47%	26%/31%	12%/14%
NWF	52%/72%	43%/55%	28%/42%	16%/20%	5%/8%
NWB+VDD	51%/91%	21%/66%	12%/40%	6%/21%	2%/11%
DWB	71%/71%	59%/64%	38%/46%	26%/29%	12%/13%
VDD	62%/84%	28%/49%	14%/30%	6%/14%	2%/7%

Table 5.4: moIWABB *adder32* minimum/maximum yield improvement results.

members (blue) from all 100 *adder32* chips with there interpolated pareto front lines (black) and best delay points for a given P_{max} (green). A yield can be calculated based on these green points. This tends to overestimate the yield because the interpolation assumes a continuous objective space between the members of the pareto front. However, the parameter space, and thus the objective space, is quantized by the biasing resolution and the binary well control.

These two different methods of calculating the yield from moIWABB give an absolute minimum yield and an estimate of the maximum yield. Figure 5.23 and Table 5.4 show the results for all modes. NWF+NWB+VDD and NWF+VDD both perform very well; increasing the 12% yield to at least 77% and at most 91%. All the modes follow similar trends to those observed from the soIWABB and gIWABB results. There tends to be a wider spread between the minimum and maximum yields for the lower performing modes that use V_{dd} biasing. This is caused by the interpolation errors discussed earlier, and can be clearly seen in the VDD and NWB+VDD plots in Figure 5.22.

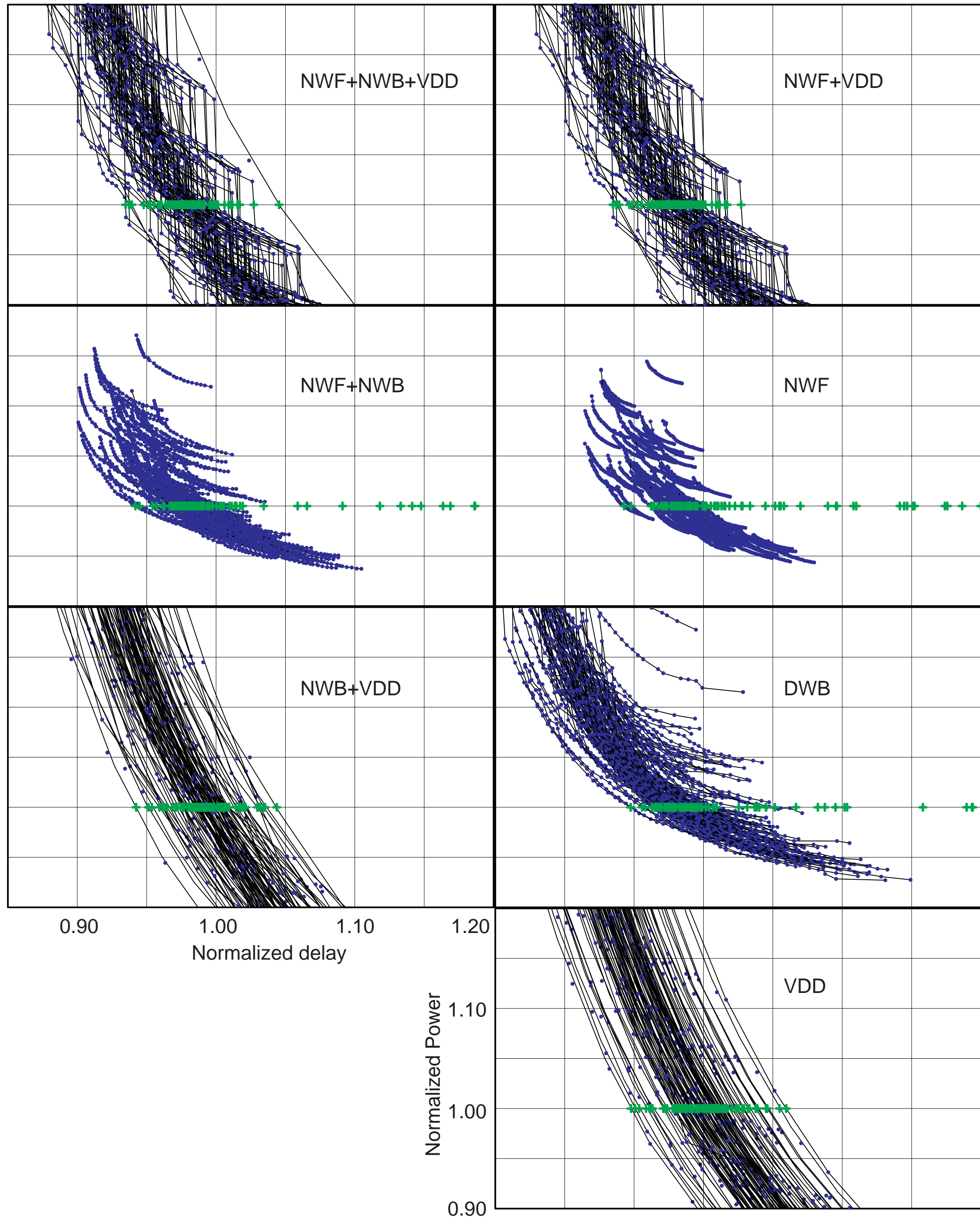


Figure 5.22: moIWABB *adder32* results with interpolated best delay points for a give P_{max}.

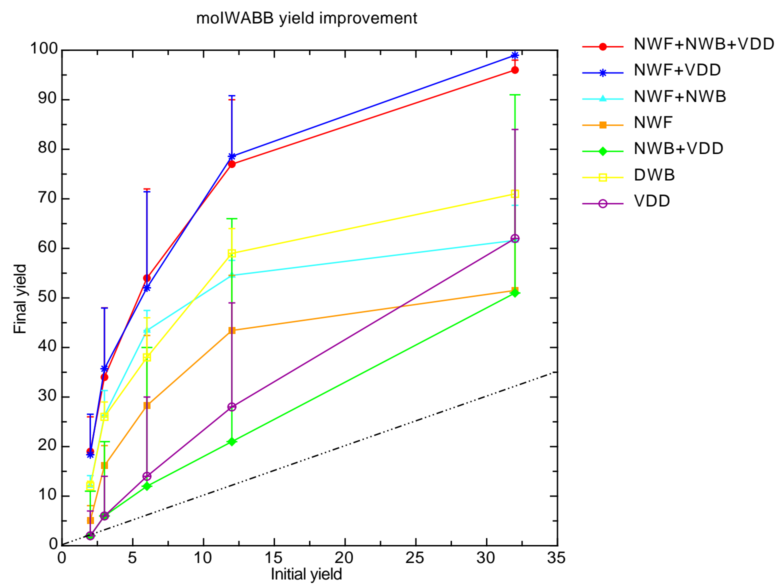


Figure 5.23: moIWABB *adder32* yield improvement plot showing minimum improvement with error bars to maximum improvement.

Chapter 6

Discussion and Conclusion

6.1 Analysis

Investigating the effectiveness of IWABB is the primary goal of this thesis. To accomplish this analysis, three different algorithms have been designed to optimize the IWABB configurations for several modes with two test circuits. IWABB proved to be significantly less effective on the *L64buP* circuit than on *adder32*. This can be explained by three factors. First, the inclusion of large driver fets in some of the wells makes forward biasing those wells cause a substantial increase in P_{op}. Other pfets in the same well are in the critical path, so the performance of the circuit would greatly benefit from such a forward bias. This causes the algorithm to tend to do nothing to those wells, which does not improve the yield. Second, *L64buP* has over four times the number of pfets, but only one additional nwell. This give IWABB significantly less granularity in addressing WID variations. Finally, the dynamic structures in *L64buP* generally have a lower stacking factor than the static structures in *adder32*. This makes the increased leakage current caused by forward biasing have a greater effect on P_{op}. This final problem is inherent in dynamic circuits, and makes the value of dynamic circuit structures in sub-100nm process technologies questionable. Dynamic circuits have significantly higher power consumption than static circuits due to higher dynamic and static power. Dynamic power consumption is higher since all clocked nodes switch on every clock cycle. Static power is higher due to the lower stacking factor throughout the dynamic nodes. Also the performance advantage of dynamic circuit structures is very slight in processes where wire delay is more prominent than gate switching delay[50]. Due to these factors, the use of dynamic circuit structures are giving way to static and other logic styles in smaller process technologies. This shift will tend to make IWABB more effective.

Analyzing the results from the three algorithms starts with establishing their effectiveness. An effective algorithm is able to find a near optimal solution, which is necessary to show the effectiveness

of IWABB. Each algorithm's efficiency is also an important factor since efficiency is directly related to the cost of implementing IWABB. Then the different IWABB modes can be compared, and finally the overall value of IWABB can be determined.

6.1.1 Algorithm effectiveness

The different IWABB algorithms operate on very different methodologies. The effectiveness of each algorithm can be based on its ability to find a near optimal solution, as well as its ability to make yield improvements. The test chip results show that all the algorithms are able to find near optimal solutions on the test chips with NWF. gIWABB had an advantage in this test as it is the only one that calculates accurate per well gradient information. This allowed it to move quickly to the optimal NWF configuration. soIWABB and moIWABB are both able to produce similar quality solutions, but their more random nature causes them to have slight errors. moIWABB has a distinct advantage over the other algorithms in that it returns an entire set of solutions. This allows it to approximate not only the optimal solution, but the optimal solution's neighborhood. The test chip results show that all the algorithms are effective in finding near optimal solutions in the NWF mode.

gIWABB is the only algorithm that has a fundamentally different method to solving NWF than the other biasing methods. The convergence test shows that gIWABB is able to successfully traverse the search space and find a good solution in NWB+VDD mode, as well as NWF. The methodology behind the gIWABB algorithm is also thoroughly tested in other research. There is no major fundamental differences between NWF and the other modes for the other two algorithms. It can be concluded that all the algorithms can effectively find near optimal solutions for all the different biasing modes.

The known optimal solutions to the test chips provide an easy method of establishing abilities of the different algorithms. Unfortunately, there is no easy baseline against which the effectiveness of the algorithms can be compared. This is the one purpose for including the DWB mode. DWB-like methods have been proposed in other research, as discussed in Chapter 2. The effectiveness of each algorithm can determined based on its ability to perform similar yield improvements in these modes to those in other research. The simple ABB (S-ABB) scheme presented in [19] is very similar to the DWB mode. [19] reports an improvement of acceptable dies from 50% to 80%[1]. Extrapolating the DWB results for each algorithm to an initial yield of 50% would show a final yield near 82%. This is consistent with [19], and we can conclude that the algorithms can effectively find good solutions.

[1]Estimated from figures.

Algorithm	Avg. Evals
soIWABB	63.1
gIWABB	34.2
moIWABB	92.4

Table 6.1: Evaluations made per chip averaged over all modes and yields for each algorithm.

6.1.2 Algorithm efficiency

The best measure of an algorithm's efficiency is the number of evaluations it needs in order to reach a good solution compared to the overall size of the search space. Table 6.1 shows the number of evaluations each algorithm made per chip, averaged over all the modes and yields. Considering that the search space of just NWF contains 2^{32} possible individuals, all of these numbers are quite small. The large number for moIWABB is misleading because an infinite number of different yields could be extracted from its solutions, but only five are counted in the averaging. Based on this data, gIWABB is the most efficient, followed by soIWABB and moIWABB, successively.

It is important to consider how these algorithmic efficiencies will scale with problem size, p. This problem space in this application can be generalized to the NWF combinations. This space grows exponentially with the number of wells, n ($p \propto e^n$). It is generally accepted that EAs can scale linearly, $O(p)$, with problem size[51; 52]. This is because the population size needs to scale as $O(\sqrt{p})$ in order to maintain genetic diversity in the larger chromosome. Also, the number of generations needed for the population to converge scales as $O(\sqrt{p})$. Each of these depends on several parameters such as population size, selection mechanisms and mutation rates. However, with the heuristics used in soIWABB and moIWABB, we conjecture they will scale linearly if not sub-linearly. The local random walk is only marginally worse than this linear scaling, as discussed in Section 3.2.1. The gradient calculations in gIWABB scale linearly with the number of nwells; logarithmically with problem size ($n \propto \ln p$). So overall gIWABB scales at least $O(p \ln p)$. Even though gIWABB is shown to use the fewest evaluations in Table 6.1, it may use the most for a larger circuit. For example, consider a circuit with 10^4 nwells, operating near 1GHz. Assuming it takes 10^6 test vectors to establish f_{op} and P_{op}, the gIWABB algorithm would require 10s to establish the nwell gradients and start searching. This can be a substantial overhead. The EA based algorithms will most likely be more efficient on extremely large problems since they do not have this overhead.

6.1.3 Mode effectiveness

With the establishment of the algorithms' ability to find near optimal solutions, regardless of the mode, the effectiveness of each mode can be analyzed. The best analysis is based on the yield

improvement each mode is able to produce. Yield improvement is the primary goal of this research and is available from each algorithm and mode. The moIWABB algorithm allows for a secondary analysis of mode effectiveness based on the objective-space slope each mode is capable of producing. This slope can be viewed as a power-performance trade-off indication.

6.1.3.1 Yield improvement metric

Yield improvement measures the ability of a mode to increase the yield from a given baseline yield. Each mode can be compared relative to all other modes, but there are also several other baselines available. The use of methods similar to VDD is well established in industry, so it can serve as a good baseline for yield improvement. DWB-like methods have also been proposed and thus can serve as a baseline. Of course, many manufactures do no post-fabrication yield enhancement, so the initial yield can be a valuable baseline as well. The trouble with using VDD or DWB for a baseline is that different algorithms performed differently using these modes. Using one of these as a baseline would favor algorithms that performed poorly in the baseline mode. Therefore, the initial yield is the best baseline for comparison of mode effectiveness based on yield improvement.

The difference between the final yield (y_f) and the initial yield (y_i), averaged across all the different yield points, for each mode is shown in Table 6.2. The final yield for moIWABB is the average between the absolute minimum and estimated maximum final yields. The best mode for each algorithm is in bold. For soIWABB and gIWABB, NWF+NWB+VDD is the best mode, beating the next nearest mode (NWF+VDD) by at least 6%. However, this is reversed in moIWABB, and NWF+VDD bests NWF+NWB+VDD by 8%. The best mode in all the algorithms is able to improve the yield by about 50%. With all the algorithms, NWB+VDD and DWB are close third and fourth place, being able to effect about a 30% increase in yield. With the exception of VDD, all the modes are able to produce at least a 20% increase. VDD's effectiveness is very dependent on the algorithm used. With soIWABB, VDD is not able to improve the yield at all. With gIWABB and moIWABB it is able to produce a 8.6% and 18.6% improvement, respectively. moIWABB's impressive performance with VDD is skewed by the falsely high maximum yield produced by interpolation errors. A more reasonable estimate of moIWABB's VDD improvement is the one based off its minimum final yield. This improvement would be 11.4%. Overall, the modes that incorporate both NWF and biasing perform significantly than modes that use only NWF or biasing.

There is a discrepancy in the moIWABB numbers in Table 6.2. The NWF+VDD mode out-performed the NWF+NWB+VDD mode. NWF+NWB+VDD should perform at least as well as NWF+VDD since it has the same biasing options with the addition of NWB. This can be explained

	$\overline{y_f - y_i}$		
Mode	soIWABB	gIWABB	moIWABB
NWF+NWB+VDD	**44.8%**	**45.0%**	42.9%
NWF+VDD	38.3%	33.6%	**50.9%**
NWF+NWB	17.8%	20.4%	30.7%
NWF	19.8%	19.8%	23.2%
NWB+VDD	28.2%	32.2%	21.1%
DWB	28.6%	30.2%	31.9%
VDD	0%	8.6%	18.6%

Table 6.2: Difference between final and initial yields averaged over all initial yields for all biasing modes.

by both the randomness involved in the algorithms and the averaging of the minimum and maximum yield improvements. Examining Figure 5.23 shows that the two modes perform nearly identically across all initial yields.

The NWF+NWB+VDD and NWF+VDD modes perform nearly as well as one another. However, the cost associated with implementing each one is rather different. Both methods need a local voltage divider and pullup pfet in each well controlled by IWABB. This adds a small amount of silicon area to each well. In the test circuits used in this thesis, the total additional silicon area used per well by these pfets was about $1\mu\text{m}^2$. Both modes also require a control signal to each well which adds signal routing costs. To generate these control signals, the addition of scan or fuse circuitry would be required. Most modern microprocessors already use an adjustable supply and power grid to generate V_{dd}, so this presents no addition cost. The NWF+NWB+VDD mode needs an additional power supply and power distribution grid to generate V_{b_p}, though. Granted, the current though this grid would be exceptionally low since it would only be required to charge the nwells. This would allow the grid to be composed of minimum width wires, and it would not require the arduous design which goes into chip-wide V_{dd} grid. However, the routing expense of adding even a minimum width power grid is not worth the small additional yield NWF+NWB+VDD provides over NWF+VDD. Taking into account the cost of implementing each mode combined with their effectiveness, NWF+VDD is the best option.

6.1.3.2 Objective-space slope metric

The objective-space slope metric reinforces the conclusions made based on the yield improvement metric. The objective-space slope is based on only the moIWABB results. Since moIWABB produces a pareto front which represents an approximation to objective-space trade-off curve, its results can be used as an approximation to the best possible solutions for each mode. The interpolated best $1/f_{op}$ point for a given power limit provides an estimate of the best solution each mode can find.

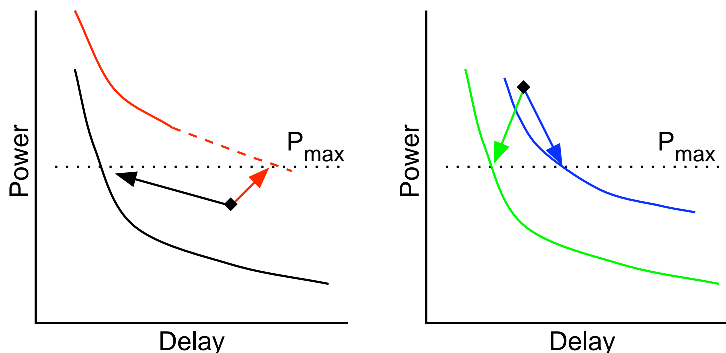


Figure 6.1: Illustration of possible objective-space slopes from moIWABB solutions.

Combining this best solution with the initial $1/f_{op}$ and P_{op} of each chip, an objective-space slope can be calculated.

Figure 6.1 illustrates the different possible slopes. Each point represents a chip's initial position in objective space and the curves are possible pareto fronts for that chip. The arrows show the direction to each interpolated best solution. The slope of each arrow is slope used in this metric. The desired slope depends on a chip's initial P_{op}. It is desirable for chips initially below P_{max} to have shallow negative slopes (black arrow). This would mean the biasing mode is able to improve the performance with only a slight increase in power. Any positive slope (red arrow) is undesirable since it would mean a decrease in performance and an increase in power. This is only possible in cases where the pareto front is extrapolated to P_{max}. The opposite is true for chips initially above P_{max}. It is desirable to have a positive slope (green arrow), since it would indicate an improvement in power and performance. Steeper negative slopes (blue curve) are more desirable than shallow ones since they represent better power-performance trade-off.

A summary of the slopes for chips initially below P_{max} is shown in Figure 6.2. Shorter bars are more desirable in both graphs. The average of all negative slopes is shown for each mode. Shorter bars indicate more desirable shallower slopes. Modes with shallower slopes generally have a better power-performance trade-off. This graph supports the previous conclusion that modes with NWF are more effective. Also shown is the number of chips which exhibit a positive slope. NWF+NWB+VDD and NWF+VDD have significantly fewer positive slopes compared to the other modes. Therefore, these modes are more effective since positive slopes indicate decreasing performance and increasing power.

Figure 6.3 shows the summary of slopes for chips initially above P_{max}. Longer bars are more desirable here. In the graph of average negative slopes, NWF+NWB+VDD, NWF+VDD and

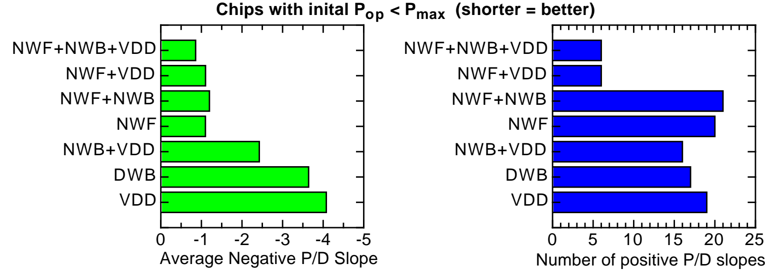


Figure 6.2: Summary of objective-space slopes of all chips initially below P_{max} for each mode from the moIWABB algorithm.

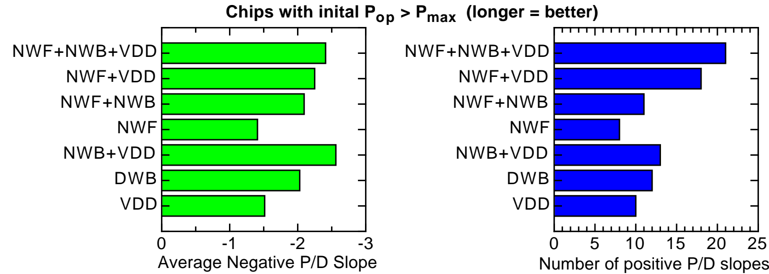


Figure 6.3: Summary of objective-space slopes of all chips initially above P_{max} for each mode from the moIWABB algorithm.

NWB+VDD have the longest bars. This means these modes have the steepest slopes, and thus the best power-performance trade-off. The graph showing the number of positive slopes shows NWF+NWB+VDD and NWF+VDD are the most desirable since they have the most highest counts.

Overall, the analysis of objective-space slopes indicates NWF+NWB+VDD and NWF+VDD have the best power-performance trade-off. This supports the conclusion based on yield improvement that these modes are the most effective. Part of the effectiveness of these modes can be attributed to their better power-performance trade-off characteristics.

6.1.4 IWABB value

The three different algorithms used to test IWABB all showed similar trends across the different modes for *adder32*. Based on this, it is expected the algorithms would perform similar to gIWABB on *L64buP*. The trends for the different modes represent the value IWABB can have in a manufacturing process. Modes incorporation NWF and a biasing are able to effect a substantial yield improvement. This improvement comes from squeezing the distribution of operating parameters. Figure 6.4 shows an example of a final distribution from gIWABB with the NWF+NWB+VDD in yield 1. It shows IWABB's ability to reduce P_{op} on some chips and raise f_{op} on other chips. Plot of resulting distributions using the other algorithms, modes or yields strongly parallel this one.

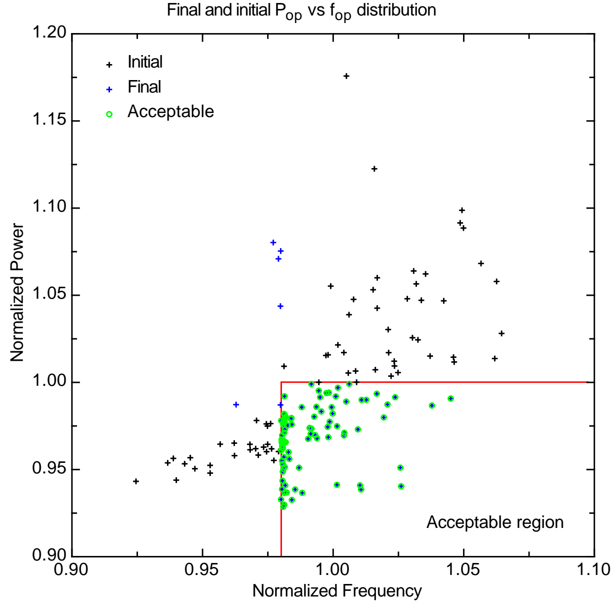


Figure 6.4: Example of initial and final chip distributions in objective space for *adder32*.

Based on the substantial yield improvement and the squeezing of the P_{op} and f_{op} distributions, it can be concluded that IWABB can be an effective tool in mitigating the effects of process variations.

6.2 Conclusion

With ever shrinking critical dimensions in the semiconductor industry, the relative magnitude of process variations will continue to increase. The wider distributions in operating parameters these variations cause will cause continued reduction in manufacturing yields. This thesis has presented a new method to reclaim this lost yield during post-fabrication testing by reducing circuit leakage power and improving performance. The method uses an adaptive body biasing applied on a individual well basis (IWABB). When considering implementation cost and effectiveness, the best IWABB method was found to be NWF+VDD. This method uses V_{dd} biasing and chooses between a locally generated nwell voltage or V_{dd}. The locally generated nwell voltage is from a voltage divider in each nwell. The NWF+VDD method only requires the addition of a single control signal to each well and scan or fuse circuitry to generate the signal. Simulations show that this method is able to improve a 12% initial yield to at least 79%. The final yield could be as high as 91%.

Optimization of the floating nwells and V_{dd} biasing would be performed during post-fabrication testing. Three algorithms have been presented to accomplish this optimization. Each algorithm performed similarly in terms of yield improvement. In terms of efficiency, the gradient based algorithm was the best. This algorithm is the worst in terms of scaling, though. For this reason, an EA based

algorithm such as moIWABB has a distinct advantage over gIWABB. Using a multiple objective EA makes it possible to obtain optimal product binning from a single optimization run.

This thesis has presented evidence demonstrating the effectiveness of the IWABB concept. By using IWABB and modern search algorithms, it is possible to mitigate the effects of process variations in semiconductor manufacturing. Reducing the effects of process variations improves manufacturing yields and increases profits. IWABB accomplishes this with a only a small increase in routing cost and silicon area.

6.3 Future Work

IWABB has been shown to be effective in simulation with a small circuit. Work is being conducted investigating its effectiveness on an actual microprocessor.

The testing done with fully floating wells (instead of voltage divider controlled wells) showed it was not an effective way of implementing IWABB. This was due to the suboptimal V_{b_p} in floating nwells. Grouping pfets into nwells based on their switching characteristics can control this floating well voltage. With wells grouped optimally, the voltage divider circuitry added to each well could be removed, and the silicon area necessary to implement IWABB reduced. Further research is needed to find methods to optimally form these groups.

Having multiple choices of floating well voltages may be a valuable addition to IWABB. This could be easily accomplished with voltage ladder in place of the voltage divider. It would also require more control signals going to each well. Investigations need to be performed on whether the benefits from such a method outweigh the expense.

Dynamically adjusting circuit operation parameters is becoming a very useful tool for reducing overall power consumption over time without impacting performance during peak demands. Given on- or off-chip control circuitry, the entire pareto front for a chip could be used as a trade-off curve for power and performance. The feasibility and value of such a mechanism needs further investigation.

REFERENCES

[1] Duane Boning and Sani Nassif. Models of process variations in device and interconnect. In Ananthea Chandrakasan, William J. Bowhill, and Frank Fox, editors, *Design of High-Performance Microprocessor Circuits*, chapter 6, pages 98–115. IEEE Press, 2001.

[2] *International Technology Roadmap for Semiconductors*. ITRS, 2002.

[3] Keith A. Bowman, Steven G. Duvall, and James D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2), February 2002.

[4] A. Srivastava, R. Bai, D. Blaauw, and D. Sylvester. Modeling and analysis of leakage power considering within-die process variations. *International Symposium on Low Power Electronic Desgin*, pages 64–67, 2002.

[5] W.C. Riordan, R. Miller, J.M. Sherman, and J. Hicks. Microprocessor reliability performance as a function of die location for a 0.25μ, five layer metal CMOS logic process. *1999 IEEE International Reliability Physics Symposium Proceedings*, pages 1–11, March 1999.

[6] Sani Nassif. Delay variability: sources, impacts and trends. *2000 IEEE International Solid-state circuits conference, Digest of technical papers*, pages 268–369, February 2000.

[7] K. Usami, K. Nogami, M. Igarashi, F. Minami, Y. Kawasaki, T. Ishikawa, M. Kanazawa, T. Aoki, M. Takano, C. Mizuno, M. Ichida, S. Sonoda, M. Takahashi, and N. Hatanaka. Automated low-power technique exploiting multiple supply voltages applied to a media processor. *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 131–134, May 1997.

[8] M. Takahashi, M. Hamada, T. Nishikawa, H. Arakida, Y. Tsuboi, T. Fujita, F. Hatori, S. Mita, K. Suzuki, A. Chiba, T. Terazawa, F. Sano, Y. Watanabe, H. Momose, K. Usami, M. Igarashi, M. Kanazawa, T. Kuroda, and T. Furuyama. A 60 mW MPEG4 video codec using clustered voltage scaling with variable supply-voltage scheme. *IEEE Journal of Solid-State Circuits*, 33(11):1772–1780, November 1998.

[9] Chunhong Chen, A. Srivastava, and M. Sarrafzadeh. On gate level power optimization using dual-supply voltages. *IEEE Transactions on Very Large Scale Integration Systems*, 9(5):616–629, October 2001.

[10] T. Kuroda and M. Hamada. Low-power CMOS digital design with dual embedded adaptive power supplies. *IEEE Journal of Solid-State Circuits*, 35(4):652–655, April 2000.

[11] Thomas Burd, Trevor Pering, Anthony Stratakos, and Robert Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, 35(11), November 2000.

[12] James Montanaro, Richard Witek, and Krishna Anne. A 160-mhz, 32-b 0.5w cmos risc processor. *IEEE Journal of Solid-State Circuits*, 31(11), November 1996.

[13] O.Y.-H. Leung, Chung-Wai Yue, Chi-Ying Tsui, and R.S. Cheng. Reducing power consumption of turbo code decoder using adaptive iteration with variable supply voltage. *Proceedings of the 1999 Internation Symposium on Low Power Electronics and Design*, pages 36–41, August 1999.

[14] H. Wann, Chenming Hu, K. Noda, D. Sinitsky, F. Assaderaghi, and J. Bokor. Channel doping engineering of mosfet with adaptable threshold voltage using body effect for low voltage and low power applications. *Proceedings of Technical Papers from 1995 International Symposium on VLSI Technology, Systems, and Applications*, pages 159–163, May 1995.

[15] T. Kuroda, T. Fujita, S. Mita, T. Nagamatu, S. Yoshioka, F. Sano, M. Norishima, M. Murota, M. Kako, M. Kinugawa, M. Kakumu, and T. Sakurai. A 0.9 V 150 MHz 10 mW 4 mm^2 2-D discrete cosine transform core processor with variable-threshold-voltage scheme. *Digest of Technical Papers from 1996 IEEE International Solid-State Circuits Conference*, pages 166–167, 437, February 1996.

[16] Hiroyuki Mizuno, Koichiro Ishibashi, and Takanori Shimura. An 18-μa standby current 1.8-v, 200-mhz microprocessor with self-substrate-biased data-retention mode. *IEEE Journal of Solid-State Circuits*, 34(11), November 1999.

[17] Masayuki Miyazaki, Goichi Ono, and Koichiro Ishibashi. A 1.2-gips/w microprocessor using speed-adaptive threshold-voltage cmos with forward bias. *IEEE Journal of Solid-State Circuits*, 37(2), February 2002.

[18] M. Miyazaki, H. Mizuno, and K. Ishibashi. A delay distribution squeezing scheme with speed-adaptive threshold-voltage CMOS (SA-Vt CMOS) for low voltage LSls. *1998 International Symposium on Low Power Electronics and Design Proceedings*, pages 48–53, August 1998.

[19] James W. Tschanz, James T. Kao, Siva G. Narenda, Raj Nair, Dimitri A. Antoniadis, Anantha P. Chandrakasan, and Vivek De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid-State Circuits*, 37(11), November 2002.

[20] T.A. Brunner. Impact of lens abberations on optical lithography. *IBM Journal of Research and Development*, 41(1/2), 1997.

[21] Michael Orshansky, Linda Milor, Pinhon Chen, Kurt Keutzer, and Chenming Hu. Impact of spatial intrachip gate length variability on the performance of high-speed digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(5):544–553, May 2002.

[22] D. J. Frank. Power-constrained cmos scaling limits. *International Business Machines Journal of Research and Development*, 46(2/3):235–244, March 2002.

[23] Voon-Yew Aaron Thean, Michael Sadd, and Jr. Bruce E. White. Effect of dopant granularity on superhalo-channel mosfet's according to two- and three-dimensional computer simulations. *IEEE Transactions on Nanotechnology*, 2(2):97–101, June 2003.

[24] T. Kuroda. Low power CMOS design challenges. *IEICE Transactions on Electronics*, E84-C(8):1021–1028, August 2001.

[25] David A. Johns and Ken Martin. *Analog Integrated Circuit Design*. John Wiley & Sons, Inc., 1997.

[26] K.K. Ng, S.A. Eshraghi, and T.D. Stanik. An improved generalized guide for MOSFET scaling. *IEEE Transactions on Electron Devices*, 40(10):1895–1897, October 1993.

[27] T. Sakurai and A.R. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *IEEE Journal of Solid-State Circuits*, 25(2):584–594, April 1990.

[28] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. *IEEE/ACM International Conference on Computer Aided Design*, pages 721–725, 2002.

[29] J.D. Jobson. *Applied Multivariate Data Analysis*. Springer-Verlag Publishing, 1992.

[30] Melanie Mitchell, John H. Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing? In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 51–58. Morgan Kaufmann Publishers, 1994.

[31] M. Yagiura and T. Ibaraki. Metaheuristics as robust and simple optimization. *Proceedings of the 1996 IEEE conference on Evolutionary Computation*, pages 541–546, 1996.

[32] Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[33] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[34] David H. Wolpert and William G. Macready. No free lunch theorm for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[35] D.B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(3):3–14, January 1994.

[36] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J. E. Rawlings, editor, *Foundations of genetic algorithms*, pages 69–93. Morgan Kaufmann, San Mateo, 1991.

[37] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms. TIK-Report 11, TIK Institut fur Technische Informatik und Kommunikationsnetze, Computer Engineering and Networks Laboratory, ETH, Swiss Federal Institute of Technology, Gloriastrasse 35, 8092 Zurich, Switzerland, 1995.

[38] D. Whitley and J. Kauth. Genitor: A different genetic algorithm. Technical Report CS-88-101, Colorado State University, 1988.

[39] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann Publishers Inc., 1989.

[40] Heinz Mtihlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm I: Continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[41] Thomas Bäck. Optimal mutation rates in genetic search. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 2–8, San Mateo, CA, USA, 1993. Morgan Kaufmann.

[42] Gabriela Ochoa, Inman Harvey, and Hilary Buxton. On recombination and optimal mutation rates. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 488–495, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[43] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strenght Pareto Evolutionary Algorithm. Technical Report 103, Swiss Federal Institue of Technology Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 2001.

[44] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strenght Pareto Evolutionary Algorithm for multiobjective optimization. In K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, and T. Fogarty, editors, *Evolutionary Methods for Design, Optimisation and Control with Applications to Industrail Problems. Proceedings of the EUROGEN2001 Conference, Athens, Greece, September 19-21, 2001*, pages 95–100. International Center for Numerical Methos in Engineering (CIMNE), Barcelona, Spain, 2002.

[45] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA–II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[46] Ghavam G. Shahidi and Fari Assaderaghi. Soi technology and circuts. In Ananthea Chandrakasan, William J. Bowhill, and Frank Fox, editors, *Design of High-Performance Microprocessor Circuits*, chapter 5. IEEE Press, 2001.

[47] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. *Digest of Technical Papers from 1994 IEEE Symposium on Low Power Electronics*, pages 8–11, October 1994.

[48] Lawrence Davis. A genetic algorithms tutorial. In *Handbook of Genetic Algorithms*, pages 1–101. Van Nostrand Reinhold, New York, 1991.

[49] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. PISA: A platform and programming language independent interface for search algorithms. In Carlos M. Fonseca, Peter J. Fleming, Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494–508, Berlin, 2003. Springer.

[50] Chang-Hoon Choi, Ki-Young Nam, Zhiping Yu, and Robert W. Dutton. Impact of gate direct tunneling current on circuit performance: A simulation study. *IEEE Transactions on Electron Devices*, 48(12):2823–2829, December 2001.

[51] Fernando G. Lobo, David E. Goldberg, and Martin Pelikan. Time complexity of genetic algorithms on exponentially scaled problems. Technical Report 2000015, University of Illinois at Urbana-Champaign, 2000.

[52] Hideki Asoh and Heinz Mühlenbein. On the mean convergence time of evolutionary algorithms without selection and mutation. Technical Report GMD-AS-TR-94-12, GMD, 1994.

Header files

```
#ifndef Chromosome_H
#define Chromosome_H



#include <vector>
#include <string>
using namespace std;

//#define  PWELL_BIAS_TEST −0.1
//#define  NWELL_BIAS_TEST 0.1
//#define  VDD_TEST −0.1
//#define  VDD_RES 0.05
#define MAX_NWELL_CHANGE 0.2
#define MAX_PWELL_CHANGE 0.2
#define MAX_VDD_CHANGE 0.1
#define MAX_PWELL .5
#define MIN_PWELL −.5
#define MAX_NWELL 1.6
#define MIN_NWELL 0.6
#define MAX_VDD 1.3
#define MIN_VDD 0.8
#define DEFAULT_VDD 1.1

// # define  PWELL_BIAS_TEST −0.1
// # define  NWELL_BIAS_TEST −0.1
// # define  VDD_RES 0.05
// # define  MAX_NWELL_CHANGE 0.2
// # define  MAX_PWELL_CHANGE 0.2
// # define  MAX_VDD_CHANGE 0.1
// # define  MAX_PWELL .5
// # define  MIN_PWELL −.5
// # define  MAX_NWELL 2.0
// # define  MIN_NWELL 1.0
// # define  MAX_VDD 1.7
// # define  MIN_VDD 1.2
// # define  DEFAULT_VDD 1.5

typedef enum
{
   Zero,
   One,
} ChromosomeValue;

//  typedef  enum
// {
//        TRIWELL,
//        SINGLEWELL_PLUS,
//        SINGLEWELL,
//        UNI,
//        CON,
//        FBOOST_TW,
//        FBOOST_SW,
//        FBOOST_UNI,
//        TEST,
// }  RunMode;
```

```
    typedef enum
    {
            CON,
60          NWF_DWB,
            NWF_NWB,
            NWF_PWB,
            DWB,
            VDD,
65          NWF_PWB_VDD,
            PWB_VDD,
            NWF_NWB_VDD,
            NWB_VDD,
            NWF,
70          NWF_VDD,

            DWF_DWB, //not used anymore
            FB_DWF_DWB, // not used anymore
            FB_NWF_VDD,
75          FB_NWF_DWB,
            FB_NWF_NWB,
            FB_NWF_PWB,
            FB_DWB,
            FB_VDD,
80          TEST, // Testing
    } RunMode;

    class Chromosome
    {
85    public:
        Chromosome(int nwell_count, int pwell_count, int chip_id);
        ~Chromosome();

        bool operator ==(const Chromosome& A);
90  //      ChromosomeValue& operator [](int index);
    //      ChromosomeValue operator [](int index) const ;
        int pwell_size () const;
        int nwell_size () const;

95      string NwellToString() const;
        string PwellToString() const;

        int Chip_Id;
        float Power;
100     float Frequency;
        float Fitness;
        float PwellBias;
        float NwellBias;
        float VDD;
105
        int NwellFloatCount();
        int PwellFloatCount();

        vector<ChromosomeValue> NwellFloats;
110     vector<ChromosomeValue> PwellFloats;



    };
115


    #endif
```

```
    #ifndef FitnessFunction_H
    #define FitnessFunction_H

    #include "Chromosome.h"
 5  #include <stdlib.h>
    #include <stdio.h>
    #include <pthread.h>
    #include <string>
    #include <fstream.h>
10  using namespace std;



    //yF
15  //#define TARGET_POWER 3.4e-9
    //#define TARGET_FREQ 10e7
    //y1
    //#define TARGET_POWER 3.3e-9
    //#define TARGET_FREQ 8.8e7
20  //y2
    //#define TARGET_POWER 3.35e-9
    //#define TARGET_FREQ 8.7e7
    //y3
    //#define TARGET_POWER 3.375e-9
25  //#define TARGET_FREQ 8.65e7
    //y4
    //#define TARGET_POWER 3.4e-9
    //#define TARGET_FREQ 8.60e7
    //y5
30  //#define TARGET_POWER 3.45e-9
    //#define TARGET_FREQ 8.55e7



35  class FitnessFunction
    {
    public:
        FitnessFunction(RunMode mode, char* filename, int nwell_size, int pwell_size, float Ft, float Pmax, int machine_num);
        ~FitnessFunction();
40
        void Evaluate(Chromosome& chromosome, Chromosome& best_chromosome);
```

```
        vector<Chromosome> Chromosome_List; // Cache list

45      float TARGET_FREQ;
        float TARGET_POWER;
        int spice_machine_num;
        int rotator;

50  private:
        bool WriteCircuit(const Chromosome& chromosome, char* filename);
        bool ParseEldo(char* filename, float& frequency, float& power, float vdd);
        float Cost(float frequency, float power);
        char* CreateFilename(const Chromosome& chromosome);
55
        int nwell_size;
        int pwell_size;

        ofstream file; // output file for this machines chromozomes
60
        RunMode run_mode;


    };
65


    #endif
```

```
    #ifndef GA_H
    #define GA_H


 5
    #include "Chromosome.h"
    #include "SelectionFunction.h"
    #include "ReproductionFunction.h"
    #include "FitnessFunction.h"
10  #include "Population.h"
    #include <stdlib.h>
    #include <time.h>


15
    void GA(Population& population, FitnessFunction& fitness_function,
            SelectionFunction& selection_function, ReproductionFunction& reproduction_function,
            int generations,time_t stoptime, int nwell_size, int pwell_size, int chip_id, Chromosome& best_chromosome);


20

    #endif
```

```
    #ifndef Population_H
    #define Population_H


 5
    #include "FitnessFunction.h"
    #include "Chromosome.h"
    #include <stdio.h>
    #include <stdlib.h>
10  #include <string>
    #include <time.h>
    #include <vector>
    #include <algorithm>

15  using namespace std;

    typedef enum
    {
            ADD,
20          REPLACE,
            REPLACE2,
    } AddMode;


25
    class Population
    {
            public:
                    Population(RunMode mode, float Ft, float Pmax, float bias_res);
30                  ~Population();

                    int RandomInitialize(int chip_id, int population_size, int nwell_size, int pwell_size,
                            FitnessFunction& fitness_function, int estimated_nwell_floats,
                    int estimated_pwell_floats, Chromosome& best_chromosome);
35                  int SmartInitialize(int chip_id, int population_size, int nwell_size, int pwell_size,
                            FitnessFunction& fitness_function, Chromosome& best_chromosome);
                            Population UpdateBias(int chip_id, int population_size, int nwell_size, int pwell_size,
                            FitnessFunction& fitness_function, Chromosome& best_chromosome);

40                  //Float slopes
                    float FvNwellFloat_slope;
                    float PvNwellFloat_slope;
                    float FvPwellFloat_slope;
                    float PvPwellFloat_slope;
45
                    //Bias slopes
                    float FvNwellBias_slope;
                    float PvNwellBias_slope;
                    float FvPwellBias_slope;
50                  float PvPwellBias_slope;
                    float FvVDDBias_slope;
                    float PvVDDBias_slope;
```

```
55              void Add(const Chromosome& chromosome, AddMode mode);

                Chromosome& operator [](int index);
                Chromosome operator [](int index) const;
                int size () const;
60              void clear();
                void push_back(const Chromosome& chromosome);
                int Find(const Chromosome& chromosome);

                float normal_frequency;
65              float normal_power;

                float TARGET_FREQ;
                float TARGET_POWER;

70              vector<Chromosome> vec_of_chromosomes;

                RunMode run_mode;

                float BIAS_RES;
75
        private:
                void UpdateNwellBias(Chromosome& chromosome, float estimated_power, int from_update);
                void UpdatePwellBias(Chromosome& chromosome, float estimated_power, int from_update);
                void UpdateVDDBias(Chromosome& chromosome, float estimated_power, int from_update);
80  };


    #endif
```

```
    #ifndef ReproductionFunction_H
    #define ReproductionFunction_H


 5
    #include "Population.h"
    #include "FitnessFunction.h"
    #include <math.h>
    #include <cmath>
10

    typedef enum
    {
        HUXR,
15  } ReproductionOperator;



    class ReproductionFunction
20  {
       public:
          ReproductionFunction(RunMode mode, ReproductionOperator reproduction_operator, int nwell_size, int pwell_size, float Ft, float Pmax
                , float bias_res);
          ~ReproductionFunction();

25        void Reproduce(int chip_id, const Population& parents, FitnessFunction& fitness_function,
                      Population& population, Chromosome& best_chromosome);

          ReproductionOperator Reproduction_Operator;
          int Nwell_Size;
30        int Pwell_Size;

          float TARGET_FREQ;
          float TARGET_POWER;

35        float BIAS_RES;

          RunMode run_mode;
    };

40

    #endif
```

```
    #ifndef SelectionFunction_H
    #define SelectionFunction_H


 5
    #include <stdlib.h>
    #include <time.h>
    #include "Population.h"

10

    typedef enum
    {
       TOURNAMENT,
15  } SelectionMode;



    class SelectionFunction
20  {
       public:
          SelectionFunction(SelectionMode selection_mode);
          ~SelectionFunction();

25        void Select(const Population& population, int parent_count, Population& parents);

          SelectionMode Selection_Mode;
```

```
};

#endif
```

Main code

```
/*
* Chromosome.cpp
*
*/


#include "Chromosome.h"
#include <iostream.h>


Chromosome::Chromosome(int nwell_size, int pwell_size, int chip_id)
{
        Chip_Id = chip_id;
        NwellBias = DEFAULT_VDD; //Chip VDD
        PwellBias = 0.00;
        VDD = DEFAULT_VDD;
        Fitness = -777.7e+7;
        Power = 777.7e+7;
        Frequency = -777.7e+7;

        // Initialize chromosome to fully connected
        NwellFloats.resize(nwell_size);
        PwellFloats.resize(pwell_size);
        for(int i= 0;i<nwell_size; i++) NwellFloats[i] = One;
        for(int i= 0;i<pwell_size; i++) PwellFloats[i] = One;

}


Chromosome::~Chromosome()
{
}


bool Chromosome::operator ==(const Chromosome& A)
{
        //check that they are the same size chromosomes
        if(NwellFloats.size() != A.NwellFloats.size() || PwellFloats.size() != A.PwellFloats.size())
        {
                cout << "Error: chromosome sizes does not match." << endl;
                cout << "Pwell sizes: " << PwellFloats.size() << " " << A.PwellFloats.size() << endl;
                cout << "Nwell sizes: " << NwellFloats.size() << " " << A.NwellFloats.size() << endl;
                exit(1);
        }
        //check to see that they are the same chip
        if(Chip_Id != A.Chip_Id) return false;
        //check biases
        if(PwellBias != A.PwellBias) return false;
        if(NwellBias != A.NwellBias) return false;
        if(VDD != A.VDD) return false;

        //check floats
//      for(int i = 0; i < nwell_size(); i++) if(NwellFloats[i] != A.NwellFloats[i]) return false;
//      for(int i = 0; i < pwell_size(); i++) if(PwellFloats[i] != A.PwellFloats[i]) return false;
        if (NwellFloats != A.NwellFloats) return false;
        if (PwellFloats != A.PwellFloats) return false;
        return true;
}


// ChromosomeValue& Chromosome::operator [](int index)
// {
//      return Data[index];
// }


// ChromosomeValue Chromosome::operator [](int index) const
// {
//      return Data[index];
// }


int Chromosome::nwell_size() const
{
   return NwellFloats.size();
}

int Chromosome::pwell_size() const
{
   return PwellFloats.size();
}

string Chromosome::NwellToString() const
{
   int i;
   string result(nwell_size(), 'n');

   for(i = 0; i < nwell_size(); i++)
   {
```

```
            if(NwellFloats[i] == One) result[i] = '1';
            else result[i] = '0';
100      }

         return result;
     }

105  string Chromosome::PwellToString() const
     {
         int i;
         string result(pwell_size(), 'p');

110      for(i = 0; i < pwell_size(); i++)
         {
            if(PwellFloats[i] == One) result[i] = '1';
            else result[i] = '0';
         }
115
         return result;
     }


120  int Chromosome::NwellFloatCount() {
             int count = 0;
             for(int i = 0; i< NwellFloats.size(); i++) {
                     if(NwellFloats[i] == Zero) count++;
             }
125          return count;
     }

     int Chromosome::PwellFloatCount() {
             int count = 0;
130          for(int i = 0; i< PwellFloats.size(); i++) {
                     if(PwellFloats[i] == Zero) count++;
             }
             return count;
     }
```

```
     #include "Chromosome.h"
     #include "SelectionFunction.h"
     #include "ReproductionFunction.h"
     #include "FitnessFunction.h"
  5  #include "Population.h"
     #include "GA.h"
     #include <stdio.h>
     #include <time.h>
     #include <fstream>
 10  //#include <fstream.h>


     #define NWELL_SIZE 32
     #define PWELL_SIZE 32
 15

     int main(int argc, char** argv)
     {
         if( argc < 7)
 20          {
             printf("Runs iABB on runinputs.\n");
             printf( "missing arguments\nuseage: command startchip finishchip run_mode bias_resolution yield# case# (delay) (power) (maxgen)\n
                 ");
             printf( "valid run_mode:\nCON\t\t0\nNWF_DWB\t\t1\nNWF_NWB\t\t2\nNWF_PWB\t\t3\n");
             printf( "DWB\t\t4\nVDD\t\t5\nNWF_PWB_VDD\t6\nPWB_VDD\t7\nNWF_NWB_VDD\t8\n");
 25          printf( "NWB_VDD\t\t9\nNWF\t\t10\nNWF_VDD\t\t11\n");
             printf( "Obselete:\nAll FB modes\nDWF_DWB\t\t30\nFB_DWF_DWB\t31\nTEST\t\t33\n\n");
             printf( "if yield=6, delay and power will be used as the targets\n");
             printf( "if maxgen is specified, all previous inputs must be present\n");
     // CON             0
 30  // NWF_DWB         1
     // NWF_NWB         2
     // NWF_PWB         3
     // DWB             4
     // VDD             5
 35  // NWF_PWB_VDD     6
     // PWB_VDD         7
     // NWF_NWB_VDD     8
     // NWB_VDD         9
     // NWF             10
 40  // NWF_VDD         11
     // Obselete:
     // DWF_DWB         30
     // FB_DWF_DWB 31
     // TEST        33
 45

             return 1;
             }

 50          int generations = 400; //was 50
             if (argc == 10) generations = atoi(argv[9]);
         int max_run_time = int(60 * 60 * 24*5); //5 days
         int population_size = 30; //must be atleast 2 was 15

 55          // yield definitions
     //      const int    size_of_yields_list  = 24; // added 8 modes for CSU
             const int size_of_yields_list = 6;
             int number_of_yields = size_of_yields_list ;
     //      float FREQ_LIST[ size_of_yields_list   ] = {0.445, 0.45, 0.455, 0.458, 0.435};
 60  //      float POWER_LIST[ size_of_yields_list   ] = {114.5, 113.5, 111.6, 110.4, 114.5};
             float FREQ_LIST[size_of_yields_list] =   {0.98, 1.00, 1.01, 1.02, 1.03, 0};
             float POWER_LIST[size_of_yields_list ]= {1.00, 1.00, 1.00, 1.00, 1.00, 0};
             if (argc >=9){
                     FREQ_LIST[size_of_yields_list-1] = 1/atof(argv[7]);
 65                  POWER_LIST[size_of_yields_list-1] = atof(argv[8]);
             }
```

```
        float FB_FREQ = 1.0;
        float FB_POWER = 180.0;

70 //       float  FB_FREQ = 10e7;
   //       float  FB_POWER = 3.4e-9;

        float TARGET_FREQ;
        float TARGET_POWER;
75      char runmode_string[20];
        RunMode run_mode;

        switch (atoi(argv[3])) {
                case 0:
80                      run_mode = CON;
                        sprintf(runmode_string, "CON");
                        printf("Starting connected eval\n");
                        break;

85              case 1:
                        run_mode = NWF_DWB;
                        sprintf(runmode_string, "NWF_DWB");
                        printf("Starting NWF_DWB\n");
                        break;
90
                case 2:
                        run_mode = NWF_NWB;
                        sprintf(runmode_string, "NWF_NWB");
                        printf("Starting NWF_NWB\n");
95                      break;

                case 3:
                        run_mode = NWF_PWB;
                        sprintf(runmode_string, "NWF_PWB");
100                     printf("Starting NWF_PWB\n");
                        break;

                case 4:
                        run_mode = DWB;
105                     sprintf(runmode_string, "DWB");
                        printf("Starting DWB\n");
                        break;

                case 5:
110                     run_mode = VDD;
                        sprintf(runmode_string, "VDD");
                        printf("Starting VDD\n");
                        break;

115             case 6:
                        run_mode = NWF_PWB_VDD;
                        sprintf(runmode_string, "NWF_PWB_VDD");
                        printf("Starting NWF_PWB_VDD\n");
                        break;
120
                case 7:
                        run_mode = PWB_VDD;
                        sprintf(runmode_string, "PWB_VDD");
                        printf("Starting PWB_VDD\n");
125                     break;

                case 8:
                        run_mode = NWF_NWB_VDD;
                        sprintf(runmode_string, "NWF_NWB_VDD");
130                     printf("Starting NWF_NWB_VDD\n");
                        break;

                case 9:
                        run_mode = NWB_VDD;
135                     sprintf(runmode_string, "NWB_VDD");
                        printf("Starting NWB_VDD\n");
                        break;

                case 10:
140                     run_mode = NWF;
                        sprintf(runmode_string, "NWF");
                        printf("Starting NWF\n");
                        break;

145             case 11:
                        run_mode = NWF_VDD;
                        sprintf(runmode_string, "NWF_VDD");
                        printf("Starting NWF_VDD\n");
                        break;
150
                case 12:
                        run_mode = FB_NWF_VDD;
                        number_of_yields = 1;
                        FREQ_LIST[0] = FB_FREQ;
155                     POWER_LIST[0] = FB_POWER;
                        sprintf(runmode_string, "FB_NWF_VDD");
                        printf("Starting FB_NWF_VDD\n");
                        break;

160             case 30:
                        run_mode = DWF_DWB;
                        sprintf(runmode_string, "DWF_DWB");
                        printf("Starting DWF_DWB\n");
                        break;
165
                case 31:
                        run_mode = FB_DWF_DWB;
                        number_of_yields = 1;
                        FREQ_LIST[0] = FB_FREQ;
170                     POWER_LIST[0] = FB_POWER;
                        sprintf(runmode_string, "FB_DWF_DWB");
                        printf("Starting FB_DWF_DWB\n");
                        break;
```

```
175             case 33:
                        run_mode = TEST;
                        sprintf(runmode_string, "TEST");
                        printf("Running test mode.\n");
                        break;
180
                default:
                        printf("Input mode was: ");
                        printf(argv[3]);
                        printf(".\ninvalid run_mode\n");
185                     printf( "useage: command startchip finishchip run_mode bias_resolution starting_pop#\n");
                        printf( "valid run_mode:\nCON\t\t0\nNWF_DWB\t\t1\nNWF_NWB\t\t2\nNWF_PWB\t\t3\n");
                        printf( "DWB\t\t4\nVDD\t\t5\nFB_NWF_DWB\t6\nFB_NWF_NWB\t7\nFB_NWF_PWB\t8\n");
                        printf( "FB_DWB\t\t9\nFB_VDD\t\t10\nNWF_VDD\t\t11\nFB_NWF_VDD\t\t12\n");
                        printf( "Obselete:\nDWF_DWB\t\t30\nFB_DWF_DWB\t31\nTEST\t\t33\n\n");
190
                        return 1;
                        break;
        }

195     int chip_id;
        int init_status = 0;
        int GA1_evals = 0;
        int number_of_good_chips = 0;

200         int startchip = atoi(argv[1]);
        int finishchip = atoi(argv[2]);
        float bias_res = atof(argv[4]);
        int starting_pop = atoi(argv[5]) - 1;
        int eldo_machine_num = atoi(argv[6]);
205     //int eldo_machine_num = 1;

        srandom(time(NULL));

        time_t ttt = time(NULL);
210     tm* ltime = localtime(&ttt);




215 //**********************************************************************************************************************

          if (run_mode == DWF_DWB || run_mode == NWF_DWB || run_mode == NWF_NWB || run_mode == NWF_PWB
                  || run_mode == FB_NWF_DWB || run_mode == FB_NWF_NWB || run_mode == FB_NWF_PWB || run_mode == NWF
                  || run_mode == NWF_VDD || run_mode == FB_NWF_VDD || run_mode == NWF_PWB_VDD || run_mode ==
                      NWF_NWB_VDD) {

220 //              for(int F_iterator = starting_pop ; F_iterator < number_of_yields ; F_iterator ++) {
                        int F_iterator = starting_pop;
                        TARGET_FREQ = FREQ_LIST[F_iterator];
    //              for(int P_iterator =0; P_iterator < number_of_yields ; P_iterator ++){
                        int P_iterator = F_iterator;
225                     TARGET_POWER = POWER_LIST[P_iterator];

                        //reset good chip counter
                        number_of_good_chips = 0;

230                     //setup output files
                        char temp_filename[256];
                        char cache_filename[256];
                        // sprintf ( cache_filename , " cache.%s.y%d.txt ", runmode_string , F_iterator +1); //name of file to store all evaluations
                               for all chips
                        // sprintf ( temp_filename ," best.%s.y%d.txt ", runmode_string , F_iterator +1);
235                     // ofstream best_log ( temp_filename ); //name of file to store only the best evals for each chip
                        // sprintf ( temp_filename ," eval.%s.y%d.txt ", runmode_string , F_iterator +1);
                        // ofstream eval_nfo ( temp_filename ); //name of file to store the quick stat info formatted as such:
                        // eval_nfo << "Chip\tGAs_required\tEvals_GA1\tEvals_GA2\tNFET_biased?\tThisChipGood?\tGoodChipsSoFar" << endl;

240                     //setup other functions
                        SelectionFunction selection_function(TOURNAMENT);
                        ReproductionFunction reproduction_function(run_mode, HUXR, NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                               TARGET_POWER, bias_res);
                        // FitnessFunction fitness_function (run_mode, cache_filename , NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                               TARGET_POWER, eldo_machine_num);
                        Population population(run_mode, TARGET_FREQ, TARGET_POWER, bias_res);
245
                        for(chip_id = startchip; chip_id <= finishchip; chip_id++){

                                sprintf(cache_filename, "%i.cache.%s.y%d.txt", chip_id, runmode_string, F_iterator+1); //name of file to
                                       store all evaluations for all chips
                                sprintf(temp_filename,"%i.best.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
250                             ofstream best_log(temp_filename); //name of file to store only the best evals for each chip
                                sprintf(temp_filename,"%i.eval.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
                                ofstream eval_nfo(temp_filename); //name of file to store the quick stat info formatted as such:
                                eval_nfo << "Chip\tGAs_required\tEvals_GA1\tEvals_GA2\tNFET_biased?\tThisChipGood?\
                                       tGoodChipsSoFar" << endl;

255                             FitnessFunction fitness_function(run_mode, cache_filename, NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                                       TARGET_POWER, eldo_machine_num);
                                Chromosome best_chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);

                                switch (population.SmartInitialize(chip_id, population_size, NWELL_SIZE, PWELL_SIZE, fitness_function,
                                       best_chromosome))
                                        {
260                                     //if chip is done, or there was something weird in initialization , call it done and move on
                                        case 1:
                                                printf("Acceptable configuration for chip %i found during initialization.\n", chip_id);
                                                printf("%i - %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount
                                                       (), best_chromosome.PwellFloatCount(),
                                                        best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
265                                                     best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);
                                                eval_nfo << chip_id << "\t0\t"<< fitness_function.Chromosome_List.size() << "\t0\t0\t";
                                                init_status = 1; //show that GA shouldn't be run
                                                break;

270                                     //normal init
```

```
                                        case 0:
                                                init_status = 0; //show that the GA should be run
                                                break;
                                        // catch all
275                                     default:
                                                printf("That shouldn't have happened (main 1) on chip %i\n Chip skipped\n\n", chip_id);
                                                eval_nfo << chip_id << "\t0\t" << fitness_function.Chromosome_List.size() << "\t0\t";
                                                init_status = 1; //show that the GA shouldn't be run
                                                break;
280                                     }


                                //if things are normal, run the GA
                                if( init_status == 0)
285                                     {
                                        init_status = 1;

                                        printf("Starting first GA for chip %i of %i to %i\n", chip_id, startchip, finishchip);
                                        GA(population, fitness_function, selection_function, reproduction_function,
290                                     generations,time(NULL)+ max_run_time, NWELL_SIZE, PWELL_SIZE, chip_id, best_chromosome);

                                        printf("Best configuration found after first GA for chip\n");
                                        printf("%i - %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount(),
                                                best_chromosome.PwellFloatCount(),
                                                best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
295                                             best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);

                                        GA1_evals = fitness_function.Chromosome_List.size();

                                        if((best_chromosome.Fitness < 0.0)) //took out power check
300                             {

                                                printf("Adjusting NFet Bias for chip %i\n", chip_id);

                                                population = population.UpdateBias(chip_id, population_size, NWELL_SIZE, PWELL_SIZE,
                                                        fitness_function, best_chromosome);
305
//no second ga                                          GA(population, fitness_function, selection_function, reproduction_function,
//                                                      generations, time(NULL)+ max_run_time, NWELL_SIZE, PWELL_SIZE, chip_id,
        best_chromosome);

                                                printf("Second pass of GA done\n");
310                                             printf("Best configuration found after second GA for chip\n");
                                                printf("%i - %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount
                                                        (), best_chromosome.PwellFloatCount(),
                                                        best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
                                                        best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);

315                                             eval_nfo << chip_id << "\t2\t" << GA1_evals
                                                        << "\t" << fitness_function.Chromosome_List.size() - GA1_evals<< "\t1\t";
                                                }

                                        else
320                                             {
                                                eval_nfo << chip_id << "\t1\t" << GA1_evals << "\t0\t0\t";
                                                }
                                        }

325                             printf("Best configuration found for chip\n");
                                printf("%i - %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount(),
                                        best_chromosome.PwellFloatCount(),
                                        best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
                                        best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);

330                             best_log << " " << best_chromosome.Chip_Id << " " << best_chromosome.NwellFloatCount() << " "
                                        << best_chromosome.PwellFloatCount() << " " << best_chromosome.Frequency << " " <<
                                                best_chromosome.Power
                                        << " " << best_chromosome.Fitness << " " << best_chromosome.PwellBias << " " <<
                                                best_chromosome.NwellBias << " "
                                        << best_chromosome.VDD << " "
                                        << best_chromosome.NwellToString().c_str() << " " << best_chromosome.PwellToString().c_str()
                                                << endl;
335
                                if (best_chromosome.Fitness > 0.0) {
                                        number_of_good_chips++;
                                        eval_nfo << "1\t" << number_of_good_chips << endl;
                                }
340                             else eval_nfo << "0\t" << number_of_good_chips << endl;

                                // clear the fitness function cache to save memory
                                fitness_function.Chromosome_List.clear();

345                     }
//                      eval_nfo.close();
//                      best_log.close();
                //}
                //}
350     }
//*****************************************************************************************************************
        else if (run_mode == CON) {

                // setup output files
355             char cache_filename[256];
                sprintf(cache_filename, "cache.%s.txt",runmode_string); //name of file to store all evaluations for all chips

            FitnessFunction fitness_function(run_mode, cache_filename, NWELL_SIZE, PWELL_SIZE, TARGET_FREQ, TARGET_POWER,
                eldo_machine_num);

360             for(chip_id = startchip; chip_id <= finishchip; chip_id++){
                        Chromosome chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
                        Chromosome dummy_best(NWELL_SIZE, PWELL_SIZE, chip_id);

//                      for (int i=0; i < NWELL_SIZE; i++) chromosome.NwellFloats[i]=Zero; // test fully floated
365

                        fitness_function.Evaluate(chromosome, dummy_best);
```

```
                        // clear the fitness function cache to save memory
370                     fitness_function .Chromosome_List.clear();
                }
        }

//*****************************************************************************************************
375     else if (run_mode == DWB || run_mode == FB_DWB) {
//              for (int F_iterator = starting_pop ; F_iterator < number_of_yields ; F_iterator ++) {
                        int F_iterator = starting_pop;
                        TARGET_FREQ = FREQ_LIST[F_iterator];
                        //for (int P_iterator =0; P_iterator < number_of_yields ; P_iterator ++){
380                     int P_iterator = F_iterator;
                        TARGET_POWER = POWER_LIST[P_iterator];

                        // reset good chip counter
                        number_of_good_chips = 0;
385
                        // setup output files
                        char temp_filename[256];
                        char cache_filename[256];
                        // sprintf ( cache_filename , " cache.%s.y%d.txt ", runmode_string , F_iterator +1); //name of file to store all evaluations
                                for all chips
390                     // sprintf ( temp_filename ," best.%s.y%d.txt ", runmode_string , F_iterator +1);
                        // ofstream best_log ( temp_filename ); // name of file to store only the best evals for each chip
                        // sprintf ( temp_filename ," eval.%s.y%d.txt ", runmode_string , F_iterator +1);
                        // ofstream eval_nfo ( temp_filename ); // name of file to store the quick stat info formatted as such:
                        // eval_nfo << "Chip\tEvals\tThisChipGood?\tGoodChipsSoFar" << endl;
395
                        // setup other functions
                        // FitnessFunction fitness_function (run_mode, cache_filename , NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                                TARGET_POWER, eldo_machine_num);

                        for(chip_id = startchip; chip_id <= finishchip; chip_id++){
400
                                sprintf(cache_filename, "%i.cache.%s.y%d.txt", chip_id, runmode_string, F_iterator+1); //name of file to
                                        store all evaluations for all chips
                                sprintf(temp_filename,"%i.best.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
                                ofstream best_log(temp_filename); //name of file to store only the best evals for each chip
                                sprintf(temp_filename,"%i.eval.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
405                             ofstream eval_nfo(temp_filename); //name of file to store the quick stat info formatted as such:
                                eval_nfo << "Chip\tGAs_required\tEvals_GA1\tEvals_GA2\tNFET_biased?\tThisChipGood?\
                                        tGoodChipsSoFar" << endl;

                                //setup other functions
                                FitnessFunction fitness_function (run_mode, cache_filename, NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                                        TARGET_POWER, eldo_machine_num);
410                             Chromosome best_chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
                                Chromosome chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
                                Chromosome biased(NWELL_SIZE, PWELL_SIZE, chip_id);
                                float FvPwellBias_slope, PvPwellBias_slope, FvNwellBias_slope, PvNwellBias_slope;

415                             fitness_function .Evaluate(chromosome, best_chromosome);
int i = 0;
                                while (best_chromosome.Fitness < 0.0 && i < 100) {
                                        float old_power = chromosome.Power;
                                        float old_frequency = chromosome.Frequency;
420                                     float old_pwell_bias = chromosome.PwellBias;
                                        float old_nwell_bias = chromosome.NwellBias;
                                        //Check Power
                                        if (chromosome.Power > TARGET_POWER) {
                                                if (random() % 100 > 70){ //use nwell to reduce power more often
425                                                     if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias =
                                                                MAX_NWELL;
                                                        if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                                        //update pwell instead
                                                        chromosome.PwellBias -= bias_res;
                                                        if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL
                                                                ;
430                                                     if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
                                                }
                                                else {
                                                        if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL
                                                                ;
                                                        if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
435                                                     //update nwell instead
                                                        chromosome.NwellBias += bias_res;
                                                        if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias =
                                                                MAX_NWELL;
                                                        if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                                }
440                                     }
                                        //check frequency
                                        else if (chromosome.Frequency < TARGET_FREQ){
                                                if (random() % 100 > 30){ //use pwell to increase speed more often
                                                        if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias =
                                                                MAX_NWELL;
445                                                     if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                                        //update pwell bias
                                                        chromosome.PwellBias += bias_res;
                                                        if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL
                                                                ;
                                                        if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
450                                             }
                                                else {
                                                        if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL
                                                                ;
                                                        if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
                                                        chromosome.NwellBias -= bias_res;
455                                                     if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias =
                                                                MAX_NWELL;
                                                        if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                                }
                                        }

460                                     i++;

                                        // printf ("%i %i %f %f %f %f %f %f %f\n", chip_id , i , chromosome.Frequency , chromosome.Power,
```

```
                chromosome. PwellBias , chromosome. NwellBias , chromosome.VDD, MIN_PWELL, MAX_PWELL);
            //Check rounding
            //chromosome. NwellBias = ( int (chromosome. NwellBias / bias_res ))* bias_res ;
465         //chromosome. PwellBias = ( int (chromosome. PwellBias / bias_res ))* bias_res ;

            fitness_function .Evaluate(chromosome, best_chromosome);
            printf("%i %i %f %f %f %f %f %f %f\n", chip_id, i, chromosome.Frequency, chromosome.Power,
                chromosome.PwellBias, chromosome.NwellBias, chromosome.VDD, MIN_PWELL, MAX_PWELL)
                ;
        }
470     printf("Best configuration found for chip\n");
        printf("%i − %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount(),
            best_chromosome.PwellFloatCount(),
            best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
            best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);

475     best_log << " " << best_chromosome.Chip_Id << " " << best_chromosome.NwellFloatCount() << " "
            << best_chromosome.PwellFloatCount() << " " << best_chromosome.Frequency << " " <<
                best_chromosome.Power
            << " " << best_chromosome.Fitness << " " << best_chromosome.PwellBias << " " <<
                best_chromosome.NwellBias << " "
            << best_chromosome.VDD << " "
            << best_chromosome.NwellToString().c_str() << " " << best_chromosome.PwellToString().c_str()
                << endl;
480
        eval_nfo << chip_id << "\t" << fitness_function.Chromosome_List.size() << "\t";
        if (best_chromosome.Fitness > 0.0) {
            number_of_good_chips++;
            eval_nfo << "1\t" << number_of_good_chips << endl;
485     }
        else eval_nfo << "0\t" << number_of_good_chips << endl;

        //Clear the fitness function cache to save memory
        fitness_function .Chromosome_List.clear();
490   }
//              best_log . close ();
//              eval_nfo . close ();
            //}
            //}
495 }

//*********************************************************************************
    else if (run_mode == PWB_VDD) {
//          for ( int F_iterator = starting_pop ; F_iterator < number_of_yields ; F_iterator ++) {
500     int F_iterator = starting_pop;
        TARGET_FREQ = FREQ_LIST[F_iterator];
        //for ( int P_iterator =0; P_iterator < number_of_yields ; P_iterator ++){
        int P_iterator = F_iterator;
        TARGET_POWER = POWER_LIST[P_iterator];
505
        // reset good chip counter
        number_of_good_chips = 0;

        //setup output files
510     char temp_filename[256];
        char cache_filename[256];
        // sprintf ( cache_filename , "cache.%s.y%d.txt", runmode_string , F_iterator +1); //name of file to store all evaluations
            for all chips
        // sprintf ( temp_filename ,"best.%s.y%d.txt", runmode_string , F_iterator +1);
        // ofstream best_log ( temp_filename ); //name of file to store only the best evals for each chip
515     // sprintf ( temp_filename ,"eval.%s.y%d.txt", runmode_string , F_iterator +1);
        // ofstream eval_nfo ( temp_filename ); //name of file to store the quick stat info formatted as such:
        // eval_nfo << "Chip\tEvals\tThisChipGood?\tGoodChipsSoFar" << endl;

        //setup other functions
520     // FitnessFunction fitness_function (run_mode , cache_filename , NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
            TARGET_POWER, eldo_machine_num);

        for(chip_id = startchip; chip_id <= finishchip; chip_id++){

            sprintf(cache_filename, "%i.cache.%s.y%d.txt", chip_id, runmode_string, F_iterator+1); //name of file to
                store all evaluations for all chips
525         sprintf(temp_filename,"%i.best.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
            ofstream best_log(temp_filename); //name of file to store only the best evals for each chip
            sprintf(temp_filename,"%i.eval.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
            ofstream eval_nfo(temp_filename); //name of file to store the quick stat info formatted as such:
            eval_nfo << "Chip\tGAs_required\tEvals_GA1\tEvals_GA2\tNFET_biased?\tThisChipGood?\
                tGoodChipsSoFar" << endl;
530
            //setup other functions
            FitnessFunction fitness_function (run_mode, cache_filename, NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                TARGET_POWER, eldo_machine_num);
            Chromosome best_chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
            Chromosome chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
535         Chromosome biased(NWELL_SIZE, PWELL_SIZE, chip_id);

            fitness_function .Evaluate(chromosome, best_chromosome);
int i = 0;
            while (best_chromosome.Fitness < 0.0 && i < 100) {
540             //Check Power
                if (chromosome.Power > TARGET_POWER) {
                    if (random() % 100 > 50){
                        //update pwell
                        chromosome.PwellBias −= bias_res;
545                     if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL
                            ;
                        if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
                    }
                    else {
                        //update VDD
550                     chromosome.VDD −= bias_res;
                        if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                        if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
                    }
                }
555             //check frequency
                else if (chromosome.Frequency < TARGET_FREQ){
```

```
                                if (random() % 100 > 50){
                                        //update pwell
                                        chromosome.PwellBias += bias_res;
560                                     if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL
                                            ;
                                        if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
                                }
                                else {
                                        //update VDD
565                                     chromosome.VDD += bias_res;
                                        if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                                        if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
                                }
                        }
570
                        i++;

                        fitness_function .Evaluate(chromosome, best_chromosome);
                        printf("%i %i %f %f %f %f %f %f %f\n", chip_id, i, chromosome.Frequency, chromosome.Power,
                               chromosome.PwellBias, chromosome.NwellBias, chromosome.VDD, MIN_PWELL, MAX_PWELL)
                               ;
575             }
                printf("Best configuration found for chip\n");
                printf("%i - %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount(),
                       best_chromosome.PwellFloatCount(),
                        best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
                        best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);
580
                best_log << " " << best_chromosome.Chip_Id << " " << best_chromosome.NwellFloatCount() << " "
                        << best_chromosome.PwellFloatCount() << " " << best_chromosome.Frequency << " " <<
                               best_chromosome.Power
                        << " " << best_chromosome.Fitness << " " << best_chromosome.PwellBias << " " <<
                               best_chromosome.NwellBias << " "
                        << best_chromosome.VDD << " "
585                     << best_chromosome.NwellToString().c_str() << " " << best_chromosome.PwellToString().c_str()
                                << endl;

                eval_nfo << chip_id << "\t" << fitness_function.Chromosome_List.size() << "\t";
                if (best_chromosome.Fitness > 0.0) {
                        number_of_good_chips++;
590                     eval_nfo << "1\t" << number_of_good_chips << endl;
                }
                else eval_nfo << "0\t" << number_of_good_chips << endl;

                //Clear the fitness function cache to save memory
595             fitness_function .Chromosome_List.clear();
        }
//                      best_log . close ();
//                      eval_nfo . close ();
                //}
600             //}
        }


//****************************************************************************************
605     else if (run_mode == NWB_VDD) {
//              for (int F_iterator = starting_pop ; F_iterator < number_of_yields ; F_iterator ++) {
                int F_iterator = starting_pop;
                TARGET_FREQ = FREQ_LIST[F_iterator];
                //for (int P_iterator =0; P_iterator < number_of_yields ; P_iterator ++){
610             int P_iterator = F_iterator;
                TARGET_POWER = POWER_LIST[P_iterator];

                // reset good chip counter
                number_of_good_chips = 0;
615
                //setup output files
                char temp_filename[256];
                char cache_filename[256];
                // sprintf ( cache_filename , " cache.%s.y%d.txt ", runmode_string , F_iterator +1); //name of file to store all evaluations
                        for all chips
620             // sprintf ( temp_filename ," best.%s.y%d.txt ", runmode_string , F_iterator +1);
                // ofstream best_log ( temp_filename ); //name of file to store only the best evals for each chip
                // sprintf ( temp_filename ," eval.%s.y%d.txt ", runmode_string , F_iterator +1);
                // ofstream eval_nfo ( temp_filename ); //name of file to store the quick stat info formatted as such:
                // eval_nfo << "Chip\tEvals\tThisChipGood?\tGoodChipsSoFar" << endl;
625
                //setup other functions
                // FitnessFunction fitness_function (run_mode , cache_filename , NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                        TARGET_POWER, eldo_machine_num);

                for(chip_id = startchip; chip_id <= finishchip; chip_id++){
630
                        sprintf(cache_filename, "%i.cache.%s.y%d.txt", chip_id, runmode_string, F_iterator+1); //name of file to
                                store all evaluations for all chips
                        sprintf(temp_filename,"%i.best.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
                        ofstream best_log(temp_filename); //name of file to store only the best evals for each chip
                        sprintf(temp_filename,"%i.eval.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
635                     ofstream eval_nfo(temp_filename); //name of file to store the quick stat info formatted as such:
                        eval_nfo << "Chip\tGAs_required\tEvals_GA1\tEvals_GA2\tNFET_biased?\tThisChipGood?\
                               tGoodChipsSoFar" << endl;

                        //setup other functions
                        FitnessFunction fitness_function(run_mode, cache_filename, NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                               TARGET_POWER, eldo_machine_num);
640                     Chromosome best_chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
                        Chromosome chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
                        Chromosome biased(NWELL_SIZE, PWELL_SIZE, chip_id);

                        fitness_function .Evaluate(chromosome, best_chromosome);
645 int i = 0;
                        while (best_chromosome.Fitness < 0.0 && i < 100) {
                                //Check Power
                                if (chromosome.Power > TARGET_POWER) {
                                        if (random() % 100 > 50){
650                                             //update nwell
                                                chromosome.NwellBias += bias_res;
```

```
                                                if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias =
                                                        MAX_NWELL;
                                                if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                        }
655                                     else {
                                                //update VDD
                                                chromosome.VDD −= bias_res;
                                                if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                                                if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
660                                     }
                                }
                                //check frequency
                                else if (chromosome.Frequency < TARGET_FREQ){
                                        if (random() % 100 > 50){
665                                             //update nwell
                                                chromosome.NwellBias −= bias_res;
                                                if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias =
                                                        MAX_NWELL;
                                                if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                        }
670                                     else {
                                                //update VDD
                                                chromosome.VDD += bias_res;
                                                if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                                                if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
675                                     }
                                }

                                i++;

680                             fitness_function .Evaluate(chromosome, best_chromosome);
                                printf("%i %i %f %f %f %f %f %f %f\n", chip_id, i, chromosome.Frequency, chromosome.Power,
                                        chromosome.PwellBias, chromosome.NwellBias, chromosome.VDD, MIN_PWELL, MAX_PWELL)
                                        ;
                        }
                        printf("Best configuration found for chip\n");
                        printf("%i − %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount(),
                                best_chromosome.PwellFloatCount(),
685                             best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
                                best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);

                        best_log << " " << best_chromosome.Chip_Id << " " << best_chromosome.NwellFloatCount() << " "
                                << best_chromosome.PwellFloatCount() << " " << best_chromosome.Frequency << " " <<
                                        best_chromosome.Power
690                             << " " << best_chromosome.Fitness << " " << best_chromosome.PwellBias << " " <<
                                        best_chromosome.NwellBias << " "
                                << best_chromosome.VDD << " "
                                << best_chromosome.NwellToString().c_str() << " " << best_chromosome.PwellToString().c_str()
                                         << endl;

                        eval_nfo << chip_id << "\t" << fitness_function.Chromosome_List.size() << "\t";
695                     if (best_chromosome.Fitness > 0.0) {
                                number_of_good_chips++;
                                eval_nfo << "1\t" << number_of_good_chips << endl;
                        }
                        else eval_nfo << "0\t" << number_of_good_chips << endl;
700
                        //Clear the fitness function cache to save memory
                        fitness_function .Chromosome_List.clear();
                }
//                      best_log . close () ;
705 //                  eval_nfo . close () ;
                //}
                //}
        }

710
//*****************************************************************************************************************
        else if (run_mode == VDD || run_mode == FB_VDD){
//              for(int F_iterator = starting_pop ; F_iterator < number_of_yields ; F_iterator ++) {
                        int F_iterator = starting_pop;
715                     TARGET_FREQ = FREQ_LIST[F_iterator];
                        //for(int P_iterator =0; P_iterator < number_of_yields ; P_iterator ++){
                        int P_iterator = F_iterator;
                        TARGET_POWER = POWER_LIST[P_iterator];

720                     //reset good chip counter
                        number_of_good_chips = 0;

                        //setup output files
                        char temp_filename[256];
725                     char cache_filename[256];
                        // sprintf ( cache_filename , " cache.%s.y%d.txt ", runmode_string , F_iterator +1); //name of file to store all evaluations
                                for all chips
                        // sprintf ( temp_filename ," best.%s.y%d.txt ", runmode_string , F_iterator +1);
                        // ofstream best_log ( temp_filename ) ; //name of file to store only the best evals for each chip
                        // sprintf ( temp_filename ," eval.%s.y%d.txt ", runmode_string , F_iterator +1);
730                     // ofstream eval_nfo ( temp_filename ) ; //name of file to store the quick stat info formatted as such:
                        // eval_nfo << "Chip\tEvals\tThisChipGood?\tGoodChipsSoFar" << endl;

                        //setup other functions
                        // FitnessFunction fitness_function (run_mode, cache_filename , NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
                                TARGET_POWER, eldo_machine_num);
735
                        for(chip_id = startchip; chip_id <= finishchip; chip_id++){

                                sprintf(cache_filename, "%i.cache.%s.y%d.txt", chip_id, runmode_string, F_iterator+1); //name of file to
                                        store all evaluations for all chips
                                sprintf(temp_filename,"%i.best.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
740                             ofstream best_log(temp_filename); //name of file to store only the best evals for each chip
                                sprintf(temp_filename,"%i.eval.%s.y%d.txt", chip_id, runmode_string, F_iterator+1);
                                ofstream eval_nfo(temp_filename); //name of file to store the quick stat info formatted as such:
                                eval_nfo << "Chip\tGAs_required\tEvals_GA1\tEvals_GA2\tNFET_biased?\tThisChipGood?\
                                        tGoodChipsSoFar" << endl;

745                             //setup other functions
                                FitnessFunction fitness_function(run_mode, cache_filename, NWELL_SIZE, PWELL_SIZE, TARGET_FREQ,
```

```
                                        TARGET_POWER, eldo_machine_num);
                                Chromosome best_chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
                                Chromosome chromosome(NWELL_SIZE, PWELL_SIZE, chip_id);
                                Chromosome biased(NWELL_SIZE, PWELL_SIZE, chip_id);
750
                                float FvVDD_slope;
                                float PvVDD_slope;

                                fitness_function .Evaluate(chromosome, best_chromosome);
755
                                if(best_chromosome.Fitness < 0){
                                        biased.VDD += bias_res;
                                        fitness_function .Evaluate(biased, best_chromosome);
                                        FvVDD_slope = (biased.Frequency − chromosome.Frequency)/(biased.VDD − chromosome.VDD); //
                                                pos
760                                     PvVDD_slope = (biased.Power − chromosome.Power)/(biased.VDD − chromosome.VDD); //pos
                                }

                                int i = 0;
                                while (best_chromosome.Fitness < 0.0 && i < 100) {
765                                     i++;
                                        float old_power = chromosome.Power;
                                        float old_frequency = chromosome.Frequency;
                                        float old_VDD = chromosome.VDD;

770                                     if(chromosome.Power > TARGET_POWER){
                                                float temp_VDD_change = (TARGET_POWER − chromosome.Power)/PvVDD_slope;
                                                temp_VDD_change = (int(temp_VDD_change/bias_res))∗bias_res;
                                                if (temp_VDD_change > MAX_VDD_CHANGE) temp_VDD_change = MAX_VDD_CHANGE;
                                                if (temp_VDD_change < −1∗MAX_VDD_CHANGE) temp_VDD_change = −1∗
                                                        MAX_VDD_CHANGE;
775                                             chromosome.VDD += temp_VDD_change;
                                                if(chromosome.VDD > MAX_VDD) chromosome.VDD = MAX_VDD;
                                                if(chromosome.VDD < MIN_VDD) chromosome.VDD = MIN_VDD;
                                        }
                                        else if (chromosome.Frequency < TARGET_FREQ) {
780                                             float temp_VDD_change = (TARGET_FREQ − chromosome.Frequency)/FvVDD_slope;
                                                temp_VDD_change = (int(temp_VDD_change/bias_res))∗bias_res;
                                                if (temp_VDD_change > MAX_VDD_CHANGE) temp_VDD_change = MAX_VDD_CHANGE;
                                                if (temp_VDD_change < −1∗MAX_VDD_CHANGE) temp_VDD_change = −1∗
                                                        MAX_VDD_CHANGE;
                                                chromosome.VDD += temp_VDD_change;
785                                             if(chromosome.VDD > MAX_VDD) chromosome.VDD = MAX_VDD;
                                                if(chromosome.VDD < MIN_VDD) chromosome.VDD = MIN_VDD;
                                        }


790                                     fitness_function .Evaluate(chromosome, best_chromosome);

                                        //update slopes
                                        if(chromosome.VDD != old_VDD){
                                                FvVDD_slope = (FvVDD_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                        VDD − old_VDD))/2;
795                                             PvVDD_slope = (PvVDD_slope + (chromosome.Power − old_power)/(chromosome.VDD −
                                                        old_VDD))/2;
                                        }
                                }

                                printf("Best configuration found for chip\n");
800                             printf("%i − %i %i %.5e %.5e %.5e %f %f %f\n", chip_id, best_chromosome.NwellFloatCount(),
                                        best_chromosome.PwellFloatCount(),
                                        best_chromosome.Frequency, best_chromosome.Power, best_chromosome.Fitness,
                                        best_chromosome.PwellBias, best_chromosome.NwellBias, best_chromosome.VDD);

                                best_log << " " << best_chromosome.Chip_Id << " " << best_chromosome.NwellFloatCount() << " "
805                                     << best_chromosome.PwellFloatCount() << " " << best_chromosome.Frequency << " " <<
                                                best_chromosome.Power
                                        << " " << best_chromosome.Fitness << " " << best_chromosome.PwellBias << " " <<
                                                best_chromosome.NwellBias << " "
                                        << best_chromosome.VDD << " "
                                        << best_chromosome.NwellToString().c_str() << " " << best_chromosome.PwellToString().c_str()
                                                << endl;

810                             eval_nfo << chip_id << "\t" << fitness_function.Chromosome_List.size() << "\t";
                                if (best_chromosome.Fitness > 0.0) {
                                        number_of_good_chips++;
                                        eval_nfo << "1\t" << number_of_good_chips << endl;
                                }
815                             else eval_nfo << "0\t" << number_of_good_chips << endl;

                                //Clear the fitness function cache to save memory
                                // fitness_function .Chromosome_List. clear ();
                        }
820 //                      best_log . close ();
    //                      eval_nfo . close ();
        //      }
        }

825 //*******************************************************************************************************
        else {
                printf ("Main: invalid run mode\n");
                return 1;
        }
830
        return 0;
}
```

```
/*
 *   FitnessFunction .cpp
 *
 */
5


#include "FitnessFunction.h"
```

```
10 #include <iostream.h>
   #include <strstream.h>
   #include <fstream.h>
   #include <stdlib.h>
   #include <stdio.h>
15 #include <string>
   #include <math.h>
   #include <time.h>

   #undef system
20
   #define ITERATOR_MAX 0
   //#define MAX_MACHINE_LOAD 10.5

   char* case_name ="add";
25
   // no_vdiv averages
   // float avg_power = 2.6962e-4;
   // float avg_freq = 2.8726e8;

30 // vdiv averages
   // float avg_power = 2.6961e-4;
   // float avg_freq = 2.8729e8;

   // vdiv2 averages
35 // float avg_power = 4.2632e-4;
   // float avg_freq = 2.8783e8;

   // vdiv3 averages
   float avg_power = 4.4314e-4;
40 float avg_freq = 2.8808e8;

   // float avg_power = 1;
   // float avg_freq = 1;

45 char* create_file_command = "cat /net/etltrk6/mnt/a/iabb/adder32_V5/spice/runinputs.vdiv3_0x1r0/%i.cserun.runinput %s.connect /net/
       etltrk6/mnt/a/iabb/adder32_V5/spice/tail.runinput.vdiv3 > /net/etltrk6/mnt/a/iabb/adder32_V5/spice/case.%s%i/cserun.runinput";

   char* run_file_command = ". /net/etltrk6/mnt/a/iabb/source_path/shbp; ispice -j -d -c %s%i /net/etltrk6/mnt/a/iabb/adder32_V5";

   char* remove_local_command = "rm -f %s.connect";
50




55
   bool FitnessFunction::WriteCircuit(const Chromosome& chromosome, char* filename){
       char fname[4096];
       sprintf(fname,"%s%s",filename,".connect");

60     FILE* f = fopen(fname,"w");

       if(!f){
           printf("could not open file %s.",fname);
           exit(1);
65     }

       // define the node names (numbers) for all the control inputs
       int vdiv_node_name[]=
      {2790, 2823, 2835, 2838, 2841, 2844, 2847, 2850, 2853,
70      2760, 2763, 2766, 2769, 2772, 2775, 2778, 2781, 2784, 2787,
        2793, 2796, 2799, 2802, 2805, 2808, 2811, 2814, 2817, 2820,
       2826, 2829, 2832};

       int pullup_node_name[]=
75    {2789, 2822, 2834, 2837, 2840, 2843, 2846, 2849, 2852,
        2759, 2762, 2765, 2768, 2771, 2774, 2777, 2780, 2783, 2786,
        2792, 2795, 2798, 2801, 2804, 2807, 2810, 2813, 2816, 2819,
       2825, 2828, 2831};

80     int nwell_bias_node_name = 2854;
       int vdiv_vdd_node_name = 2855;
       int vdd_node_name = 1;
       int substrate_node_name= 2757;

85    // print the nwell voltage
       fprintf(f, "40 1\n");
       fprintf(f, "%i 0 %f 0 0\n", nwell_bias_node_name, chromosome.NwellBias);
       fprintf(f, "103\n");

90    // print VDD value in the runinput form
       fprintf(f, "40 1\n");
       fprintf(f, "%i 0 %f 0 0\n", vdd_node_name, chromosome.VDD);
       fprintf(f, "103\n");

95    // print the pullup_control signals in the runinput form
   //     for(int i = 0; i < chromosome.nwell_size(); i++)
       for(int i = chromosome.nwell_size()-1; i >= 0; i--)
       {
           fprintf(f, "40 1\n");
100        fprintf(f, "%i 0 %f 0 0\n", pullup_node_name[i], chromosome.NwellFloats[i] == One
               ? float(0.0) : chromosome.VDD);
           fprintf(f, "103\n");
       }
   //     fprintf(f, "40 1\n");
105 //     fprintf(f, "%i 0 0.0 0 0\n", pullup_node_name[0]);
   //     fprintf(f, "103\n");

      // print the vdiv_control signals in the runinput form
   //     fprintf(f, "40 1\n");
110 //     fprintf(f, "%i 0 %f 0 0\n", vdiv_node_name[0], chromosome.VDD);
   //     fprintf(f, "103\n");
       for(int i = chromosome.nwell_size()-1; i >= 0; i--)
       {
           fprintf(f, "40 1\n");
```

```
115             fprintf(f, "%i 0 %f 0 0\n", vdiv_node_name[i], chromosome.NwellFloats[i] == One
                  ? chromosome.VDD : float(0.0));
              fprintf(f, "103\n");
          }

120     // print the substrate voltage
          fprintf(f, "40  1\n");
          fprintf(f, "%i 0 %f 0 0\n", substrate_node_name, chromosome.PwellBias);
          fprintf(f, "103\n");

125     // print the vdiv vdd node voltage
          fprintf(f, "40  1\n");
          fprintf(f, "%i 0 %f 0 0\n", vdiv_vdd_node_name, chromosome.VDD);
          fprintf(f, "103\n");

130 /*
          fprintf (f, " WaveDef VCC V DC %f\n", chromosome.VDD);
          fprintf (f, " WaveDef VCC1 V DC %f\n", chromosome.VDD);
          fprintf (f, " Source  VCC1 vdd1\n");
          fprintf (f, " WaveDef  substrate   V DC %f\n", chromosome.PwellBias);
135       fprintf (f, " Source   substrate    substrate \n");
          if (run_mode == VDD || run_mode == NWF_VDD){
                  fprintf (f, " WaveDef NWELL_BIAS V DC %f\n", chromosome.VDD);
                  fprintf (f, " Source  NWELL_BIAS NWELL_BIAS\n");
              }
140           else {
                  fprintf (f, " WaveDef NWELL_BIAS V DC %f\n", chromosome.NwellBias);
                  fprintf (f, " Source  NWELL_BIAS NWELL_BIAS\n");
              }
          fprintf (f," Source  VCC   nwell0_vdiv_control  \n");
145       for (int  i = 0;  i < chromosome. nwell_size  ();  i++){
              fprintf (f," Source  %s  nwell% d_vdiv_control  \n", chromosome. NwellFloats [i] == One ? "VCC" : "VSS" ,  i+1);
          }
          fprintf (f," Source  VSS   nwell0_pullup_control  \n");
          for (int  i = 0;  i < chromosome. nwell_size  ();  i++){
150           fprintf (f," Source  %s  nwell% d_pullup_control  \n", chromosome. NwellFloats [i] == One ? "VSS" : "VCC" , i+1);
          }
*/
          fclose(f);

155       char command[2048];
          sprintf(command, create_file_command, chromosome.Chip_Id, filename, case_name, spice_machine_num);
          system(command);

          return 0;
160 }

    bool FitnessFunction::ParseEldo(char* filename, float& frequency, float& power, float vdd){
          char command[2048];
          char buffer[4096];
165       float timing, d1, d2, delay, machine_load;
          FILE *fpower, *fdelay1, *fdelay2, *p;

          //look  for  an  open  machine,  and  then  run  on  that  machine

170       //once  an  available  machine  is  found,  copy  the  runinput  file  and  run  it.

          // Run  ispice  and  parse  the  output
          sprintf( command, run_file_command, case_name, spice_machine_num);
          system(command);
175
          printf ("Measuring delays   ......\ n");

//          sprintf (command, ". /net/ etltrk6 /mnt/a/iabb/ source_path /shbp ;  cd  /net/ etltrk6 /mnt/a/iabb/adder32_V5/ spice / case .%s%i;  hpspice  -p raw
            alias  -s -nowindows -c \"set noformat\" -c \"print  integ (i(vdd) ,0.0 n,12.0 n) /(12.0 n)\" -c \"print  dlyrr (Cin, Cout)\" -c \"print  dlyff (Cin
            ,Cout)\"", case_name ,  spice_machine_num );
              sprintf(command, ". /net/etltrk6/mnt/a/iabb/source_path/shbp; cd /net/etltrk6/mnt/a/iabb/adder32_V5/spice/case.%s%i; hpspice -
                   p raw alias -s -nowindows -c \"include /net/etltrk6/mnt/a/iabb/adder32_V5/spice/measure\"", case_name,
                   spice_machine_num);
180           p = popen(command, "r");
              if(!p){
                printf("pipe failed  for delay1\n");
                exit(0);
              }
185           fscanf(p, "%s", buffer);
              fscanf(p, "%s",buffer);
              fscanf(p, "%s",buffer);
              fscanf(p, "%s",buffer);
              fscanf(p, "%s",buffer);
190           power= vdd*fabs(atof(buffer));
              fscanf(p, "%s",buffer);
              power += vdd*fabs(atof(buffer));
              power = power / avg_power;
              fscanf(p, "%s",buffer);
195           d1 = atof(buffer);
              fscanf(p, "%s",buffer);
              d2 = atof(buffer);
              pclose(p);
//            frequency  = (1/((d1 + d2)/2))/4.4041 e8;
200           frequency = (1/((d1 + d2)/2))/avg_freq;

          // Remove  circuit   files
          sprintf(command, remove_local_command, filename);
          system( command );
205

          return 0;
    }

210

    // constructor
    FitnessFunction::FitnessFunction(RunMode mode, char* filename, int nwell_size, int pwell_size, float Ft, float  Pmax, int machine_num)
    : file (filename)
215 {
              run_mode = mode;
              TARGET_FREQ = Ft;
```

```
        TARGET_POWER = Pmax;
        spice_machine_num=machine_num; //this is the first machine number. Eg. =500 will run on 500, 501, 502
        rotator=ITERATOR_MAX; //start at the max, so the iterator will move it to 0 on the first run
}

// destructor
FitnessFunction::~FitnessFunction()
{

}


//Main function to evaluate chromosome.
void FitnessFunction::Evaluate(Chromosome& chromosome, Chromosome& best_chromosome)
{

    //**********************************************************
    //* Search the cache for the chromozome.                  *
    //**********************************************************
    for(int i = 0;  i < Chromosome_List.size(); i++){
        if(Chromosome_List[i] == chromosome){
            chromosome = Chromosome_List[i];
            chromosome.Fitness= Cost(chromosome.Frequency, chromosome.Power);
            return;
        }
    }

    //**********************************************************
    //* Fitness was not found in cache so run the the simulator *
    //**********************************************************
    // generate a unique filename for this evaluation
    char* filename = CreateFilename(chromosome);

    // write the .connect and .cir
    WriteCircuit(chromosome, filename);

    //Run SPICE and parse output for the .cir
    ParseEldo(filename, chromosome.Frequency, chromosome.Power, chromosome.VDD);

    // Evaluate fitness (cost)
    chromosome.Fitness= Cost(chromosome.Frequency, chromosome.Power);

    //Add Chromosome to cache
    Chromosome_List.push_back(chromosome);

    //Check to see if it is the best so far
    if(chromosome.Fitness > best_chromosome.Fitness) best_chromosome = chromosome;

    //Write to log file
    file << " " << chromosome.Chip_Id << " " << chromosome.NwellFloatCount() << " "
    << chromosome.PwellFloatCount() << " " << chromosome.Frequency << " " << chromosome.Power
    << " " << chromosome.Fitness << " " << chromosome.PwellBias << " " << chromosome.NwellBias << " "
    << chromosome.VDD << " "
    << chromosome.NwellToString().c_str() << " " << chromosome.PwellToString().c_str() << endl;

}

// this function creates a filename for the new .cir and .connect files
// it is first ten bits of the chromosome with a random number appended
// the entire chromosome is printed in the cache file and the best file
char* FitnessFunction::CreateFilename(const Chromosome& chromosome){
    char* filename = (char*)malloc(sizeof(char)*256);
    int number_of_bits_to_show = chromosome.nwell_size();
    if (number_of_bits_to_show > 10) number_of_bits_to_show = 10;
    char buffer[256];
    for(int i = 0; i < number_of_bits_to_show; i++)
        buffer[i] = chromosome.NwellFloats[i] == Zero ? '0' : '1' ;

    buffer[number_of_bits_to_show] = '\0';

    sprintf(filename,"%s_%ld",buffer,random());

    return filename;
}


float FitnessFunction::Cost(float frequency, float power){
    float fitness = 0;

    if(run_mode == FB_DWF_DWB || run_mode == FB_NWF_DWB || run_mode == FB_NWF_NWB || run_mode == FB_NWF_PWB){
                // if the chip meets the requirement, give it a bonus. If not give it a penalty
                if (frequency >= TARGET_FREQ && power >= TARGET_POWER) //region I: -power
                        fitness = -1 * pow((TARGET_POWER - power) / TARGET_POWER, 2);

                else if(frequency < TARGET_FREQ && power > TARGET_POWER) //if region II: -1*(10*power+freq)
                        fitness = -1 * (pow((TARGET_FREQ - frequency) / TARGET_FREQ, 2) + 10*pow((TARGET_POWER - power)
                                / TARGET_POWER, 2));

                else if(frequency < TARGET_FREQ && power < TARGET_POWER) //if region III: -freq
                        fitness = -1 * pow((TARGET_FREQ - frequency) / TARGET_FREQ, 2);

                else if(frequency > TARGET_FREQ && power < TARGET_POWER) //if region IV: power+freq (good)
                        fitness = pow((TARGET_FREQ - frequency) / TARGET_FREQ, 2) + pow((TARGET_POWER - power) /
                                TARGET_POWER, 2);

                return fitness;
    }
        else {
                // if the chip meets the requirement, give it a bonus. If not give it a penalty
                if (frequency >= TARGET_FREQ && power >= TARGET_POWER) //region I: -power
                        fitness = -1 * pow((TARGET_POWER - power) / TARGET_POWER, 2);

                else if(frequency < TARGET_FREQ && power > TARGET_POWER) //if region II: -1*(power+freq)
                        fitness = -1 * (pow((TARGET_FREQ - frequency) / TARGET_FREQ, 2) + pow((TARGET_POWER - power) /
                                TARGET_POWER, 2));

                else if(frequency < TARGET_FREQ && power < TARGET_POWER) //if region III: -freq
```

```
                        fitness = -1 * pow((TARGET_FREQ - frequency) / TARGET_FREQ, 2);

                else if(frequency > TARGET_FREQ && power < TARGET_POWER) //if region IV: power+freq (good)
325                     fitness = pow((TARGET_FREQ - frequency) / TARGET_FREQ, 2) + pow((TARGET_POWER - power) /
                                TARGET_POWER, 2);

                return fitness;
        }
}
```

```
#include "GA.h"


5   void GA(Population& population, FitnessFunction& fitness_function,
            SelectionFunction& selection_function, ReproductionFunction& reproduction_function,
            int generations, time_t stoptime, int nwell_size, int pwell_size, int chip_id, Chromosome& best_chromosome)
    {
        int i;
10      Population parents(DWF_DWB, 0, 0, 0); //mode and targets don't matter here, so just use whatever.

        for(i = 0; i < generations && time(NULL) < stoptime && best_chromosome.Fitness < 0.0; i++)
        {
//              printf("Population (%i)\n", i);
15  //              population.Print();
//              printf("\n");

            // Select parents and add them to the parents population
            selection_function.Select(population, 2, parents);
20          // Reproduce with the selected parents, evaluate their child,
            // and add it to the population with Population.Add(child,REPLACE2)
            reproduction_function.Reproduce(chip_id, parents, fitness_function, population, best_chromosome);

        }
25
//      printf("Population (%i)\n", generations);
//      population.Print();
//      printf("\n");
    }
```

```
#include "Population.h"


    Population::Population(RunMode mode, float Ft, float Pmax, float bias_res)
5   {
            //Float slopes
            float FvNwellFloat_slope = 1e-50; //make the F slope low so PvF slope is high (bad)
            float PvNwellFloat_slope = 1e50; // make the P slope high so PvF slope is high (bad)
            float FvPwellFloat_slope = 1e-50;
10          float PvPwellFloat_slope = 1e50;

            //Bias slopes
            float FvNwellBias_slope = -1e-50;
            float PvNwellBias_slope = -1e50;
15          float FvPwellBias_slope = 1e-50;
            float PvPwellBias_slope = 1e50;

            run_mode = mode;

20          TARGET_FREQ = Ft;
            TARGET_POWER = Pmax;

            BIAS_RES = bias_res;
    }
25


    Population::~Population()
    {
30  }


    int Population::SmartInitialize(int chip_id, int population_size, int nwell_size, int pwell_size, FitnessFunction& fitness_function,
            Chromosome& best_chromosome)
    {
35                  //return 0 to run GA normally
                    //return 1 to call the chip done
                    //return 2 to exit

            if(run_mode == DWF_DWB || run_mode == FB_DWF_DWB) {
40
                    int i;
                    int j;
                    int estimated_nwell_floats;
                    int estimated_pwell_floats;
45
                    Chromosome connected_test(nwell_size, pwell_size, chip_id);
                    Chromosome floated_test(nwell_size, pwell_size, chip_id);
                    Chromosome biased(nwell_size, pwell_size, chip_id);
                    Chromosome chromosome(nwell_size, pwell_size, chip_id);
50
                    //reset best_chromosome
                    best_chromosome = chromosome;

                    //Evaluate fully connected and fully floated to estimate the number of floats
55                  //needed to meet delay requirements by line between connected and floated chips.
                    //These shouldn't be included in the initial population

                    //Generate and evaluate fully connected chip
                    fitness_function.Evaluate(connected_test, best_chromosome);
60                  printf("Fully connected properties for chip:\n");
                    printf("%i - %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
                            connected_test.Fitness, connected_test.NwellBias, connected_test.PwellBias);
```

```
                normal_frequency = connected_test.Frequency;
65              normal_power = connected_test.Power;

                // if the connected chip is acceptable , return true
                if (connected_test.Fitness > 0) return 1;

70
                //***************************************************
                //Chip isn't good initially , so let's fix it up.
                //Generate and Evaluate fully floated chip and determine NwellFloat_slopes
                int temp_float_count = 0;
75              for(i = 0; i < nwell_size; i++) {
                        if( nwell_size/2 < random() % nwell_size){
                                floated_test.NwellFloats[i] = Zero;
                                temp_float_count++;
                        }
80              }
                fitness_function.Evaluate(floated_test, best_chromosome);
                if (best_chromosome.Fitness > 0) return 1;
                FvNwellFloat_slope = (floated_test.Frequency - normal_frequency)/temp_float_count; //pos
                PvNwellFloat_slope = (floated_test.Power - normal_power)/temp_float_count; //pos
85              // restore floated_test
                for(int i=0; i< nwell_size; i++) floated_test.NwellFloats[i] = One;

                // determine PwellFloat_slope
                for(i = 0; i < pwell_size; i++) {
90                      if( nwell_size/2 < random() % nwell_size){
                                floated_test.PwellFloats[i] = Zero;
                                temp_float_count++;
                        }
                }
95              fitness_function.Evaluate(floated_test, best_chromosome);
                if (best_chromosome.Fitness > 0) return 1;
                FvPwellFloat_slope = (floated_test.Frequency - normal_frequency)/temp_float_count; //pos
                PvPwellFloat_slope = (floated_test.Power - normal_power)/temp_float_count; //pos

100             //Generate and evaluate Pwell biased slopes
                biased.PwellBias += (BIAS_RES*-1);
                fitness_function.Evaluate(biased, best_chromosome);
                if (best_chromosome.Fitness > 0) return 1;
                FvPwellBias_slope = (biased.Frequency - normal_frequency)/(BIAS_RES*-1); //pos
105             PvPwellBias_slope = (biased.Power - normal_power)/(BIAS_RES*-1); //pos
                // restore biased chip
                biased.PwellBias = 0.0;

                //Determine Nwell biased slopes
110             biased.NwellBias += BIAS_RES;
                fitness_function.Evaluate(biased, best_chromosome);
                if (best_chromosome.Fitness > 0) return 1;
                FvNwellBias_slope = (biased.Frequency - normal_frequency)/BIAS_RES; //neg
                PvNwellBias_slope = (biased.Power - normal_power)/BIAS_RES; //neg
115

                estimated_nwell_floats = int((TARGET_FREQ - normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
                estimated_pwell_floats = int((TARGET_FREQ - normal_frequency) / FvPwellFloat_slope) + 1; // add one to make F a bit higher

120             // limit the estimated floats
                if ( estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                if ( estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;

                // call RandomInitialize
125             return RandomInitialize(chip_id, population_size, nwell_size , pwell_size , fitness_function ,
                        estimated_nwell_floats , estimated_pwell_floats , best_chromosome);

        }

130
//**************************************************************************************************************
        else if (run_mode == NWF_DWB || run_mode == FB_NWF_DWB) {
                int i;
                int j;
135             int estimated_nwell_floats;
                int estimated_pwell_floats;

                Chromosome connected_test(nwell_size, pwell_size, chip_id);
                Chromosome floated_test(nwell_size, pwell_size , chip_id);
140             Chromosome biased(nwell_size, pwell_size, chip_id);
                Chromosome chromosome(nwell_size, pwell_size, chip_id);

                // reset best_chromosome
                best_chromosome = chromosome;
145
                // Evaluate fully connected and fully floated to estimate the number of floats
                //needed to meet delay requirements by line between connected and floated chips.
                //These shouldn't be included in the initial population

150             //Generate and evaluate fully connected chip
                fitness_function.Evaluate(connected_test, best_chromosome);
                printf("Fully connected properties for chip:\n");
                printf("%i - %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
                        connected_test.Fitness, connected_test.NwellBias, connected_test.PwellBias);
155
                normal_frequency = connected_test.Frequency;
                normal_power = connected_test.Power;

                // if the connected chip is acceptable , return true
160             if (connected_test.Fitness > 0) return 1;


                //***************************************************
                //Chip isn't good initially , so let's fix it up.
165             //Generate and Evaluate fully floated chip and determine NwellFloat_slopes
                int temp_float_count = 0;
                for(i = 0; i < nwell_size; i++) {
                        if( nwell_size/2 < random() % nwell_size){
                                floated_test.NwellFloats[i] = Zero;
```

```
170                              temp_float_count++;
                         }
                  }
                  fitness_function .Evaluate( floated_test , best_chromosome);
                  if (best_chromosome.Fitness > 0) return 1;
175               FvNwellFloat_slope = (floated_test.Frequency − normal_frequency)/temp_float_count; //pos
                  PvNwellFloat_slope = (floated_test.Power − normal_power)/temp_float_count; //pos

                  //Generate and evaluate Pwell biased slopes
                  biased.PwellBias += (BIAS_RES*−1);
180               fitness_function .Evaluate(biased, best_chromosome);
                  if (best_chromosome.Fitness > 0) return 1;
                  FvPwellBias_slope = (biased.Frequency − normal_frequency)/(BIAS_RES*−1); //pos
                  PvPwellBias_slope = (biased.Power − normal_power)/(BIAS_RES*−1); //pos
                  // restore biased chip
185               biased.PwellBias = 0.0;

                  //Determine Nwell biased slopes
                  biased.NwellBias += BIAS_RES;
                  fitness_function .Evaluate(biased, best_chromosome);
190               if (best_chromosome.Fitness > 0) return 1;
                  FvNwellBias_slope = (biased.Frequency − normal_frequency)/BIAS_RES; //neg
                  PvNwellBias_slope = (biased.Power − normal_power)/BIAS_RES; //neg


195               // estimate the number of floats needed to make a population with about that many floats
                  // determine which is better to float based on PvFslope and some randomness
                  //so most of the time , do the more favorable , else do the opposite
                  estimated_nwell_floats = int((TARGET_FREQ − normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
                  estimated_pwell_floats = −1; //this should insure no pwells are ever floated
200
                  // limit the estimated floats
                  if ( estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                  if ( estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;

205               // call RandomInitialize
                  return RandomInitialize(chip_id, population_size, nwell_size , pwell_size , fitness_function ,
                         estimated_nwell_floats , estimated_pwell_floats , best_chromosome);

         }
210 //*****************************************************************************************************
         else if(run_mode == NWF_NWB || run_mode == FB_NWF_NWB){
                  int i;
                  int j;
                  int estimated_nwell_floats;
215               int estimated_pwell_floats;

                  Chromosome connected_test(nwell_size, pwell_size, chip_id);
                  Chromosome floated_test(nwell_size, pwell_size , chip_id);
                  Chromosome biased(nwell_size, pwell_size, chip_id);
220               Chromosome chromosome(nwell_size, pwell_size, chip_id);

                  // reset best_chromosome
                  best_chromosome = chromosome;

225               // Evaluate fully connected and fully floated to estimate the number of floats
                  //needed to meet delay requirements by line between connected and floated chips.
                  //These shouldn't be included in the initial population

                  //Generate and evaluate fully connected chip
230               fitness_function .Evaluate(connected_test, best_chromosome);
                  printf("Fully connected properties for chip:\n");
                  printf("%i − %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
                         connected_test.Fitness , connected_test.NwellBias, connected_test.PwellBias);

235               normal_frequency = connected_test.Frequency;
                  normal_power = connected_test.Power;

                  // if the connected chip is acceptable , return true
                  if (connected_test.Fitness > 0) return 1;
240

                  //*****************************************************
                  //Chip isn't good initially , so let's fix it up.
                  //Generate and Evaluate fully floated chip and determine NwellFloat_slopes
245               int temp_float_count = 0;
                  for(i = 0; i < nwell_size; i++) {
                         if( nwell_size/2 < random() % nwell_size){
                                floated_test .NwellFloats[i] = Zero;
                               temp_float_count++;
250                      }
                  }
                  fitness_function .Evaluate( floated_test , best_chromosome);
                  if (best_chromosome.Fitness > 0) return 1;
                  FvNwellFloat_slope = (floated_test.Frequency − normal_frequency)/temp_float_count; //pos
255               PvNwellFloat_slope = (floated_test.Power − normal_power)/temp_float_count; //pos

                  //Determine Nwell biased slopes
                  biased.NwellBias += BIAS_RES;
                  fitness_function .Evaluate(biased, best_chromosome);
260               if (best_chromosome.Fitness > 0) return 1;
                  FvNwellBias_slope = (biased.Frequency − normal_frequency)/BIAS_RES; //neg
                  PvNwellBias_slope = (biased.Power − normal_power)/BIAS_RES; //neg


265               // estimate the number of floats needed to make a population with about that many floats
                  // determine which is better to float based on PvFslope and some randomness
                  //so most of the time , do the more favorable , else do the opposite
                  estimated_nwell_floats = int((TARGET_FREQ − normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
                  estimated_pwell_floats = −1; //this should insure no pwells are ever floated
270
                  // limit the estimated floats
                  if ( estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                  if ( estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;

275               // call RandomInitialize
                  return RandomInitialize(chip_id, population_size, nwell_size , pwell_size , fitness_function ,
```

```
                    estimated_nwell_floats , estimated_pwell_floats , best_chromosome);

        }
280
//******************************************************************************************************
        else if(run_mode == NWF_PWB || run_mode == FB_NWF_PWB){
                int i;
                int j;
285             int estimated_nwell_floats;
                int estimated_pwell_floats;

                Chromosome connected_test(nwell_size, pwell_size, chip_id);
                Chromosome floated_test(nwell_size, pwell_size , chip_id);
290             Chromosome biased(nwell_size, pwell_size, chip_id);
                Chromosome chromosome(nwell_size, pwell_size, chip_id);

                // reset best_chromosome
                best_chromosome = chromosome;
295
                // Evaluate fully connected and fully floated to estimate the number of floats
                //needed to meet delay requirements by line between connected and floated chips.
                //These shouldn't be included in the initial population

300             //Generate and evaluate fully connected chip
                fitness_function.Evaluate(connected_test, best_chromosome);
                printf("Fully connected properties for chip:\n");
                printf("%i - %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
                        connected_test.Fitness, connected_test.NwellBias, connected_test.PwellBias);
305
                normal_frequency = connected_test.Frequency;
                normal_power = connected_test.Power;

                // if the connected chip is acceptable , return true
310             if (connected_test.Fitness > 0) return 1;


                //****************************************************
                //Chip isn't good initially , so let's fix it up.
315             //Generate and Evaluate fully floated chip and determine NwellFloat_slopes
                int temp_float_count = 0;
                for(i = 0; i < nwell_size; i++) {
                        if(nwell_size/2 < random() % nwell_size){
                                floated_test.NwellFloats[i] = Zero;
320                             temp_float_count++;
                        }
                }
                fitness_function.Evaluate(floated_test , best_chromosome);
                if (best_chromosome.Fitness > 0) return 1;
325             FvNwellFloat_slope = (floated_test.Frequency - normal_frequency)/temp_float_count; //pos
                PvNwellFloat_slope = (floated_test.Power - normal_power)/temp_float_count; //pos

                //Determine Nwell biased slopes
                biased.PwellBias += (BIAS_RES*-1);
330             fitness_function.Evaluate(biased, best_chromosome);
                if (best_chromosome.Fitness > 0) return 1;
                FvPwellBias_slope = (biased.Frequency - normal_frequency)/(BIAS_RES*-1); //neg
                PvPwellBias_slope = (biased.Power - normal_power)/(BIAS_RES*-1); //neg

335
                // estimate the number of floats needed to make a population with about that many floats
                // determine which is better to float based on PvFslope and some randomness
                //so most of the time, do the more favorable , else do the opposite
                estimated_nwell_floats = int((TARGET_FREQ - normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
340             estimated_pwell_floats = -1; //this should insure no pwells are ever floated

                // limit the estimated floats
                if ( estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                if ( estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;
345
                // call RandomInitialize
                return RandomInitialize(chip_id, population_size, nwell_size , pwell_size , fitness_function ,
                        estimated_nwell_floats , estimated_pwell_floats , best_chromosome);

350     }

//*****************************************************************************
        else if (run_mode == NWF_PWB_VDD) {
                int i;
355             int j;
                int estimated_nwell_floats;
                int estimated_pwell_floats;

                Chromosome connected_test(nwell_size, pwell_size, chip_id);
360             Chromosome floated_test(nwell_size, pwell_size , chip_id);
                Chromosome biased(nwell_size, pwell_size, chip_id);
                Chromosome chromosome(nwell_size, pwell_size, chip_id);

                // reset best_chromosome
365             best_chromosome = chromosome;

                // Evaluate fully connected and fully floated to estimate the number of floats
                //needed to meet delay requirements by line between connected and floated chips.
                //These shouldn't be included in the initial population
370
                //Generate and evaluate fully connected chip
                fitness_function.Evaluate(connected_test, best_chromosome);
                printf("Fully connected properties for chip:\n");
                printf("%i - %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
375                     connected_test.Fitness, connected_test.NwellBias, connected_test.PwellBias);

                normal_frequency = connected_test.Frequency;
                normal_power = connected_test.Power;

380             // if the connected chip is acceptable , return true
                if (connected_test.Fitness > 0) return 1;
```

```
                    //****************************************************
385                 //Chip isn't good initially , so let's fix it up.
                    //Generate and Evaluate fully floated chip and determine NwellFloat_slopes
                    int temp_float_count = 0;
                    for(i = 0; i < nwell_size; i++) {
                            if(nwell_size/2 < random() % nwell_size){
390                                 floated_test.NwellFloats[i] = Zero;
                                    temp_float_count++;
                            }
                    }
                    fitness_function.Evaluate(floated_test, best_chromosome);
395                 if (best_chromosome.Fitness > 0) return 1;
                    FvNwellFloat_slope = (floated_test.Frequency − normal_frequency)/temp_float_count; //pos
                    PvNwellFloat_slope = (floated_test.Power − normal_power)/temp_float_count; //pos

                    //Generate and evaluate Pwell biased slopes
400                 biased.PwellBias += (BIAS_RES*−1);
                    fitness_function.Evaluate(biased, best_chromosome);
                    if (best_chromosome.Fitness > 0) return 1;
                    FvPwellBias_slope = (biased.Frequency − normal_frequency)/(BIAS_RES*−1); //pos
                    PvPwellBias_slope = (biased.Power − normal_power)/(BIAS_RES*−1); //pos
405                 // restore biased chip
                    biased.PwellBias = 0.0;

                    //Determine VDD biased slopes
                    biased.VDD += BIAS_RES;
410                 fitness_function.Evaluate(biased, best_chromosome);
                    if (best_chromosome.Fitness > 0) return 1;
                    FvVDDBias_slope = (biased.Frequency − normal_frequency)/BIAS_RES; //pos
                    PvVDDBias_slope = (biased.Power − normal_power)/BIAS_RES; //pos

415
                    // estimate the number of floats needed to make a population with about that many floats
                    // determine which is better to float based on PvFslope and some randomness
                    //so most of the time, do the more favorable , else do the opposite
                    estimated_nwell_floats = int((TARGET_FREQ − normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
420                 estimated_pwell_floats = −1; //this should insure no pwells are ever floated

                    // limit the estimated floats
                    if ( estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                    if ( estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;
425
                    // call RandomInitialize
                    return RandomInitialize(chip_id, population_size, nwell_size, pwell_size, fitness_function,
                            estimated_nwell_floats, estimated_pwell_floats, best_chromosome);

430         }

    //****************************************************************************
            else if (run_mode == NWF_NWB_VDD) {
                    int i;
435                 int j;
                    int estimated_nwell_floats;
                    int estimated_pwell_floats;

                    Chromosome connected_test(nwell_size, pwell_size, chip_id);
440                 Chromosome floated_test(nwell_size, pwell_size, chip_id);
                    Chromosome biased(nwell_size, pwell_size, chip_id);
                    Chromosome chromosome(nwell_size, pwell_size, chip_id);

                    // reset best_chromosome
445                 best_chromosome = chromosome;

                    // Evaluate fully connected and fully floated to estimate the number of floats
                    //needed to meet delay requirements by line between connected and floated chips.
                    //These shouldn't be included in the initial population
450
                    // Generate and evaluate fully connected chip
                    fitness_function.Evaluate(connected_test, best_chromosome);
                    printf("Fully connected properties for chip:\n");
                    printf("%i − %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
455                         connected_test.Fitness, connected_test.NwellBias, connected_test.PwellBias);

                    normal_frequency = connected_test.Frequency;
                    normal_power = connected_test.Power;

460                 // if the connected chip is acceptable , return true
                    if (connected_test.Fitness > 0) return 1;


                    //****************************************************
465                 //Chip isn't good initially , so let's fix it up.
                    //Generate and Evaluate fully floated chip and determine NwellFloat_slopes
                    int temp_float_count = 0;
                    for(i = 0; i < nwell_size; i++) {
                            if(nwell_size/2 < random() % nwell_size){
470                                 floated_test.NwellFloats[i] = Zero;
                                    temp_float_count++;
                            }
                    }
                    fitness_function.Evaluate(floated_test, best_chromosome);
475                 if (best_chromosome.Fitness > 0) return 1;
                    FvNwellFloat_slope = (floated_test.Frequency − normal_frequency)/temp_float_count; //pos
                    PvNwellFloat_slope = (floated_test.Power − normal_power)/temp_float_count; //pos

                    //Generate and evaluate Nwell biased slopes
480                 biased.NwellBias += BIAS_RES;
                    fitness_function.Evaluate(biased, best_chromosome);
                    if (best_chromosome.Fitness > 0) return 1;
                    FvNwellBias_slope = (biased.Frequency − normal_frequency)/BIAS_RES; //neg
                    PvNwellBias_slope = (biased.Power − normal_power)/BIAS_RES; //neg
485                 // restore biased chip
                    biased.PwellBias = 0.0;

                    //Determine VDD biased slopes
                    biased.VDD += BIAS_RES;
490                 fitness_function.Evaluate(biased, best_chromosome);
```

```
                if (best_chromosome.Fitness > 0) return 1;
                FvVDDBias_slope = (biased.Frequency - normal_frequency)/BIAS_RES; //pos
                PvVDDBias_slope = (biased.Power - normal_power)/BIAS_RES; //pos
495
                // estimate the number of floats needed to make a population with about that many floats
                // determine which is better to float based on PvFslope and some randomness
                //so most of the time, do the more favorable , else do the opposite
                estimated_nwell_floats = int((TARGET_FREQ - normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
500             estimated_pwell_floats = -1; //this should insure no pwells are ever floated

                // limit the estimated floats
                if ( estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                if ( estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;
505
                // call RandomInitialize
                return RandomInitialize(chip_id, population_size, nwell_size , pwell_size , fitness_function ,
                        estimated_nwell_floats , estimated_pwell_floats , best_chromosome);

510     }

//****************************************************************************************************
        else if(run_mode == NWF){
                int i;
515             int j;
                int estimated_nwell_floats;
                int estimated_pwell_floats;

                Chromosome connected_test(nwell_size, pwell_size, chip_id);
520             Chromosome floated_test(nwell_size, pwell_size , chip_id);
                Chromosome biased(nwell_size, pwell_size, chip_id);
                Chromosome chromosome(nwell_size, pwell_size, chip_id);

                // reset best_chromosome
525             best_chromosome = chromosome;

                // Evaluate fully connected and fully floated to estimate the number of floats
                //needed to meet delay requirements by line between connected and floated chips.
                //These shouldn't be included in the initial population
530
                // Generate and evaluate fully connected chip
                fitness_function .Evaluate(connected_test, best_chromosome);
                printf("Fully connected properties for chip:\n");
                printf("%i - %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
535                     connected_test.Fitness, connected_test.NwellBias, connected_test.PwellBias);

                normal_frequency = connected_test.Frequency;
                normal_power = connected_test.Power;

540             //if the connected chip is acceptable , return true
                if (connected_test.Fitness > 0) return 1;


                //****************************************************
545             //Chip isn't good initially , so let's fix it up.
                // Generate and Evaluate fully floated chip and determine NwellFloat_slopes
                int temp_float_count = 0;
                for(i = 0; i < nwell_size; i++) {
                        if( nwell_size/2 < random() % nwell_size){
550                             floated_test .NwellFloats[i] = Zero;
                                temp_float_count++;
                        }
                }
                fitness_function .Evaluate(floated_test , best_chromosome);
555             if (best_chromosome.Fitness > 0) return 1;
                FvNwellFloat_slope = (floated_test.Frequency - normal_frequency)/temp_float_count; //pos
                PvNwellFloat_slope = (floated_test.Power - normal_power)/temp_float_count; //pos

                // estimate the number of floats needed to make a population with about that many floats
560             // determine which is better to float based on PvFslope and some randomness
                //so most of the time, do the more favorable , else do the opposite
                estimated_nwell_floats = int((TARGET_FREQ - normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
                estimated_pwell_floats = -1; //this should insure no pwells are ever floated

565             // limit the estimated floats
                if ( estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                if ( estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;

                // call RandomInitialize
570             return RandomInitialize(chip_id, population_size, nwell_size , pwell_size , fitness_function ,
                        estimated_nwell_floats , estimated_pwell_floats , best_chromosome);

        }

575 //****************************************************************************************************
        else if(run_mode == NWF_VDD || run_mode == FB_NWF_VDD){
                int i;
                int j;
                int estimated_nwell_floats;
580             int estimated_pwell_floats;

                Chromosome connected_test(nwell_size, pwell_size, chip_id);
                Chromosome floated_test(nwell_size, pwell_size , chip_id);
                Chromosome biased(nwell_size, pwell_size, chip_id);
585             Chromosome chromosome(nwell_size, pwell_size, chip_id);

                // reset best_chromosome
                best_chromosome = chromosome;

590             // Evaluate fully connected and fully floated to estimate the number of floats
                //needed to meet delay requirements by line between connected and floated chips.
                //These shouldn't be included in the initial population

                // Generate and evaluate fully connected chip
595             fitness_function .Evaluate(connected_test, best_chromosome);
                printf("Fully connected properties for chip:\n");
                printf("%i - %.5e %.5e %.5e %f %f\n", chip_id, connected_test.Frequency, connected_test.Power,
```

```
                    connected_test.Fitness, connected_test.NwellBias, connected_test.PwellBias);
600             normal_frequency = connected_test.Frequency;
                normal_power = connected_test.Power;

                //if the connected chip is acceptable , return true
                if (connected_test.Fitness > 0) return 1;
605

                //****************************************************
                //Chip isn't good initially , so let's fix it up.
                //Generate and Evaluate fully floated chip and determine NwellFloat_slopes
610             int temp_float_count = 0;
                for(i = 0; i < nwell_size; i++) {
                        if(nwell_size/2 < random() % nwell_size){
                                floated_test.NwellFloats[i] = Zero;
                                temp_float_count++;
615                     }
                }
                fitness_function.Evaluate(floated_test, best_chromosome);
                if (best_chromosome.Fitness > 0) return 1;
                FvNwellFloat_slope = (floated_test.Frequency - normal_frequency)/temp_float_count; //pos
620             PvNwellFloat_slope = (floated_test.Power - normal_power)/temp_float_count; //pos

                //Determine VDD biased slopes
                biased.VDD += BIAS_RES;
                fitness_function.Evaluate(biased, best_chromosome);
625             if (best_chromosome.Fitness > 0) return 1;
                FvVDDBias_slope = (biased.Frequency - normal_frequency)/BIAS_RES; //pos
                PvVDDBias_slope = (biased.Power - normal_power)/BIAS_RES; //pos


630             // estimate the number of floats needed to make a population with about that many floats
                // determine which is better to float based on PvFslope and some randomness
                //so most of the time, do the more favorable , else do the opposite
                estimated_nwell_floats = int((TARGET_FREQ - normal_frequency) / FvNwellFloat_slope) + 1; // add one to make F a bit higher
                estimated_pwell_floats = -1; //this should insure no pwells are ever floated
635
                // limit the estimated floats
                if (estimated_nwell_floats > nwell_size) estimated_nwell_floats = nwell_size;
                if (estimated_pwell_floats > pwell_size) estimated_pwell_floats = pwell_size;

640             // call RandomInitialize
                return RandomInitialize(chip_id, population_size, nwell_size, pwell_size, fitness_function,
                        estimated_nwell_floats, estimated_pwell_floats, best_chromosome);

        }
645

//*****************************************************************************************************
        else {
                printf("Bad run mode in Poplation:SmartInitialize()\n");
650             return 2;
        }
}


655

int Population::RandomInitialize(int chip_id, int population_size, int nwell_size, int pwell_size,
        FitnessFunction& fitness_function, int estimated_nwell_floats, int estimated_pwell_floats, Chromosome& best_chromosome)
{
660
    clear();
    float temp_nwell_change, temp_pwell_change, temp_vdd_change;

    if(run_mode == DWF_DWB || run_mode == NWF_DWB || run_mode == FB_DWF_DWB || run_mode == FB_NWF_DWB){
665             for(int j = 0; j < population_size; j++)
                {
                        Chromosome chromosome(nwell_size, pwell_size, chip_id);
                        if (best_chromosome.Fitness < 0.0)
                        {
670                             int nwell_float_count = 0;
                                int pwell_float_count = 0;
                                //determine if biasing is more effective than floating in increasing frequency
                                float best_Fslope = min(min(PvNwellFloat_slope/FvNwellFloat_slope, PvPwellFloat_slope/FvPwellFloat_slope
                                        ),
                                        min(PvNwellBias_slope/FvNwellBias_slope, PvPwellBias_slope/FvPwellBias_slope));
675                             if (best_Fslope == PvNwellFloat_slope/FvNwellFloat_slope) {
                                        //Nwell float is the best method so just let the floater take care of things
                                }
                                else if (best_Fslope == PvPwellFloat_slope/FvPwellFloat_slope) {
                                        //Pwell float is the best method so just let the floater take care of things
680                             }
                                else if (best_Fslope == PvNwellBias_slope/FvNwellBias_slope) {
                                        temp_nwell_change = .75*(TARGET_FREQ - normal_frequency) / FvNwellBias_slope;
                                        temp_nwell_change = (int (temp_nwell_change/BIAS_RES))*BIAS_RES;
                                        if (temp_nwell_change > MAX_NWELL_CHANGE) temp_nwell_change = MAX_NWELL_CHANGE;
685                                     if (temp_nwell_change < -1*MAX_NWELL_CHANGE) temp_nwell_change = -1*
                                                MAX_NWELL_CHANGE;

                                        if (temp_nwell_change != 0.0) {
                                                //Nwell bias is the best method so bias 3/4 of the way and let the floater do some
                                                estimated_nwell_floats = int(.25*estimated_nwell_floats);
690                                             estimated_pwell_floats = int(.25*estimated_pwell_floats);
                                                //change nwell bias based on F
                                                chromosome.NwellBias += temp_nwell_change;
                                                if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias = MAX_NWELL;
                                                if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
695                                     }
                                }
                                else if (best_Fslope == PvPwellBias_slope/FvPwellBias_slope) {
                                        temp_pwell_change = .75*(TARGET_FREQ - normal_frequency) / FvPwellBias_slope;
                                        temp_pwell_change = (int(temp_pwell_change/BIAS_RES))*BIAS_RES;
700                                     if (temp_pwell_change > MAX_PWELL_CHANGE) temp_pwell_change = MAX_PWELL_CHANGE;
                                        if (temp_pwell_change < -1*MAX_PWELL_CHANGE) temp_pwell_change = -1*
                                                MAX_PWELL_CHANGE;
```

```
                    if (temp_pwell_change != 0.0) {
                            //Pwell bias is the best method so bias 3/4 of the way and let the floater do some
705                         estimated_nwell_floats = int(.25*estimated_nwell_floats);
                            estimated_pwell_floats = int(.25*estimated_pwell_floats);
                            //change pwell bias based on F
                            chromosome.PwellBias += temp_pwell_change;
                            if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL;
710                         if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
                    }
            }
            else printf ("!****ERROR: bad compare in population, biasing might not be optimal. Not a huge deal.\n");

715         // build the chromosome and count the number of floats
            if (random() %100 > 25){
                    if (PvNwellFloat_slope/FvNwellFloat_slope < PvPwellFloat_slope/FvPwellFloat_slope ||
                        estimated_pwell_floats < 0){
                            for(int i = 0; i < nwell_size; i++)
                            {
720                                 if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] =
                                        One;
                                    else
                                    {
                                            chromosome.NwellFloats[i] = Zero;
725                                         nwell_float_count++;
                                    }
                            }
                    }
                    else {
                            for(int i = 0; i < pwell_size; i++)
730                         {
                                    if(estimated_pwell_floats <= random() % pwell_size) chromosome.PwellFloats[i] =
                                        One;
                                    else
                                    {
                                            chromosome.PwellFloats[i] = Zero;
735                                         pwell_float_count++;
                                    }
                            }
                    }
            }
740         else {
                    if (PvNwellFloat_slope/FvNwellFloat_slope > PvPwellFloat_slope/FvPwellFloat_slope ||
                        estimated_pwell_floats < 0){
                            for(int i = 0; i < nwell_size; i++)
                            {
                                    if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] =
                                        One;
745                                 else
                                    {
                                            chromosome.NwellFloats[i] = Zero;
                                            nwell_float_count++;
                                    }
750                         }
                    }
                    else {
                            for(int i = 0; i < pwell_size; i++)
                            {
755                                 if(estimated_pwell_floats <= random() % pwell_size) chromosome.PwellFloats[i] =
                                        One;
                                    else
                                    {
                                            chromosome.PwellFloats[i] = Zero;
                                            pwell_float_count++;
760                                 }
                            }
                    }
            }

765         //based on the number of floats , estimate the power increase and predict an pwell/nwell bias
            float estimated_power = normal_power + (nwell_float_count * PvNwellFloat_slope) + (pwell_float_count *
                PvPwellFloat_slope) +
                    ((chromosome.NwellBias − DEFAULT_VDD) * PvNwellBias_slope) + (chromosome.PwellBias *
                        PvPwellBias_slope);
            float estimated_freq = normal_frequency + (nwell_float_count * FvNwellFloat_slope) + (pwell_float_count *
                FvPwellFloat_slope) +
                    ((chromosome.NwellBias − DEFAULT_VDD) * FvNwellBias_slope) + (chromosome.PwellBias *
                        FvPwellBias_slope);
770 //take out the power check , just let the biasing always happen, try to hit the right power right off
//          if ( estimated_power > TARGET_POWER) {
                    if (random() % 100 > 25){
                            if(PvNwellBias_slope/FvNwellBias_slope > PvPwellBias_slope/FvPwellBias_slope) { //neg/neg
                                =pos
                                    //Change nwell Bias based on P
775                                 UpdateNwellBias(chromosome, estimated_power, 0);
                            }
                            else {
                                    //Change pwell bias based on P
                                    UpdatePwellBias(chromosome, estimated_power, 0);
780                         }
                    }
                    else {
                            if(PvNwellBias_slope/FvNwellBias_slope < PvPwellBias_slope/FvPwellBias_slope) {
                                    //Change nwell Bias based on P
785                                 UpdateNwellBias(chromosome, estimated_power, 0);
                            }
                            else {
                                    //Change pwell bias based on P
                                    UpdatePwellBias(chromosome, estimated_power, 0);
790                         }
                    }
//          }
//          else {
//                  if (random() %100 > 25){
795 //                          if ( PvNwellBias_slope / FvNwellBias_slope < PvPwellBias_slope / FvPwellBias_slope ) {
//                                  //change nwell bias based on F
//                                  temp_nwell_change = (TARGET_FREQ − estimated_freq) / FvNwellBias_slope ;
```

```
//                                              if ( temp_nwell_change > MAX_NWELL_CHANGE) temp_nwell_change =
        MAX_NWELL_CHANGE;
//                                              if ( temp_nwell_change < -1*MAX_NWELL_CHANGE) temp_nwell_change = -1*
        MAX_NWELL_CHANGE;
800     //                                      chromosome.NwellBias += temp_nwell_change ;
//                                              if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias = MAX_NWELL;
//                                              if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
//                                      }
//                                      else {
805     //                                      //change pwell bias based on F
//                                              temp_pwell_change = (TARGET_FREQ - estimated_freq) / FvPwellBias_slope ;
//                                              if ( temp_pwell_change > MAX_PWELL_CHANGE) temp_pwell_change =
        MAX_PWELL_CHANGE;
//                                              if ( temp_pwell_change < -1*MAX_PWELL_CHANGE) temp_pwell_change = -1*
        MAX_PWELL_CHANGE;
//                                              chromosome.PwellBias += temp_pwell_change ;
810     //                                      if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL;
//                                              if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
//                                      }
//                              }
//                              else {
815     //                              if ( PvNwellBias_slope / FvNwellBias_slope > PvPwellBias_slope / FvPwellBias_slope ) {
//                                              //change nwell bias based on F
//                                              temp_nwell_change = (TARGET_FREQ - estimated_freq) / FvNwellBias_slope ;
//                                              if ( temp_nwell_change > MAX_NWELL_CHANGE) temp_nwell_change =
        MAX_NWELL_CHANGE;
//                                              if ( temp_nwell_change < -1*MAX_NWELL_CHANGE) temp_nwell_change = -1*
        MAX_NWELL_CHANGE;
820     //                                      chromosome.NwellBias += temp_nwell_change ;
//                                              if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias = MAX_NWELL;
//                                              if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
//                                      }
//                                      else {
825     //                                      //change pwell bias based on F
//                                              temp_pwell_change = (TARGET_FREQ - estimated_freq) / FvPwellBias_slope ;
//                                              if ( temp_pwell_change > MAX_PWELL_CHANGE) temp_pwell_change =
        MAX_PWELL_CHANGE;
//                                              if ( temp_pwell_change < -1*MAX_PWELL_CHANGE) temp_pwell_change = -1*
        MAX_PWELL_CHANGE;
//                                              chromosome.PwellBias += temp_pwell_change ;
830     //                                      if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL;
//                                              if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
//                                      }
//                              }
//                      }
835
                                //now one could possibly check to see if this nfet bias will cause the power to drop enough that
                                //adding a float would be necessary .
                                //since biasing was done here , we can't update the slopes for P&F for Nwell nor NFet

840                     // evaluate the chromosome and add it . Then update the FvNwell_slope , PvNwell_slope ( if no nfet biasing )
                        fitness_function .Evaluate(chromosome, best_chromosome);
                        Add(chromosome, ADD);

                }
845     }

        return 0;
}
//*********************************************
850 else if(run_mode == NWF_PWB_VDD){
        for(int j = 0; j < population_size; j++)
        {
                Chromosome chromosome(nwell_size, pwell_size, chip_id);
                if (best_chromosome.Fitness < 0.0)
855             {
                        int nwell_float_count = 0;
                        int pwell_float_count = 0;
                        //determine if biasing is more effective than floating in increasing frequency
                        float best_Fslope = min(PvNwellFloat_slope/FvNwellFloat_slope,
860                             min(PvVDDBias_slope/FvVDDBias_slope, PvPwellBias_slope/FvPwellBias_slope));
                        if (best_Fslope == PvNwellFloat_slope/FvNwellFloat_slope) {
                                //Nwell float is the best method so just let the floater take care of things
                        }
                        else if (best_Fslope == PvVDDBias_slope/FvVDDBias_slope) {
865                             temp_vdd_change = .75*(TARGET_FREQ - normal_frequency) / FvVDDBias_slope;
                                temp_vdd_change = (int (temp_vdd_change/BIAS_RES))*BIAS_RES;
                                if (temp_vdd_change > MAX_VDD_CHANGE) temp_vdd_change = MAX_VDD_CHANGE;
                                if (temp_vdd_change < -1*MAX_VDD_CHANGE) temp_vdd_change = -1* MAX_VDD_CHANGE;

870                             if (temp_vdd_change != 0.0) {
                                        //Nwell bias is the best method so bias 3/4 of the way and let the floater do some
                                        estimated_nwell_floats = int(.25*estimated_nwell_floats);
                                        estimated_pwell_floats = int(.25*estimated_pwell_floats);
                                        //change nwell bias based on F
875                                     chromosome.VDD += temp_vdd_change;
                                        if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                                        if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
                                }
                        }
880                     else if (best_Fslope == PvPwellBias_slope/FvPwellBias_slope) {
                                temp_pwell_change = .75*(TARGET_FREQ - normal_frequency) / FvPwellBias_slope;
                                temp_pwell_change = (int(temp_pwell_change/BIAS_RES))*BIAS_RES;
                                if (temp_pwell_change > MAX_PWELL_CHANGE) temp_pwell_change = MAX_PWELL_CHANGE;
                                if (temp_pwell_change < -1*MAX_PWELL_CHANGE) temp_pwell_change = -1*
                                        MAX_PWELL_CHANGE;
885
                                if (temp_pwell_change != 0.0) {
                                        //Pwell bias is the best method so bias 3/4 of the way and let the floater do some
                                        estimated_nwell_floats = int(.25*estimated_nwell_floats);
                                        estimated_pwell_floats = int(.25*estimated_pwell_floats);
890                                     //change pwell bias based on F
                                        chromosome.PwellBias += temp_pwell_change;
                                        if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL;
                                        if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
                                }
895                     }
```

```
                                        else printf ("!****ERROR: bad compare in population, biasing might not be optimal. Not a huge deal.\n");

                                        // build the chromosome and count the number of floats
                                                for(int i = 0; i < nwell_size; i++)
900                                             {
                                                        if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] =
                                                                One;
                                                        else
                                                        {
                                                                chromosome.NwellFloats[i] = Zero;
905                                                             nwell_float_count++;
                                                        }
                                                }
                                        //based on the number of floats , estimate the power increase and predict an pwell/nwell bias
                                        float estimated_power = normal_power + (nwell_float_count * PvNwellFloat_slope) +
910                                             ((chromosome.NwellBias − DEFAULT_VDD) * PvNwellBias_slope) + (chromosome.PwellBias *
                                                        PvPwellBias_slope);
                                        float estimated_freq = normal_frequency + (nwell_float_count * FvNwellFloat_slope) +
                                                ((chromosome.NwellBias − DEFAULT_VDD) * FvNwellBias_slope) + (chromosome.PwellBias *
                                                        FvPwellBias_slope);
                                                if (random() % 100 > 50){
                                                        if(PvVDDBias_slope/FvVDDBias_slope > PvPwellBias_slope/FvPwellBias_slope) { //neg/neg=
                                                                pos
915                                                             //Change VDD Bias based on P
                                                                UpdateVDDBias(chromosome, estimated_power, 0);
                                                        }
                                                        else {
                                                                //Change pwell bias based on P
920                                                             UpdatePwellBias(chromosome, estimated_power, 0);
                                                        }
                                                }
                                                else {
                                                        if(PvVDDBias_slope/FvVDDBias_slope < PvPwellBias_slope/FvPwellBias_slope) {
925                                                             //Change VDD Bias based on P
                                                                UpdateVDDBias(chromosome, estimated_power, 0);
                                                        }
                                                        else {
                                                                //Change pwell bias based on P
930                                                             UpdatePwellBias(chromosome, estimated_power, 0);
                                                        }
                                                }

                                                //now one could possibly check to see if this nfet bias will cause the power to drop enough that
935                                             //adding a float would be necessary .
                                                // since biasing was done here , we can't update the slopes for P&F for Nwell nor NFet

                                        // evaluate the chromosome and add it . Then update the FvNwell_slope , PvNwell_slope ( if no nfet biasing )
                                        fitness_function .Evaluate(chromosome, best_chromosome);
940                                     Add(chromosome, ADD);

                                }
                        }

945                     return 0;
                }

                //********************************************
                else if(run_mode == NWF_NWB_VDD){
950                     for(int j = 0; j < population_size; j++)
                        {
                                Chromosome chromosome(nwell_size, pwell_size, chip_id);
                                if (best_chromosome.Fitness < 0.0)
                                {
955                                     int nwell_float_count = 0;
                                        int pwell_float_count = 0;
                                        //determine if biasing is more effective than floating in increasing frequency
                                        float best_Fslope = min(PvNwellFloat_slope/FvNwellFloat_slope,
                                                min(PvVDDBias_slope/FvVDDBias_slope, PvNwellBias_slope/FvNwellBias_slope));
960                                     if (best_Fslope == PvNwellFloat_slope/FvNwellFloat_slope) {
                                                //Nwell float is the best method so just let the floater take care of things
                                        }
                                        else if (best_Fslope == PvVDDBias_slope/FvVDDBias_slope) {
                                                temp_vdd_change = .75*(TARGET_FREQ − normal_frequency) / FvVDDBias_slope;
965                                             temp_vdd_change = (int (temp_vdd_change/BIAS_RES))*BIAS_RES;
                                                if (temp_vdd_change > MAX_VDD_CHANGE) temp_vdd_change = MAX_VDD_CHANGE;
                                                if (temp_vdd_change < −1*MAX_VDD_CHANGE) temp_vdd_change = −1* MAX_VDD_CHANGE;

                                                if (temp_vdd_change != 0.0) {
970                                                     //Nwell bias is the best method so bias 3/4 of the way and let the floater do some
                                                        estimated_nwell_floats = int(.25*estimated_nwell_floats);
                                                        estimated_pwell_floats = int(.25*estimated_pwell_floats);
                                                        //change nwell bias based on F
                                                        chromosome.VDD += temp_vdd_change;
975                                                     if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                                                        if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
                                                }
                                        }
                                        else if (best_Fslope == PvNwellBias_slope/FvNwellBias_slope) {
980                                             temp_nwell_change = .75*(TARGET_FREQ − normal_frequency) / FvNwellBias_slope;
                                                temp_nwell_change = (int(temp_nwell_change/BIAS_RES))*BIAS_RES;
                                                if (temp_nwell_change > MAX_NWELL_CHANGE) temp_nwell_change = MAX_NWELL_CHANGE;
                                                if (temp_nwell_change < −1*MAX_NWELL_CHANGE) temp_nwell_change = −1*
                                                        MAX_NWELL_CHANGE;

985                                             if (temp_nwell_change != 0.0) {
                                                        //Nwell bias is the best method so bias 3/4 of the way and let the floater do some
                                                        estimated_nwell_floats = int(.25*estimated_nwell_floats);
                                                        estimated_pwell_floats = int(.25*estimated_pwell_floats);
                                                        //change nwell bias based on F
990                                                     chromosome.NwellBias += temp_nwell_change;
                                                        if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias = MAX_NWELL;
                                                        if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                                }
                                        }
995                                     else printf ("!****ERROR: bad compare in population, biasing might not be optimal. Not a huge deal.\n");

                                        // build the chromosome and count the number of floats
```

```
                                        for(int i = 0; i < nwell_size; i++)
                                        {
1000                                            if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] =
                                                        One;
                                                else
                                                {
                                                        chromosome.NwellFloats[i] = Zero;
                                                        nwell_float_count++;
1005                                            }
                                        }
                                //based on the number of floats , estimate the power increase and predict an pwell/nwell bias
                                float estimated_power = normal_power + (nwell_float_count * PvNwellFloat_slope) +
                                        ((chromosome.NwellBias − DEFAULT_VDD) * PvNwellBias_slope) + (chromosome.PwellBias *
                                                PvPwellBias_slope);
1010                            float estimated_freq = normal_frequency + (nwell_float_count * FvNwellFloat_slope) +
                                        ((chromosome.NwellBias − DEFAULT_VDD) * FvNwellBias_slope) + (chromosome.PwellBias *
                                                FvPwellBias_slope);
                                        if (random() % 100 > 50){
                                                if(PvVDDBias_slope/FvVDDBias_slope > PvNwellBias_slope/FvNwellBias_slope) { //neg/neg=
                                                        pos
                                                        //Change VDD Bias based on P
1015                                                    UpdateVDDBias(chromosome, estimated_power, 0);
                                                }
                                                else {
                                                        //Change nwell bias based on P
                                                        UpdateNwellBias(chromosome, estimated_power, 0);
1020                                            }
                                        }
                                        else {
                                                if(PvVDDBias_slope/FvVDDBias_slope < PvNwellBias_slope/FvNwellBias_slope) {
                                                        //Change VDD Bias based on P
1025                                                    UpdateVDDBias(chromosome, estimated_power, 0);
                                                }
                                                else {
                                                        //Change nwell bias based on P
                                                        UpdateNwellBias(chromosome, estimated_power, 0);
1030                                            }
                                        }

                                        //now one could possibly check to see if this nfet bias will cause the power to drop enough that
                                        //adding a float would be necessary .
1035                                    // since biasing was done here , we can't update the slopes for P&F for Nwell nor NFet

                                // evaluate the chromosome and add it . Then update the FvNwell_slope , PvNwell_slope ( if no nfet biasing )
                                fitness_function .Evaluate(chromosome, best_chromosome);
                                Add(chromosome, ADD);
1040
                        }
                }

                return 0;
1045    }
        //*************************************************************************************************************
        else if(run_mode == NWF_NWB || run_mode == FB_NWF_NWB){
                for(int j = 0; j < population_size; j++)
                {
1050                    Chromosome chromosome(nwell_size, pwell_size, chip_id);
                        if (best_chromosome.Fitness < 0.0)
                        {
                                int nwell_float_count = 0;
                                int pwell_float_count = 0;
1055                            //determine if biasing is more effective than floating in increasing frequency
                                float best_Fslope = min(PvNwellFloat_slope/FvNwellFloat_slope, PvNwellBias_slope/FvNwellBias_slope);
                                if (best_Fslope == PvNwellFloat_slope/FvNwellFloat_slope) {
                                        //Nwell float is the best method so just let the floater take care of things
                                }
1060                            else if (best_Fslope == PvNwellBias_slope/FvNwellBias_slope) {
                                        temp_nwell_change = .75*(TARGET_FREQ − normal_frequency) / FvNwellBias_slope;
                                        temp_nwell_change = (int (temp_nwell_change/BIAS_RES))*BIAS_RES;
                                        if (temp_nwell_change > MAX_NWELL_CHANGE) temp_nwell_change = MAX_NWELL_CHANGE;
                                        if (temp_nwell_change < −1*MAX_NWELL_CHANGE) temp_nwell_change = −1*
                                                MAX_NWELL_CHANGE;
1065
                                        if (temp_nwell_change != 0.0) {
                                                //Nwell bias is the best method so bias 3/4 of the way and let the floater do some
                                                estimated_nwell_floats = int(.25*estimated_nwell_floats);
                                                estimated_pwell_floats = int(.25*estimated_pwell_floats);
1070                                            //change nwell bias based on F
                                                chromosome.NwellBias += temp_nwell_change;
                                                if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias = MAX_NWELL;
                                                if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                                        }
1075                            }
                                else printf ("!****ERROR: bad compare in population, biasing might not be optimal. Not a huge deal.\n");

                                // build the chromosome and count the number of floats
                                for(int i = 0; i < nwell_size; i++)
1080                            {
                                        if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] = One;
                                        else
                                        {
                                                chromosome.NwellFloats[i] = Zero;
1085                                            nwell_float_count++;
                                        }
                                }

                                //based on the number of floats , estimate the power increase and predict an pwell/nwell bias
1090                            float estimated_power = normal_power + (nwell_float_count * PvNwellFloat_slope) +
                                        ((chromosome.NwellBias − DEFAULT_VDD) * PvNwellBias_slope);
                                float estimated_freq = normal_frequency + (nwell_float_count * FvNwellFloat_slope) +
                                        ((chromosome.NwellBias − DEFAULT_VDD) * FvNwellBias_slope);
        //take out the power check , just let the biasing always happen , try to hit the right power right off
1095    //              if ( estimated_power > TARGET_POWER) {
                                        //Change nwell Bias based on P
                                        UpdateNwellBias(chromosome, estimated_power, 1);
                                //}
```

```
1100                            //now one could possibly check to see if this nfet bias will cause the power to drop enough that
                                //adding a float would be necessary.
                                //since biasing was done here, we can't update the slopes for P&F for Nwell nor NFet

                                // evaluate the chromosome and add it. Then update the FvNwell_slope , PvNwell_slope ( if no nfet biasing )
1105                            fitness_function.Evaluate(chromosome, best_chromosome);
                                Add(chromosome, ADD);

                        }
                }
1110
                return 0;
        }

        //*****************************************************************************************************
1115    else if(run_mode == NWF_PWB || run_mode == FB_NWF_PWB){
                for(int j = 0; j < population_size; j++)
                {
                        Chromosome chromosome(nwell_size, pwell_size, chip_id);
                        if (best_chromosome.Fitness < 0.0)
1120                    {
                                int nwell_float_count = 0;
                                int pwell_float_count = 0;
                                //determine if biasing is more effective than floating in increasing frequency
                                float best_Fslope = min(PvNwellFloat_slope/FvNwellFloat_slope, PvPwellBias_slope/FvPwellBias_slope);
1125                            if (best_Fslope == PvNwellFloat_slope/FvNwellFloat_slope) {
                                        //Nwell float is the best method so just let the floater take care of things
                                }
                                else if (best_Fslope == PvPwellBias_slope/FvPwellBias_slope) {
                                        temp_pwell_change = .75*(TARGET_FREQ - normal_frequency) / FvPwellBias_slope;
1130                                    temp_pwell_change = (int (temp_pwell_change/BIAS_RES))*BIAS_RES;
                                        if (temp_pwell_change > MAX_PWELL_CHANGE) temp_pwell_change = MAX_PWELL_CHANGE;
                                        if (temp_pwell_change < -1*MAX_PWELL_CHANGE) temp_pwell_change = -1*
                                                MAX_PWELL_CHANGE;

                                        if (temp_pwell_change != 0.0) {
1135                                            //Pwell bias is the best method so bias 3/4 of the way and let the floater do some
                                                estimated_nwell_floats = int(.25*estimated_nwell_floats);
                                                estimated_pwell_floats = int(.25*estimated_pwell_floats);
                                                //change nwell bias based on F
                                                chromosome.PwellBias += temp_nwell_change;
1140                                            if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL;
                                                if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
                                        }
                                }
                                else printf ("!****ERROR: bad compare in population, biasing might not be optimal. Not a huge deal.\n");
1145
                                // build the chromosome and count the number of floats
                                for(int i = 0; i < nwell_size; i++)
                                        {
                                                if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] = One;
1150                                            else
                                                {
                                                        chromosome.NwellFloats[i] = Zero;
                                                        nwell_float_count++;
                                                }
1155                                    }

                                //based on the number of floats, estimate the power increase and predict an pwell/nwell bias
                                float estimated_power = normal_power + (nwell_float_count * PvNwellFloat_slope) +
                                        ((chromosome.PwellBias - 0) * PvPwellBias_slope);
1160                            float estimated_freq = normal_frequency + (nwell_float_count * FvNwellFloat_slope) +
                                        ((chromosome.PwellBias - 0) * FvPwellBias_slope);
//take out the power check, just let the biasing always happen, try to hit the right power right off
//              if ( estimated_power > TARGET_POWER) {
                                        //Change pwell Bias based on P
1165                                    UpdatePwellBias(chromosome, estimated_power, 1);
                                //}

                                //now one could possibly check to see if this nfet bias will cause the power to drop enough that
                                //adding a float would be necessary.
1170                            //since biasing was done here, we can't update the slopes for P&F for Nwell nor NFet

                                // evaluate the chromosome and add it. Then update the FvNwell_slope , PvNwell_slope ( if no nfet biasing )
                                fitness_function.Evaluate(chromosome, best_chromosome);
                                Add(chromosome, ADD);
1175
                        }
                }

                return 0;
1180    }

        //*****************************************************************************************************
        else if(run_mode == NWF){
                for(int j = 0; j < population_size; j++)
1185            {
                        Chromosome chromosome(nwell_size, pwell_size, chip_id);
                        if (best_chromosome.Fitness < 0.0)
                        {
                                int nwell_float_count = 0;
1190
                                // build the chromosome and count the number of floats
                                for(int i = 0; i < nwell_size; i++)
                                        {
                                                if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] = One;
1195                                            else
                                                {
                                                        chromosome.NwellFloats[i] = Zero;
                                                        nwell_float_count++;
                                                }
1200                                    }

                                float estimated_power = normal_power + (nwell_float_count * PvNwellFloat_slope);
                                float estimated_freq = normal_frequency + (nwell_float_count * FvNwellFloat_slope);
                                //now one could possibly check to see if this nfet bias will cause the power to drop enough that
1205                            //adding a float would be necessary.
```

```
                                        // since biasing was done here, we can't update the slopes for P&F for Nwell nor NFet

                                        // evaluate the chromosome and add it. Then update the FvNwell_slope, PvNwell_slope (if no nfet biasing)
                                        fitness_function.Evaluate(chromosome, best_chromosome);
1210                                    Add(chromosome, ADD);

                                }
                        }

1215                    return 0;
                }
                //*************************************************************************************************
                else if(run_mode == NWF_VDD || run_mode == FB_NWF_VDD){
                        for(int j = 0; j < population_size; j++)
1220                    {
                                Chromosome chromosome(nwell_size, pwell_size, chip_id);
                                if (best_chromosome.Fitness < 0.0)
                                {
                                        int nwell_float_count = 0;
1225                                    int pwell_float_count = 0;
                                        //determine if biasing is more effective than floating in increasing frequency
                                        float best_Fslope = min(PvNwellFloat_slope/FvNwellFloat_slope, PvVDDBias_slope/FvVDDBias_slope);
                                        if (best_Fslope == PvNwellFloat_slope/FvNwellFloat_slope) {
                                                //Nwell float is the best method so just let the floater take care of things
1230                                    }
                                        else if (best_Fslope == PvVDDBias_slope/FvVDDBias_slope) {
                                                temp_vdd_change = .75*(TARGET_FREQ - normal_frequency) / FvVDDBias_slope;
                                                temp_vdd_change = (int (temp_vdd_change/BIAS_RES))*BIAS_RES;
                                                if (temp_vdd_change > BIAS_RES*2) temp_vdd_change = 2*BIAS_RES;
1235                                            if (temp_vdd_change < -1*BIAS_RES*2) temp_pwell_change = -1* BIAS_RES*2;

                                                if (temp_vdd_change != 0.0) {
                                                        //vdd bias is the best method so bias 3/4 of the way and let the floater do some
                                                        estimated_nwell_floats = int(.25*estimated_nwell_floats);
1240                                                    estimated_pwell_floats = int(.25*estimated_pwell_floats);
                                                        //change nwell bias based on F
                                                        chromosome.VDD += temp_vdd_change;
                                                        if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                                                        if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
1245                                            }
                                        }
                                        else printf ("!****ERROR: bad compare in population, biasing might not be optimal. Not a huge deal.\nThis
                                                may be a sign of a bigger problem.\n");

                                        // build the chromosome and count the number of floats
1250                                    for(int i = 0; i < nwell_size; i++)
                                                {
                                                        if(estimated_nwell_floats <= random() % nwell_size) chromosome.NwellFloats[i] = One;
                                                        else
                                                        {
1255                                                            chromosome.NwellFloats[i] = Zero;
                                                                nwell_float_count++;
                                                        }
                                                }

1260                                    //based on the number of floats, estimate the power increase and predict vdd bias
                                        float estimated_power = normal_power + (nwell_float_count * PvNwellFloat_slope) +
                                                ((chromosome.VDD - DEFAULT_VDD) * PvVDDBias_slope);
                                        float estimated_freq = normal_frequency + (nwell_float_count * FvNwellFloat_slope) +
                                                ((chromosome.PwellBias - DEFAULT_VDD) * FvVDDBias_slope);
1265            //take out the power check, just let the biasing always happen, try to hit the right power right off
                //              if (estimated_power > TARGET_POWER) {
                                                //Change pwell Bias based on P
                                                UpdateVDDBias(chromosome, estimated_power, 1);
                                        //}
1270
                                        //now one could possibly check to see if this nfet bias will cause the power to drop enough that
                                        //adding a float would be necessary.
                                        // since biasing was done here, we can't update the slopes for P&F for Nwell nor NFet

1275                                    // evaluate the chromosome and add it. Then update the FvNwell_slope, PvNwell_slope (if no nfet biasing)
                                        fitness_function.Evaluate(chromosome, best_chromosome);
                                        Add(chromosome, ADD);

                                }
1280                    }

                        return 0;
                }

1285    //*************************************************************************************************
                else {
                        printf("Bad run mode in Poplation:RandomInitialize()\n");
                        return 2;
                }
1290    }


        void Population::UpdateNwellBias(Chromosome& chromosome, float estimated_power, int from_update){
                if (from_update == 0 && (chromosome.NwellBias >= MAX_NWELL || chromosome.NwellBias <= MIN_NWELL)){
1295                    if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias = MAX_NWELL;
                        if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                        UpdatePwellBias(chromosome, estimated_power, 1);
                }
                else {
1300                    float temp_nwell_change = (TARGET_POWER - estimated_power) / PvNwellBias_slope;
                        temp_nwell_change = (int(temp_nwell_change/BIAS_RES))*BIAS_RES;
                        if (temp_nwell_change > MAX_NWELL_CHANGE) temp_nwell_change = MAX_NWELL_CHANGE;
                        if (temp_nwell_change < -1*MAX_NWELL_CHANGE) temp_nwell_change = -1* MAX_NWELL_CHANGE;
                        chromosome.NwellBias += temp_nwell_change;
1305                    if (chromosome.NwellBias >= MAX_NWELL) chromosome.NwellBias = MAX_NWELL;
                        if (chromosome.NwellBias <= MIN_NWELL) chromosome.NwellBias = MIN_NWELL;
                }
                return;
        }
1310
        void Population::UpdatePwellBias(Chromosome& chromosome, float estimated_power, int from_update){
```

```
          if (from_update == 0 && (chromosome.PwellBias >= MAX_PWELL || chromosome.PwellBias <= MIN_PWELL)){
                    if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL;
                    if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
1315                UpdateNwellBias(chromosome, estimated_power, 1);
          }
          else {
                    float temp_pwell_change = (TARGET_POWER − estimated_power) / PvPwellBias_slope;
                    temp_pwell_change = (int(temp_pwell_change/BIAS_RES))∗BIAS_RES;
1320                if (temp_pwell_change > MAX_PWELL_CHANGE) temp_pwell_change = MAX_PWELL_CHANGE;
                    if (temp_pwell_change < −1∗MAX_PWELL_CHANGE) temp_pwell_change = −1∗ MAX_PWELL_CHANGE;
                    chromosome.PwellBias += temp_pwell_change;
                    if (chromosome.PwellBias >= MAX_PWELL) chromosome.PwellBias = MAX_PWELL;
                    if (chromosome.PwellBias <= MIN_PWELL) chromosome.PwellBias = MIN_PWELL;
1325      }
          return;
}

void Population::UpdateVDDBias(Chromosome& chromosome, float estimated_power, int from_update){
1330      if (from_update == 0 && (chromosome.VDD >= MAX_VDD || chromosome.VDD <= MIN_VDD)){
                    if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                    if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
                    UpdateVDDBias(chromosome, estimated_power, 1);
          }
1335      else {
                    float temp_VDD_change = (TARGET_POWER − estimated_power) / PvVDDBias_slope;
                    temp_VDD_change = (int(temp_VDD_change/BIAS_RES))∗BIAS_RES;
                    if (temp_VDD_change > 2∗BIAS_RES) temp_VDD_change = 2∗BIAS_RES;
                    if (temp_VDD_change < −1∗2∗BIAS_RES) temp_VDD_change = −1∗ 2∗BIAS_RES;
1340                chromosome.VDD += temp_VDD_change;
                    if (chromosome.VDD >= MAX_VDD) chromosome.VDD = MAX_VDD;
                    if (chromosome.VDD <= MIN_VDD) chromosome.VDD = MIN_VDD;
          }
          return;
1345 }


Population Population::UpdateBias(int chip_id, int population_size, int nwell_size , int pwell_size , FitnessFunction& fitness_function,
       Chromosome& best_chromosome)
{
1350    int j;
        int i;

        Population newly_biased_pop(run_mode, TARGET_FREQ, TARGET_POWER, BIAS_RES);
        Chromosome chromosome(nwell_size, pwell_size, chip_id);
1355    // evaluate best chromosome in the population with an NFet Bias of −.005V
        //and use that as a model to predict the needed bias

        //make new population with a predicted nfet bias
        //based on the power of the old . stop if a good chip is found
1360    if(run_mode == DWF_DWB || run_mode == NWF_DWB || run_mode == FB_DWF_DWB || run_mode == FB_NWF_DWB){
                  for(j = 0; j < population_size; j++)
                  {
                            if (best_chromosome.Fitness < 0.0)
                            {
1365                                  chromosome = vec_of_chromosomes[j];
                                      float old_power = chromosome.Power;
                                      float old_frequency = chromosome.Frequency;
                                      float old_pwell_bias = chromosome.PwellBias;
                                      float old_nwell_bias = chromosome.NwellBias;
1370
                                      if (random() % 100 > 25){
                                                if(PvNwellBias_slope/FvNwellBias_slope > PvPwellBias_slope/FvPwellBias_slope) {
                                                          UpdateNwellBias(chromosome, chromosome.Power, 0);
                                                }
1375                                            else {
                                                          UpdatePwellBias(chromosome, chromosome.Power, 0);
                                                }
                                      }
                                      else {
1380                                            if(PvNwellBias_slope/FvNwellBias_slope < PvPwellBias_slope/FvPwellBias_slope) {
                                                          UpdateNwellBias(chromosome, chromosome.Power, 0);
                                                }
                                                else {
                                                          UpdatePwellBias(chromosome, chromosome.Power, 0);
1385                                            }
                                      }


                                      fitness_function .Evaluate(chromosome, best_chromosome);
1390                                  newly_biased_pop.Add(chromosome, ADD);
                                      // update nfet slopes
                                      if (chromosome.NwellBias != old_nwell_bias && chromosome.PwellBias == old_pwell_bias) {
                                                FvNwellBias_slope = (FvNwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                       NwellBias − old_nwell_bias)) / 2;
                                                PvNwellBias_slope = (PvNwellBias_slope + (chromosome.Power − old_power)/(chromosome.NwellBias
                                                       − old_nwell_bias)) / 2;
1395                                  }
                                      else if (chromosome.NwellBias == old_nwell_bias && chromosome.PwellBias != old_pwell_bias) {
                                                FvPwellBias_slope = (FvPwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                       PwellBias − old_pwell_bias)) / 2;
                                                PvPwellBias_slope = (PvPwellBias_slope + (chromosome.Power − old_power)/(chromosome.PwellBias
                                                       − old_pwell_bias)) / 2;
                                      }
1400                        }
                  }
          }

//∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗
1405
          else if(run_mode == NWF_NWB || run_mode == FB_NWF_NWB){
                  for(j = 0; j < population_size; j++)
                  {
                            if (best_chromosome.Fitness < 0.0)
1410                        {
                                      chromosome = vec_of_chromosomes[j];
                                      float old_power = chromosome.Power;
                                      float old_frequency = chromosome.Frequency;
```

```
                                float  old_nwell_bias = chromosome.NwellBias;
1415
                                UpdateNwellBias(chromosome, chromosome.Power, 1);

                                fitness_function .Evaluate(chromosome, best_chromosome);
                                newly_biased_pop.Add(chromosome, ADD);
1420                            // update  nfet  slopes
                                if (chromosome.NwellBias != old_nwell_bias) {
                                        FvNwellBias_slope = (FvNwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                NwellBias − old_nwell_bias)) / 2;
                                        PvNwellBias_slope = (PvNwellBias_slope + (chromosome.Power − old_power)/(chromosome.NwellBias
                                                 − old_nwell_bias)) / 2;
                                }
1425                    }
                }
        }

   //****************************************************************************************************
1430
        else  if(run_mode == NWF_PWB || run_mode == FB_NWF_PWB){
                for(j = 0; j < population_size; j++)
                {
                        if  (best_chromosome.Fitness < 0.0)
1435                    {
                                chromosome = vec_of_chromosomes[j];
                                float old_power = chromosome.Power;
                                float old_frequency = chromosome.Frequency;
                                float  old_pwell_bias = chromosome.PwellBias;
1440
                                UpdatePwellBias(chromosome, chromosome.Power, 1);

                                fitness_function .Evaluate(chromosome, best_chromosome);
                                newly_biased_pop.Add(chromosome, ADD);
1445                            // update  nfet  slopes
                                if (chromosome.PwellBias != old_pwell_bias) {
                                        FvPwellBias_slope = (FvPwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                PwellBias − old_pwell_bias)) / 2;
                                        PvPwellBias_slope = (PvPwellBias_slope + (chromosome.Power − old_power)/(chromosome.PwellBias
                                                 − old_pwell_bias)) / 2;
                                }
1450                    }
                }
        }

      //*******************************************************************
1455  if(run_mode == NWF_PWB_VDD){
                for(j = 0; j < population_size; j++)
                {
                        if  (best_chromosome.Fitness < 0.0)
                        {
1460                            chromosome = vec_of_chromosomes[j];
                                float old_power = chromosome.Power;
                                float old_frequency = chromosome.Frequency;
                                float  old_pwell_bias = chromosome.PwellBias;
                                float old_VDD = chromosome.VDD;
1465
                                if  (random() % 100 > 25){
                                        if(PvVDDBias_slope/FvVDDBias_slope > PvPwellBias_slope/FvPwellBias_slope) {
                                                UpdateVDDBias(chromosome, chromosome.Power, 0);
                                        }
1470                                    else {
                                                UpdatePwellBias(chromosome, chromosome.Power, 0);
                                        }
                                }
                                else {
1475                                    if(PvVDDBias_slope/FvVDDBias_slope < PvPwellBias_slope/FvPwellBias_slope) {
                                                UpdateVDDBias(chromosome, chromosome.Power, 0);
                                        }
                                        else {
                                                UpdatePwellBias(chromosome, chromosome.Power, 0);
1480                                    }
                                }


                                fitness_function .Evaluate(chromosome, best_chromosome);
1485                            newly_biased_pop.Add(chromosome, ADD);
                                // update  nfet  slopes
                                if  (chromosome.VDD != old_VDD && chromosome.PwellBias == old_pwell_bias) {
                                        FvVDDBias_slope = (FvNwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                VDD − old_VDD)) / 2;
                                        PvVDDBias_slope = (PvNwellBias_slope + (chromosome.Power − old_power)/(chromosome.VDD −
                                                old_VDD)) / 2;
1490                            }
                                else  if  (chromosome.VDD == old_VDD && chromosome.PwellBias != old_pwell_bias) {
                                        FvPwellBias_slope = (FvPwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                PwellBias − old_pwell_bias)) / 2;
                                        PvPwellBias_slope = (PvPwellBias_slope + (chromosome.Power − old_power)/(chromosome.PwellBias
                                                 − old_pwell_bias)) / 2;
                                }
1495                    }
                }
        }

      //*******************************************************************
1500  if(run_mode == NWF_NWB_VDD){
                for(j = 0; j < population_size; j++)
                {
                        if  (best_chromosome.Fitness < 0.0)
                        {
1505                            chromosome = vec_of_chromosomes[j];
                                float old_power = chromosome.Power;
                                float old_frequency = chromosome.Frequency;
                                float  old_nwell_bias = chromosome.NwellBias;
                                float old_VDD = chromosome.VDD;
1510
                                if  (random() % 100 > 25){
                                        if(PvVDDBias_slope/FvVDDBias_slope > PvNwellBias_slope/FvNwellBias_slope) {
```

```
                                                UpdateVDDBias(chromosome, chromosome.Power, 0);
                                        }
1515                                    else {
                                                UpdateNwellBias(chromosome, chromosome.Power, 0);
                                        }
                                }
                                else {
1520                                    if(PvVDDBias_slope/FvVDDBias_slope < PvNwellBias_slope/FvNwellBias_slope) {
                                                UpdateVDDBias(chromosome, chromosome.Power, 0);
                                        }
                                        else {
                                                UpdateNwellBias(chromosome, chromosome.Power, 0);
1525                                    }
                                }


                                fitness_function .Evaluate(chromosome, best_chromosome);
1530                            newly_biased_pop.Add(chromosome, ADD);
                                // update nfet slopes
                                if (chromosome.VDD != old_VDD && chromosome.NwellBias == old_nwell_bias) {
                                        FvVDDBias_slope = (FvNwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                VDD − old_VDD)) / 2;
                                        PvVDDBias_slope = (PvNwellBias_slope + (chromosome.Power − old_power)/(chromosome.VDD −
                                                old_VDD)) / 2;
1535                            }
                                else if (chromosome.VDD == old_VDD && chromosome.NwellBias != old_nwell_bias) {
                                        FvNwellBias_slope = (FvNwellBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                NwellBias − old_nwell_bias)) / 2;
                                        PvNwellBias_slope = (PvNwellBias_slope + (chromosome.Power − old_power)/(chromosome.NwellBias
                                                − old_nwell_bias)) / 2;
                                }
1540                    }
                }
        }

//**************************************************************************************************************
1545
        else if(run_mode == NWF){
                for(j = 0; j < population_size; j++)
                {
                        if (best_chromosome.Fitness < 0.0)
1550                    {
                                chromosome = vec_of_chromosomes[j];
                                float old_power = chromosome.Power;
                                float old_frequency = chromosome.Frequency;

1555                            newly_biased_pop.Add(chromosome, ADD);
                        }
                }
        }

1560 //**************************************************************************************************************

        else if(run_mode == NWF_VDD || run_mode == FB_NWF_VDD){
                for(j = 0; j < population_size; j++)
                {
1565                    if (best_chromosome.Fitness < 0.0)
                        {
                                chromosome = vec_of_chromosomes[j];
                                float old_power = chromosome.Power;
                                float old_frequency = chromosome.Frequency;
1570                            float old_VDD = chromosome.VDD;

                                UpdateVDDBias(chromosome, chromosome.Power, 1);

                                fitness_function .Evaluate(chromosome, best_chromosome);
1575                            newly_biased_pop.Add(chromosome, ADD);
                                // update nfet slopes
                                if (chromosome.VDD != old_VDD) {
                                        FvVDDBias_slope = (FvVDDBias_slope + (chromosome.Frequency − old_frequency)/(chromosome.
                                                VDD − old_VDD)) / 2;
                                        PvVDDBias_slope = (PvVDDBias_slope + (chromosome.Power − old_power)/(chromosome.VDD −
                                                old_VDD)) / 2;
1580                            }
                        }
                }
        }

1585 //**************************************************************************************************************
        else printf("Bad run mode in Population:UpdateBias()\n\n");

    vec_of_chromosomes.clear();
    return newly_biased_pop;
1590 }




1595
void Population::Add(const Chromosome& chromosome, AddMode mode)
{
    int i;
    float worst_fitness;
1600 int worst_index;

    if(mode == ADD)
    {
        push_back(chromosome);
1605 }
    else if(mode == REPLACE)
    {
        worst_fitness = vec_of_chromosomes[0].Fitness;
        worst_index = 0;
1610    for(i = 0; i < size(); i++)
        {
            if(vec_of_chromosomes[i].Fitness < worst_fitness)
            {
```

```
                worst_fitness = vec_of_chromosomes[i].Fitness;
1615            worst_index = i;
            }
        }
        vec_of_chromosomes[worst_index] = chromosome;
    }
1620 else if(mode == REPLACE2)
    {
        worst_fitness = vec_of_chromosomes[0].Fitness;
        worst_index = 0;
        for(i = 0; i < size(); i++)
1625    {
            if(vec_of_chromosomes[i] == chromosome) return;
            if(vec_of_chromosomes[i].Fitness < worst_fitness)
            {
                worst_fitness = vec_of_chromosomes[i].Fitness;
1630            worst_index = i;
            }
        }
        vec_of_chromosomes[worst_index] = chromosome;
    }
1635 else
    {
        printf("Population::Add - Invalid add mode\n");
    }
}
1640


Chromosome& Population::operator [](int index)
{
1645    return vec_of_chromosomes[index];
}



1650 Chromosome Population::operator [](int index) const
{
    return vec_of_chromosomes[index];
}

1655

int Population::size() const
{
    return vec_of_chromosomes.size();
1660 }



void Population::clear()
1665 {
    return vec_of_chromosomes.clear();
}


1670
void Population::push_back(const Chromosome& chromosome)
{
    vec_of_chromosomes.push_back(chromosome);
}
1675


int Population::Find(const Chromosome& chromosome)
{
1680    int i;

    for(i = 0; i < size(); i++)
    {
        if(vec_of_chromosomes[i] == chromosome) return i;
1685    }

    return -1;
}
```

```
#include "ReproductionFunction.h"

ReproductionFunction::ReproductionFunction(RunMode mode, ReproductionOperator reproduction_operator, int nwell_size, int pwell_size,
        float Ft, float Pmax, float bias_res)
{
5   Reproduction_Operator = reproduction_operator;
    Nwell_Size = nwell_size;
    Nwell_Size = pwell_size;
    run_mode = mode;

10          TARGET_FREQ = Ft;
            TARGET_POWER = Pmax;

            BIAS_RES = bias_res;

15 }



ReproductionFunction::~ReproductionFunction()
20 {
}



25 void ReproductionFunction::Reproduce(int chip_id, const Population& parents, FitnessFunction& fitness_function,
        Population& population, Chromosome& best_chromosome)
{
```

```
        Chromosome child(parents[0].nwell_size(), parents [0]. pwell_size () , chip_id);
30
        if(Reproduction_Operator == HUXR && run_mode == DWF_DWB)
        {
            // repeat  generation  for  each  pair  of  parents
            for(int parent_id = 0; parent_id < parents.size () ; parent_id+=2)
35          {
                //determine  which  parent  is  more  fit
                int stronger_parent = 0;
                if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                    stronger_parent = 1;
40
                // generate  the  child
                for(int j = 0; j < child. nwell_size () ; j++)
                {
                    if(parents[parent_id ]. NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
45                      child . NwellFloats[j] = parents[parent_id ]. NwellFloats[j ];
                    else //if  the  parents  are  different  ,  insert  bits  from random parents  favoring  the  faster  one
                    {
                        if(random() % 100 < 75){
                            child . NwellFloats[j] = parents[parent_id + stronger_parent]. NwellFloats[j ];
50                                          }
                        else {
                            child . NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent − 1))) ]. NwellFloats[j ];
                                            }
                    }
55              }
                            for(int j = 0; j < child. pwell_size () ; j++)
                {
                    if(parents[parent_id ]. PwellFloats[j] == parents[parent_id+1].PwellFloats[j])
                        child . PwellFloats[j] = parents[parent_id ]. PwellFloats[j ];
60                  else //if  the  parents  are  different  ,  insert  bits  from random parents  favoring  the  faster  one
                    {
                        if(random() % 100 < 75){
                            child . PwellFloats[j] = parents[parent_id + stronger_parent]. PwellFloats[j ];
                                            }
65                      else {
                            child . PwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent − 1))) ]. PwellFloats[j ];
                                            }
                    }
                }
70


                //have  the  NFet_Bias  be :
                //if  the avg power  of  the  parents  is  too  high ,  reduce  the  nfet  bias  by one  step
75              //if  the  avg  power  is  good ,  give  it  the  avg  nfet  bias
                float avg_parental_pwell_bias = (parents[parent_id ]. PwellBias + parents[parent_id+1].PwellBias) / 2;
                            float avg_parental_nwell_bias = (parents[parent_id ]. NwellBias + parents[parent_id+1].NwellBias) / 2;
                if ((parents[parent_id ]. Power + parents[parent_id+1].Power) / 2 > TARGET_POWER) {
                    if (random() % 100 >= 50 ){
80                                          child . PwellBias = avg_parental_pwell_bias − BIAS_RES;
                                            child . NwellBias = avg_parental_nwell_bias;
                                    }
                                    else {
                                            child . PwellBias = avg_parental_pwell_bias;
85                                          child . NwellBias = avg_parental_nwell_bias + BIAS_RES;
                                    }
                }
                else {
                    child . PwellBias = avg_parental_pwell_bias;
90                                  child . NwellBias = avg_parental_nwell_bias;
                            }

                //Mutate  the  PwellBias
                if(random() %100 < 10) child.PwellBias −= BIAS_RES;
95              if(random() %100 < 10) child.PwellBias += BIAS_RES;

                //Mutate  the  NwellBias
                if(random() %100 < 10) child.NwellBias −= BIAS_RES;
                if(random() %100 < 10) child.NwellBias += BIAS_RES;
100
                //Mutating  by  shuffling  bits
                if(random() % 10 == 0) // 10% mutation rate
                {
                    for (int j = 0; j < child. nwell_size () ; j++)
105                 {
                        int index1 = random()%child.nwell_size();
                        int index2 = random()%child.nwell_size();
                        int tmp = child.NwellFloats[index1] == One ? 1 : 0;
                        child . NwellFloats[index1] = child. NwellFloats[index2];
110                     child . NwellFloats[index2] = tmp == 1 ? One: Zero;
                    }
                }
                if(random() % 10 == 0) // 10% mutation rate
                {
115                 for (int j = 0; j < child. pwell_size () ; j++)
                    {
                        int index1 = random()%child.pwell_size();
                        int index2 = random()%child.pwell_size();
                        int tmp = child.PwellFloats[index1] == One ? 1 : 0;
120                     child . PwellFloats[index1] = child. PwellFloats[index2];
                        child . PwellFloats[index2] = tmp == 1 ? One: Zero;
                    }
                }

125
                //Also  Mutate  by  switching  bits
                // this  could  decrease  the  number  of  floats  which  might  drastically
                //change  the  characteristics  of  the  child
                //so  this  rate  needs  to  be  low ...
130             //but  the  check  below  will  atleast  maintain  the  floats  ...  so  mutate  away
                for (int j = 0; j < child. nwell_size () ; j++)
                {
                    if(random()%15 == 0) //7% mutation rate
                    {
135                     int tmp = random() % child.nwell_size();
```

```
                        if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                        else child.NwellFloats[tmp] = One;
                    }
                }
140             for (int j = 0; j < child.pwell_size(); j++)
                {
                    if(random()%15 == 0) //7% mutation rate
                    {
                        int tmp = random() % child.pwell_size();
145                     if(child.PwellFloats[tmp] == One) child.PwellFloats[tmp] = Zero;
                        else child.PwellFloats[tmp] = One;
                    }
                }

150
                //There is the problem that the chromosomes my eventually converge to all floats or connects
                //Observed in test runs
                //Fix this by maintaining ( within +/- 1) based on frequency of parents
                //Count the floats in parent one and two
155             int p1_nwell_floats = parents[parent_id].NwellFloatCount();
                int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
                int p1_pwell_floats = parents[parent_id].PwellFloatCount();
                int p2_pwell_floats = parents[parent_id+1].PwellFloatCount();
                int child_nwell_floats = child.NwellFloatCount();
160             int child_pwell_floats = child.PwellFloatCount();

                int avg_nwell_floats = int((p1_nwell_floats + p2_nwell_floats) / 2);
                int avg_pwell_floats = int((p1_pwell_floats + p2_pwell_floats) / 2);
                //if the average parent frequency is low, float one more than the avg floats
165             if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
                {
                    if(random() % 100 < 50){
                                    while(child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it
                                        comes with more
                                    {
170                                         int index = random() % child.nwell_size();
                                            if(child.NwellFloats[index] == One)
                                            {
                                                    child.NwellFloats[index] = Zero;
                                                    child_nwell_floats += 1;
175                                         }
                                    }
                            }
                            else {
                                    while(child_pwell_floats - avg_pwell_floats < 0) // this will make a diff of 0, and leave it if it
                                        comes with more
180                                 {
                                            int index = random() % child.pwell_size();
                                            if(child.PwellFloats[index] == One)
                                            {
                                                    child.PwellFloats[index] = Zero;
185                                                 child_pwell_floats += 1;
                                            }
                                    }
                    }
                }
190

                //if the average parent frequency is high enoug, float no more than the avg floats
                if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
                {
195                 if(random() % 100 < 50){
                                    while(child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it
                                        comes with less
                                    {
                                            int index = random() % child.nwell_size();
                                            if(child.NwellFloats[index] == Zero)
200                                         {
                                                    child.NwellFloats[index] = One;
                                                    child_nwell_floats -= 1;
                                            }
                                    }
205                         }
                            else {
                                    while(child_pwell_floats - avg_pwell_floats > 0) // this will make a diff of 0, and leave it if it
                                        comes with less
                                    {
                                            int index = random() % child.pwell_size();
210                                         if(child.PwellFloats[index] == Zero)
                                            {
                                                    child.PwellFloats[index] = One;
                                                    child_pwell_floats -= 1;
                                            }
215                                 }
                            }
                }

                //Round the biases
220             child.NwellBias = (int(child.NwellBias/BIAS_RES))*BIAS_RES;
                child.PwellBias = (int(child.PwellBias/BIAS_RES))*BIAS_RES;

                    //Apply limits to biases
                    if (child.NwellBias > MAX_NWELL) child.NwellBias = MAX_NWELL;
225                 if (child.NwellBias < MIN_NWELL) child.NwellBias = MIN_NWELL;
                    if (child.PwellBias > MAX_PWELL) child.PwellBias = MAX_PWELL;
                    if (child.PwellBias < MIN_PWELL) child.PwellBias = MIN_PWELL;
                // Evaluate child
                fitness_function.Evaluate(child, best_chromosome);
230
                //add it to the population
                population.Add(child, REPLACE2);

        }
235 }

//****************************************************************************************************************************
```

```
        else if (Reproduction_Operator == HUXR && (run_mode == NWF_DWB || run_mode == FB_NWF_DWB))
        {
240         // repeat generation for each pair of parents
            for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
            {
                //determine which parent is more fit
                int stronger_parent = 0;
245             if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                    stronger_parent = 1;

                // generate the child
                for(int j = 0; j < child.nwell_size(); j++)
250             {
                    if(parents[parent_id].NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
                        child.NwellFloats[j] = parents[parent_id].NwellFloats[j];
                    else //if the parents are different , insert bits from random parents favoring the faster one
                    {
255                     if(random() % 100 < 75){
                            child.NwellFloats[j] = parents[parent_id + stronger_parent].NwellFloats[j];
                                        }
                        else {
                            child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))].NwellFloats[j];
260                                     }
                    }
                }

                //have the NFet_Bias be:
265             //if the avg power of the parents is too high, reduce the nfet bias by 10% of the average
                //if the avg power is good, give it the avg nfet bias
                float avg_parental_pwell_bias = (parents[parent_id].PwellBias + parents[parent_id+1].PwellBias) / 2;
                            float avg_parental_nwell_bias = (parents[parent_id].NwellBias + parents[parent_id+1].NwellBias) / 2;
                if ((parents[parent_id].Power + parents[parent_id+1].Power) / 2 > TARGET_POWER) {
270                 if (random() % 100 >= 50 ){
                                        child.PwellBias = avg_parental_pwell_bias - BIAS_RES;
                                        child.NwellBias = avg_parental_nwell_bias;
                                }
                                else {
275                                     child.PwellBias = avg_parental_pwell_bias;
                                        child.NwellBias = avg_parental_nwell_bias + BIAS_RES;
                                }
                }
                else {
280                 child.PwellBias = avg_parental_pwell_bias;
                                child.NwellBias = avg_parental_nwell_bias;
                        }

                //Mutate the PwellBias
285             if(random() %100 < 10) child.PwellBias -= BIAS_RES;
                if(random() %100 < 10) child.PwellBias += BIAS_RES;

                //Mutate the NwellBias
                if(random() %100 < 10) child.NwellBias -= BIAS_RES;
290             if(random() %100 < 10) child.NwellBias += BIAS_RES;

                //Mutating by shuffling bits
                if(random() % 10 == 0) // 10% mutation rate
                {
295                 for (int j = 0; j < child.nwell_size(); j++)
                    {
                        int index1 = random()%child.nwell_size();
                        int index2 = random()%child.nwell_size();
                        int tmp = child.NwellFloats[index1] == One ? 1 : 0;
300                     child.NwellFloats[index1] = child.NwellFloats[index2];
                        child.NwellFloats[index2] = tmp == 1 ? One: Zero;
                    }
                }

305
                //Also Mutate by switching bits
                //this could decrease the number of floats which might drastically
                //change the characteristics of the child
                //so this rate needs to be low ...
310             //but the check below will atleast maintain the floats ... so mutate away
                for (int j = 0; j < child.nwell_size(); j++)
                {
                    if(random()%15 == 0) //7% mutation rate
                    {
315                     int tmp = random() % child.nwell_size();
                        if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                        else child.NwellFloats[tmp] = One;
                    }
                }
320

                //There is the problem that the chromosomes my eventually converge to all floats or connects
                //Observed in test runs
                //Fix this by maintaining ( within +/- 1) based on frequency of parents
325             //Count the floats in parent one and two
                int p1_nwell_floats = parents[parent_id].NwellFloatCount();
                int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
                int child_nwell_floats = child.NwellFloatCount();

330             int avg_nwell_floats = int((p1_nwell_floats + p2_nwell_floats) / 2);
                //if the average parent frequency is low, float one more than the avg floats
                if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
                {
                                while(child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it comes with
                                        more
335                             {
                                        int index = random() % child.nwell_size();
                                        if(child.NwellFloats[index] == One)
                                        {
                                                child.NwellFloats[index] = Zero;
340                                             child_nwell_floats += 1;
                                        }
                                }
                }
```

```
345
            //if the average parent frequency is high enoug, float no more than the avg floats
            if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
            {
                    while( child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it comes with
                         less
350                 {
                         int index = random() % child.nwell_size();
                         if(child.NwellFloats[index] == Zero)
                         {
                              child.NwellFloats[index] = One;
355                           child_nwell_floats -= 1;
                         }
                    }
            }

360         //Round biases
            child.NwellBias = (int(child.NwellBias/BIAS_RES))*BIAS_RES;
            child.PwellBias = (int(child.PwellBias/BIAS_RES))*BIAS_RES;

                    //Apply limits to biases
365         if (child.NwellBias > MAX_NWELL) child.NwellBias = MAX_NWELL;
            if (child.NwellBias < MIN_NWELL) child.NwellBias = MIN_NWELL;
            if (child.PwellBias > MAX_PWELL) child.PwellBias = MAX_PWELL;
            if (child.PwellBias < MIN_PWELL) child.PwellBias = MIN_PWELL;

370         //Evaluate child
            fitness_function.Evaluate(child, best_chromosome);

            //add it to the population
            population.Add(child, REPLACE2);
375
        }
    }

//***********************************************************************************************************************

380
        else if(Reproduction_Operator == HUXR && run_mode == FB_DWF_DWB)
    {
        //repeat generation for each pair of parents
        for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
385     {
            //determine which parent is more fit
            int stronger_parent = 0;
            if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                stronger_parent = 1;
390
            //generate the child
            for(int j = 0; j < child.nwell_size(); j++)
            {
                if(parents[parent_id].NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
395                 child.NwellFloats[j] = parents[parent_id].NwellFloats[j];
                else //if the parents are different , insert bits from random parents favoring the faster one
                {
                    if(random() % 100 < 75){
                        child.NwellFloats[j] = parents[parent_id + stronger_parent].NwellFloats[j];
400                                 }
                    else {
                        child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))].NwellFloats[j];
                                    }
                }
405         }
                    for(int j = 0; j < child.pwell_size(); j++)
            {
                if(parents[parent_id].PwellFloats[j] == parents[parent_id+1].PwellFloats[j])
                    child.PwellFloats[j] = parents[parent_id].PwellFloats[j];
410             else //if the parents are different , insert bits from random parents favoring the faster one
                {
                    if(random() % 100 < 75){
                        child.PwellFloats[j] = parents[parent_id + stronger_parent].PwellFloats[j];
                                    }
415                 else {
                        child.PwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))].PwellFloats[j];
                                    }
                }
            }
420


            //have the NFet_Bias be:
            //if the avg power of the parents is too high, reduce the nfet bias by 10% of the average
425         //if the avg power is good, give it the avg nfet bias
            float avg_parental_pwell_bias = (parents[parent_id].PwellBias + parents[parent_id+1].PwellBias) / 2;
                    float avg_parental_nwell_bias = (parents[parent_id].NwellBias + parents[parent_id+1].NwellBias) / 2;
            if ((parents[parent_id].Power + parents[parent_id+1].Power) / 2 > TARGET_POWER) {
                if (random() % 100 >= 50 ){
430                              child.PwellBias = avg_parental_pwell_bias - BIAS_RES;
                                 child.NwellBias = avg_parental_nwell_bias;
                        }
                        else {
                                 child.PwellBias = avg_parental_pwell_bias;
435                              child.NwellBias = avg_parental_nwell_bias + BIAS_RES;
                        }
            }
            else {
                if (random() % 100 >= 50 ){
440                              child.PwellBias = avg_parental_pwell_bias + BIAS_RES;
                                 child.NwellBias = avg_parental_nwell_bias;
                        }
                        else {
                                 child.PwellBias = avg_parental_pwell_bias;
445                              child.NwellBias = avg_parental_nwell_bias - BIAS_RES;
                        }
                    }
```

```
        //Mutate the PwellBias
450     if(random() %100 < 10) child.PwellBias -= BIAS_RES;
        if(random() %100 < 10) child.PwellBias += BIAS_RES;

        //Mutate the NwellBias
        if(random() %100 < 10) child.NwellBias -= BIAS_RES;
455     if(random() %100 < 10) child.NwellBias += BIAS_RES;

        //Mutating by shuffling bits
        if(random() % 10 == 0) // 10% mutation rate
        {
460         for (int j = 0; j < child.nwell_size(); j++)
            {
                int index1 = random()%child.nwell_size();
                int index2 = random()%child.nwell_size();
                int tmp = child.NwellFloats[index1] == One ? 1 : 0;
465             child.NwellFloats[index1] = child.NwellFloats[index2];
                child.NwellFloats[index2] = tmp == 1 ? One: Zero;
            }
        }
        if(random() % 10 == 0) // 10% mutation rate
470     {
            for (int j = 0; j < child.pwell_size(); j++)
            {
                int index1 = random()%child.pwell_size();
                int index2 = random()%child.pwell_size();
475             int tmp = child.PwellFloats[index1] == One ? 1 : 0;
                child.PwellFloats[index1] = child.PwellFloats[index2];
                child.PwellFloats[index2] = tmp == 1 ? One: Zero;
            }
        }
480

        //Also Mutate by switching bits
        //this could decrease the number of floats which might drastically
        //change the characteristics of the child
485     //so this rate needs to be low ...
        //but the check below will atleast maintain the floats ... so mutate away
        for (int j = 0; j < child.nwell_size(); j++)
        {
            if(random()%15 == 0) //7% mutation rate
490         {
                int tmp = random() % child.nwell_size();
                if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                else child.NwellFloats[tmp] = One;
            }
495     }
        for (int j = 0; j < child.pwell_size(); j++)
        {
            if(random()%15 == 0) //7% mutation rate
            {
500             int tmp = random() % child.pwell_size();
                if(child.PwellFloats[tmp] == One) child.PwellFloats[tmp] = Zero;
                else child.PwellFloats[tmp] = One;
            }
        }
505

        //There is the problem that the chromosomes my eventually converge to all floats or connects
        //Observed in test runs
        //Fix this by maintaining ( within +/- 1) based on frequency of parents
510     //Count the floats in parent one and two
        int p1_nwell_floats = parents[parent_id].NwellFloatCount();
        int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
        int p1_pwell_floats = parents[parent_id].PwellFloatCount();
        int p2_pwell_floats = parents[parent_id+1].PwellFloatCount();
515     int child_nwell_floats = child.NwellFloatCount();
        int child_pwell_floats = child.PwellFloatCount();

        int avg_nwell_floats = int((p1_nwell_floats + p2_nwell_floats) / 2);
        int avg_pwell_floats = int((p1_pwell_floats + p2_pwell_floats) / 2);
520     //if the average parent frequency is low, float one more than the avg floats
        if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
        {
            if(random() % 100 < 50){
                                while(child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it
                                        comes with more
525                             {
                                        int index = random() % child.nwell_size();
                                        if(child.NwellFloats[index] == One)
                                        {
                                                child.NwellFloats[index] = Zero;
530                                             child_nwell_floats += 1;
                                        }
                                }
                        }
                        else {
535                             while(child_pwell_floats - avg_pwell_floats < 0) // this will make a diff of 0, and leave it if it
                                        comes with more
                                {
                                        int index = random() % child.pwell_size();
                                        if(child.PwellFloats[index] == One)
                                        {
540                                             child.PwellFloats[index] = Zero;
                                                child_pwell_floats += 1;
                                        }
                                }
            }
545     }


        //if the average parent frequency is high enoug, float no more than the avg floats
        if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
550     {
            if(random() % 100 < 50){
                                while(child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it
                                        comes with less
```

```
                                        {
                                                int index = random() % child.nwell_size();
555                                             if(child.NwellFloats[index] == Zero)
                                                {
                                                        child.NwellFloats[index] = One;
                                                        child_nwell_floats  −= 1;
                                                }
560                                     }
                                }
                                else {
                                        while( child_pwell_floats − avg_pwell_floats > 0) // this will make a diff of 0, and leave it if it
                                                comes with less
                                        {
565                                             int index = random() % child.pwell_size();
                                                if(child.PwellFloats[index] == Zero)
                                                {
                                                        child.PwellFloats[index] = One;
                                                        child_pwell_floats  −= 1;
570                                             }
                                        }
                                }
                }

575             //Round biases
                child.NwellBias = (int(child.NwellBias/BIAS_RES))∗BIAS_RES;
                child.PwellBias = (int(child.PwellBias/BIAS_RES))∗BIAS_RES;

                    //Apply limits to biases
580                 if (child.NwellBias > MAX_NWELL) child.NwellBias = MAX_NWELL;
                    if (child.NwellBias < MIN_NWELL) child.NwellBias = MIN_NWELL;
                    if (child.PwellBias > MAX_PWELL) child.PwellBias = MAX_PWELL;
                    if (child.PwellBias < MIN_PWELL) child.PwellBias = MIN_PWELL;
                // Evaluate child
585             fitness_function.Evaluate(child, best_chromosome);

                //add it to the population
                population.Add(child, REPLACE2);

590         }
        }

    //****************************************************************************************************************************

        else if (Reproduction_Operator == HUXR && (run_mode == NWF_NWB || run_mode == FB_NWF_NWB))
595     {
            // repeat generation for each pair of parents
            for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
            {
                // determine which parent is more fit
600             int stronger_parent = 0;
                if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                    stronger_parent = 1;

                // generate the child
605             for(int j = 0; j < child.nwell_size(); j++)
                {
                    if(parents[parent_id].NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
                        child.NwellFloats[j] = parents[parent_id].NwellFloats[j];
                    else // if the parents are different , insert bits from random parents favoring the faster one
610                 {
                        if(random() % 100 < 75){
                            child.NwellFloats[j] = parents[parent_id + stronger_parent].NwellFloats[j];
                                        }
                        else {
615                         child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent − 1)))].NwellFloats[j];
                                        }
                    }
                }

620             //have the NFet_Bias be:
                // if the avg power of the parents is too high, reduce the nfet bias by 10% of the average
                // if the avg power is good, give it the avg nfet bias
                        float avg_parental_nwell_bias = (parents[parent_id].NwellBias + parents[parent_id+1].NwellBias) / 2;
                if ((parents[parent_id].Power + parents[parent_id+1].Power) / 2 > TARGET_POWER)
625                             child.NwellBias = avg_parental_nwell_bias + BIAS_RES;
                else child.NwellBias = avg_parental_nwell_bias;

                //Mutate the NwellBias
                if(random() %100 < 10) child.NwellBias −= BIAS_RES;
630             if(random() %100 < 10) child.NwellBias += BIAS_RES;

                //Mutating by shuffling bits
                if(random() % 10 == 0) // 10% mutation rate
                {
635                 for (int j = 0; j < child.nwell_size(); j++)
                    {
                        int index1 = random()%child.nwell_size();
                        int index2 = random()%child.nwell_size();
                        int tmp = child.NwellFloats[index1] == One ? 1 : 0;
640                     child.NwellFloats[index1] = child.NwellFloats[index2];
                        child.NwellFloats[index2] = tmp == 1 ? One: Zero;
                    }
                }

645
                //Also Mutate by switching bits
                // this could decrease the number of floats which might drastically
                //change the characteristics of the child
                //so this rate needs to be low ...
650             //but the check below will atleast maintain the floats ... so mutate away
                for (int j = 0; j < child.nwell_size(); j++)
                {
                    if(random()%15 == 0) //7% mutation rate
                    {
655                     int tmp = random() % child.nwell_size();
                        if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                        else child.NwellFloats[tmp] = One;
```

```
                }
            }
660
            //There is the problem that the chromosomes my eventually converge to all floats or connects
            //Observed in test runs
            //Fix this by maintaining ( within +/- 1) based on frequency of parents
665         //Count the floats in parent one and two
            int p1_nwell_floats = parents[parent_id ].NwellFloatCount();
            int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
            int child_nwell_floats = child.NwellFloatCount();

670         int avg_nwell_floats = int(( p1_nwell_floats + p2_nwell_floats) / 2);
            //if the average parent frequency is low, float one more than the avg floats
            if((parents[parent_id ]. Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
            {
                        while( child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it comes with
                            more
675                     {
                            int index = random() % child.nwell_size();
                            if(child.NwellFloats[index] == One)
                            {
                                child.NwellFloats[index] = Zero;
680                             child_nwell_floats += 1;
                            }
                        }
            }

685
            //if the average parent frequency is high enoug, float no more than the avg floats
            if((parents[parent_id ]. Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
            {
                        while( child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it comes with
                            less
690                     {
                            int index = random() % child.nwell_size();
                            if(child.NwellFloats[index] == Zero)
                            {
                                child.NwellFloats[index] = One;
695                             child_nwell_floats -= 1;
                            }
                        }
            }

700         //Round biases
            child.NwellBias = (int(child.NwellBias/BIAS_RES))*BIAS_RES;

                    //Apply limits to biases
                if ( child.NwellBias > MAX_NWELL) child.NwellBias = MAX_NWELL;
705             if ( child.NwellBias < MIN_NWELL) child.NwellBias = MIN_NWELL;

            // Evaluate child
            fitness_function.Evaluate(child, best_chromosome);

710         //add it to the population
            population.Add(child, REPLACE2);

        }
    }
715
//********************************************************************************************************************

    else if (Reproduction_Operator == HUXR && (run_mode == NWF_PWB || run_mode == FB_NWF_PWB))
    {
        // repeat generation for each pair of parents
720     for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
        {
            //determine which parent is more fit
            int stronger_parent = 0;
            if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
725             stronger_parent = 1;

            // generate the child
            for(int j = 0; j < child. nwell_size(); j++)
            {
730             if(parents[parent_id ]. NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
                    child.NwellFloats[j] = parents[parent_id ]. NwellFloats[j ];
                else //if the parents are different , insert bits from random parents favoring the faster one
                {
                    if(random() % 100 < 75){
735                     child.NwellFloats[j] = parents[parent_id + stronger_parent]. NwellFloats[j ];
                            }
                    else {
                        child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))]. NwellFloats[j ];
                            }
740             }
            }

            //have the NFet_Bias be:
            //if the avg power of the parents is too high, reduce the nfet bias by 10% of the average
745         //if the avg power is good, give it the avg nfet bias
            float avg_parental_pwell_bias = (parents[parent_id ]. PwellBias + parents[parent_id+1].PwellBias) / 2;
            if ((parents[parent_id ]. Power + parents[parent_id+1].Power) / 2 > TARGET_POWER)
                        child.PwellBias = avg_parental_pwell_bias - BIAS_RES;
            else child.PwellBias = avg_parental_pwell_bias;
750

            //Mutate the PwellBias
            if(random() %100 < 10) child.PwellBias -= BIAS_RES;
            if(random() %100 < 10) child.PwellBias += BIAS_RES;
755
            //Mutating by shuffling bits
            if(random() % 10 == 0) // 10% mutation rate
            {
                for (int j = 0; j < child. nwell_size(); j++)
760             {
                    int index1 = random()%child.nwell_size();
```

```
                    int index2 = random()%child.nwell_size();
                    int tmp = child.NwellFloats[index1] == One ? 1 : 0;
                    child.NwellFloats[index1] = child.NwellFloats[index2];
765                 child.NwellFloats[index2] = tmp == 1 ? One: Zero;
                }
            }


770         //Also Mutate by switching bits
            //this could decrease the number of floats which might drastically
            //change the characteristics of the child
            //so this rate needs to be low...
            //but the check below will atleast maintain the floats ... so mutate away
775         for (int j = 0; j < child.nwell_size(); j++)
            {
                if(random()%15 == 0) //7% mutation rate
                {
                    int tmp = random() % child.nwell_size();
780                 if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                    else child.NwellFloats[tmp] = One;
                }
            }

785
            //There is the problem that the chromosomes my eventually converge to all floats or connects
            //Observed in test runs
            //Fix this by maintaining (within +/- 1) based on frequency of parents
            //Count the floats in parent one and two
790         int p1_nwell_floats = parents[parent_id].NwellFloatCount();
            int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
            int child_nwell_floats = child.NwellFloatCount();

            int avg_nwell_floats = int((p1_nwell_floats + p2_nwell_floats) / 2);
795         //if the average parent frequency is low, float one more than the avg floats
            if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
            {
                        while( child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it comes with
                                more
                        {
800                             int index = random() % child.nwell_size();
                                if(child.NwellFloats[index] == One)
                                {
                                        child.NwellFloats[index] = Zero;
                                        child_nwell_floats += 1;
805                             }
                        }
            }


810         //if the average parent frequency is high enoug, float no more than the avg floats
            if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
            {
                        while( child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it comes with
                                less
                        {
815                             int index = random() % child.nwell_size();
                                if(child.NwellFloats[index] == Zero)
                                {
                                        child.NwellFloats[index] = One;
                                        child_nwell_floats -= 1;
820                             }
                        }
            }

            //Round biases
825         child.PwellBias = (int(child.PwellBias/BIAS_RES))*BIAS_RES;

                        //Apply limits to biases
                if (child.PwellBias > MAX_PWELL) child.PwellBias = MAX_PWELL;
                if (child.PwellBias < MIN_PWELL) child.PwellBias = MIN_PWELL;
830
            //Evaluate child
            fitness_function.Evaluate(child, best_chromosome);

            //add it to the population
835         population.Add(child, REPLACE2);

        }
    }

840 //*****************************************************************************************************************************
    else if (Reproduction_Operator == HUXR && run_mode == NWF_PWB_VDD )
    {
        //repeat generation for each pair of parents
        for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
845     {
            //determine which parent is more fit
            int stronger_parent = 0;
            if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                stronger_parent = 1;
850
            //generate the child
            for(int j = 0; j < child.nwell_size(); j++)
            {
                if(parents[parent_id].NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
855                 child.NwellFloats[j] = parents[parent_id].NwellFloats[j];
                else //if the parents are different , insert bits from random parents favoring the faster one
                {
                    if(random() % 100 < 75){
                        child.NwellFloats[j] = parents[parent_id + stronger_parent].NwellFloats[j];
860                                     }
                    else {
                        child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))].NwellFloats[j];
                                        }
                }
865         }
```

```
        //have the NFet_Bias be:
        //if the avg power of the parents is too high, reduce the nfet bias by 10% of the average
        //if the avg power is good, give it the avg nfet bias
870     float avg_parental_pwell_bias = (parents[parent_id].PwellBias + parents[parent_id+1].PwellBias) / 2;
        float avg_parental_VDD = (parents[parent_id].VDD + parents[parent_id+1].VDD) / 2;
        if ((parents[parent_id].Power + parents[parent_id+1].Power) / 2 > TARGET_POWER) {
            if (random() % 100 >= 50 ){
                                child.PwellBias = avg_parental_pwell_bias - BIAS_RES;
875                             child.VDD = avg_parental_VDD;
                        }
                        else {
                                child.PwellBias = avg_parental_pwell_bias;
                                child.VDD= avg_parental_VDD - BIAS_RES;
880                     }
        }
        else {
            child.PwellBias = avg_parental_pwell_bias;
                        child.VDD = avg_parental_VDD;
885             }

        //Mutate the PwellBias
        if(random() %100 < 10) child.PwellBias -= BIAS_RES;
        if(random() %100 < 10) child.PwellBias += BIAS_RES;
890
        //Mutate the VDD
        if(random() %100 < 10) child.VDD -= BIAS_RES;
        if(random() %100 < 10) child.VDD += BIAS_RES;

895     //Mutating by shuffling bits
        if(random() % 10 == 0) // 10% mutation rate
        {
            for (int j = 0; j < child.nwell_size(); j++)
            {
900             int index1 = random()%child.nwell_size();
                int index2 = random()%child.nwell_size();
                int tmp = child.NwellFloats[index1] == One ? 1 : 0;
                child.NwellFloats[index1] = child.NwellFloats[index2];
                child.NwellFloats[index2] = tmp == 1 ? One: Zero;
905         }
        }


        //Also Mutate by switching bits
910     //this could decrease the number of floats which might drastically
        //change the characteristics of the child
        //so this rate needs to be low ...
        //but the check below will atleast maintain the floats ... so mutate away
        for (int j = 0; j < child.nwell_size(); j++)
915     {
            if(random()%15 == 0) //7% mutation rate
            {
                int tmp = random() % child.nwell_size();
                if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
920             else child.NwellFloats[tmp] = One;
            }
        }


925     //There is the problem that the chromosomes my eventually converge to all floats or connects
        //Observed in test runs
        //Fix this by maintaining (within +/- 1) based on frequency of parents
        //Count the floats in parent one and two
        int p1_nwell_floats = parents[parent_id].NwellFloatCount();
930     int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
        int child_nwell_floats = child.NwellFloatCount();

        int avg_nwell_floats = int((p1_nwell_floats + p2_nwell_floats) / 2);
        //if the average parent frequency is low, float one more than the avg floats
935     if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
        {
                        while(child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it comes with
                                more
                        {
                                int index = random() % child.nwell_size();
940                             if(child.NwellFloats[index] == One)
                                {
                                        child.NwellFloats[index] = Zero;
                                        child_nwell_floats += 1;
                                }
945                     }
        }


        //if the average parent frequency is high enoug, float no more than the avg floats
950     if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
        {
                        while(child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it comes with
                                less
                        {
                                int index = random() % child.nwell_size();
955                             if(child.NwellFloats[index] == Zero)
                                {
                                        child.NwellFloats[index] = One;
                                        child_nwell_floats -= 1;
                                }
960                     }
        }

        //Round biases
        child.VDD = (int(child.VDD/BIAS_RES))*BIAS_RES;
965     child.PwellBias = (int(child.PwellBias/BIAS_RES))*BIAS_RES;

                //Apply limits to biases
            if (child.VDD > MAX_VDD) child.VDD = MAX_VDD;
            if (child.VDD < MIN_VDD) child.VDD = MIN_VDD;
970         if (child.PwellBias > MAX_PWELL) child.PwellBias = MAX_PWELL;
```

```
                if (child.PwellBias < MIN_PWELL) child.PwellBias = MIN_PWELL;

            // Evaluate child
            fitness_function.Evaluate(child, best_chromosome);
975
            //add it to the population
            population.Add(child, REPLACE2);

        }
980   }

//*****************************************************************************************************************************************

    else if (Reproduction_Operator == HUXR && run_mode == NWF_NWB_VDD )
    {
985     // repeat generation for each pair of parents
        for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
        {
            // determine which parent is more fit
            int stronger_parent = 0;
990         if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                stronger_parent = 1;

            // generate the child
            for(int j = 0; j < child.nwell_size(); j++)
995         {
                if(parents[parent_id].NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
                    child.NwellFloats[j] = parents[parent_id].NwellFloats[j];
                else // if the parents are different , insert bits from random parents favoring the faster one
                {
1000                if(random() % 100 < 75){
                        child.NwellFloats[j] = parents[parent_id + stronger_parent].NwellFloats[j];
                                        }
                    else {
                        child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))].NwellFloats[j];
1005                                    }
                }
            }

            //have the NFet_Bias be:
1010        // if the avg power of the parents is too high, reduce the nfet bias by 10% of the average
            // if the avg power is good, give it the avg nfet bias
            float avg_parental_nwell_bias = (parents[parent_id].NwellBias + parents[parent_id+1].NwellBias) / 2;
            float avg_parental_VDD = (parents[parent_id].VDD + parents[parent_id+1].VDD) / 2;
            if ((parents[parent_id].Power + parents[parent_id+1].Power) / 2 > TARGET_POWER) {
1015            if (random() % 100 >= 50 ){
                                        child.NwellBias = avg_parental_nwell_bias - BIAS_RES;
                                        child.VDD = avg_parental_VDD;
                                }
                                else {
1020                                    child.NwellBias = avg_parental_nwell_bias;
                                        child.VDD= avg_parental_VDD - BIAS_RES;
                                }
            }
            else {
1025            child.NwellBias = avg_parental_nwell_bias;
                                child.VDD = avg_parental_VDD;
                        }

            //Mutate the PwellBias
1030        if(random() %100 < 10) child.NwellBias -= BIAS_RES;
            if(random() %100 < 10) child.NwellBias += BIAS_RES;

            //Mutate the VDD
            if(random() %100 < 10) child.VDD -= BIAS_RES;
1035        if(random() %100 < 10) child.VDD += BIAS_RES;

            //Mutating by shuffling bits
            if(random() % 10 == 0) // 10% mutation rate
            {
1040            for (int j = 0; j < child.nwell_size(); j++)
                {
                    int index1 = random()%child.nwell_size();
                    int index2 = random()%child.nwell_size();
                    int tmp = child.NwellFloats[index1] == One ? 1 : 0;
1045                child.NwellFloats[index1] = child.NwellFloats[index2];
                    child.NwellFloats[index2] = tmp == 1 ? One: Zero;
                }
            }

1050
            //Also Mutate by switching bits
            // this could decrease the number of floats which might drastically
            //change the characteristics of the child
            //so this rate needs to be low ...
1055        //but the check below will atleast maintain the floats ... so mutate away
            for (int j = 0; j < child.nwell_size(); j++)
            {
                if(random()%15 == 0) //7% mutation rate
                {
1060                int tmp = random() % child.nwell_size();
                    if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                    else child.NwellFloats[tmp] = One;
                }
            }
1065


            //There is the problem that the chromosomes my eventually converge to all floats or connects
            // Observed in test runs
            //Fix this by maintaining ( within +/- 1) based on frequency of parents
1070        //Count the floats in parent one and two
            int p1_nwell_floats = parents[parent_id].NwellFloatCount();
            int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
            int child_nwell_floats = child.NwellFloatCount();

1075        int avg_nwell_floats = int((p1_nwell_floats + p2_nwell_floats) / 2);
            // if the average parent frequency is low, float one more than the avg floats
```

```
            if((parents[parent_id]. Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
            {
                        while( child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it comes with
                            more
1080                    {
                            int index = random() % child.nwell_size();
                            if(child.NwellFloats[index] == One)
                            {
                                child.NwellFloats[index] = Zero;
1085                            child_nwell_floats += 1;
                            }
                        }
            }

1090
            //if the average parent frequency is high enoug, float no more than the avg floats
            if((parents[parent_id]. Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
            {
                        while( child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it comes with
                            less
1095                    {
                            int index = random() % child.nwell_size();
                            if(child.NwellFloats[index] == Zero)
                            {
                                child.NwellFloats[index] = One;
1100                            child_nwell_floats -= 1;
                            }
                        }
            }

1105        //Round biases
            child.VDD = (int(child.VDD/BIAS_RES))*BIAS_RES;
            child.NwellBias = (int(child.NwellBias/BIAS_RES))*BIAS_RES;

                    //Apply limits to biases
1110          if (child.VDD > MAX_VDD) child.VDD = MAX_VDD;
              if (child.VDD < MIN_VDD) child.VDD = MIN_VDD;
              if (child.NwellBias > MAX_NWELL) child.NwellBias = MAX_NWELL;
              if (child.NwellBias < MIN_NWELL) child.NwellBias = MIN_NWELL;

1115        //Evaluate child
            fitness_function.Evaluate(child, best_chromosome);

            //add it to the population
            population.Add(child, REPLACE2);
1120
        }
    }

  //*******************************************************************************************************************

1125    else if (Reproduction_Operator == HUXR && run_mode == NWF)
    {
        //repeat generation for each pair of parents
        for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
        {
1130        //determine which parent is more fit
            int stronger_parent = 0;
            if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                stronger_parent = 1;

1135        //generate the child
            for(int j = 0; j < child.nwell_size(); j++)
            {
                if(parents[parent_id].NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
                    child.NwellFloats[j] = parents[parent_id].NwellFloats[j];
1140            else //if the parents are different , insert bits from random parents favoring the faster one
                {
                    if(random() % 100 < 75){
                        child.NwellFloats[j] = parents[parent_id + stronger_parent].NwellFloats[j];
                                        }
1145                else {
                        child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))].NwellFloats[j];
                                        }
                }
            }
1150
            //Mutating by shuffling bits
            if(random() % 10 == 0) // 10% mutation rate
            {
                for (int j = 0; j < child.nwell_size(); j++)
1155            {
                    int index1 = random()%child.nwell_size();
                    int index2 = random()%child.nwell_size();
                    int tmp = child.NwellFloats[index1] == One ? 1 : 0;
                    child.NwellFloats[index1] = child.NwellFloats[index2];
1160                child.NwellFloats[index2] = tmp == 1 ? One: Zero;
                }
            }


1165        //Also Mutate by switching bits
            //this could decrease the number of floats which might drastically
            //change the characteristics of the child
            //so this rate needs to be low ...
            //but the check below will atleast maintain the floats ... so mutate away
1170        for (int j = 0; j < child.nwell_size(); j++)
            {
                if(random()%15 == 0) //7% mutation rate
                {
                    int tmp = random() % child.nwell_size();
1175                if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                    else child.NwellFloats[tmp] = One;
                }
            }

1180
```

```
            //There is the problem that the chromosomes my eventually converge to all floats or connects
            //Observed in test runs
            //Fix this by maintaining ( within +/- 1) based on frequency of parents
            //Count the floats in parent one and two
1185        int p1_nwell_floats = parents[parent_id].NwellFloatCount();
            int p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
            int child_nwell_floats = child.NwellFloatCount();

            int avg_nwell_floats = int((p1_nwell_floats + p2_nwell_floats) / 2);
1190        //if the average parent frequency is low, float one more than the avg floats
            if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
            {
                        while(child_nwell_floats - avg_nwell_floats < 0) // this will make a diff of 0, and leave it if it comes with
                            more
                        {
1195                        int index = random() % child.nwell_size();
                            if(child.NwellFloats[index] == One)
                            {
                                child.NwellFloats[index] = Zero;
                                child_nwell_floats += 1;
1200                        }
                        }
            }


1205        //if the average parent frequency is high enoug, float no more than the avg floats
            if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
            {
                        while(child_nwell_floats - avg_nwell_floats > 0) // this will make a diff of 0, and leave it if it comes with
                            less
                        {
1210                        int index = random() % child.nwell_size();
                            if(child.NwellFloats[index] == Zero)
                            {
                                child.NwellFloats[index] = One;
                                child_nwell_floats -= 1;
1215                        }
                        }
            }

            //Evaluate child
1220        fitness_function.Evaluate(child, best_chromosome);

            //add it to the population
            population.Add(child, REPLACE2);

1225    }
    }

//*****************************************************************************************************************************

    else if (Reproduction_Operator == HUXR && (run_mode == NWF_VDD || run_mode == FB_NWF_VDD))
1230 {
        //repeat generation for each pair of parents
        for(int parent_id = 0; parent_id < parents.size(); parent_id+=2)
        {
            //determine which parent is more fit
1235        int stronger_parent = 0;
            if(parents[parent_id+1].Frequency > parents[parent_id].Frequency)
                stronger_parent = 1;

            //generate the child
1240        for(int j = 0; j < child.nwell_size(); j++)
            {
                if(parents[parent_id].NwellFloats[j] == parents[parent_id+1].NwellFloats[j])
                    child.NwellFloats[j] = parents[parent_id].NwellFloats[j];
                else //if the parents are different, insert bits from random parents favoring the faster one
1245            {
                    if(random() % 100 < 75){
                        child.NwellFloats[j] = parents[parent_id + stronger_parent].NwellFloats[j];
                                }
                    else {
1250                    child.NwellFloats[j] = parents[parent_id + int(fabs(float(stronger_parent - 1)))].NwellFloats[j];
                                }
                }
            }

1255        //have the VDD_Bias be:
            //if the avg power of the parents is too high, reduce the nfet bias by 10% of the average
            //if the avg power is good, give it the avg nfet bias
            float avg_parental_VDD = (parents[parent_id].VDD + parents[parent_id+1].VDD) / 2;
            if ((parents[parent_id].Power + parents[parent_id+1].Power) / 2 > TARGET_POWER)
1260                    child.VDD = avg_parental_VDD - BIAS_RES;
            else child.VDD = avg_parental_VDD;


            //Mutate the VDD
1265        if(random() %100 < 10) child.VDD -= BIAS_RES;
            if(random() %100 < 10) child.VDD += BIAS_RES;

            //Mutating by shuffling bits
            if(random() % 10 == 0) // 10% mutation rate
1270        {
                for (int j = 0; j < child.nwell_size(); j++)
                {
                    int index1 = random()%child.nwell_size();
                    int index2 = random()%child.nwell_size();
1275                int tmp = child.NwellFloats[index1] == One ? 1 : 0;
                    child.NwellFloats[index1] = child.NwellFloats[index2];
                    child.NwellFloats[index2] = tmp == 1 ? One: Zero;
                }
            }
1280

            //Also Mutate by switching bits
            //this could decrease the number of floats which might drastically
            //change the characteristics of the child
```

```
1285            //so  this  rate  needs  to  be  low ...
                //but  the  check  below  will  atleast  maintain  the  floats  ...  so  mutate  away
                for (int j = 0; j < child.nwell_size(); j++)
                {
                    if(random()%15 == 0) //7% mutation rate
1290                {
                        int tmp = random() % child.nwell_size();
                        if(child.NwellFloats[tmp] == One) child.NwellFloats[tmp] = Zero;
                        else child.NwellFloats[tmp] = One;
                    }
1295            }


                //There  is  the  problem  that  the  chromosomes my  eventually  converge  to  all  floats  or  connects
                //Observed  in  test  runs
1300            //Fix  this  by  maintaining  ( within +/- 1)  based  on  frequency  of  parents
                //Count  the  floats  in  parent  one and  two
                int  p1_nwell_floats = parents[parent_id].NwellFloatCount();
                int  p2_nwell_floats = parents[parent_id+1].NwellFloatCount();
                int  child_nwell_floats  = child.NwellFloatCount();
1305
                int  avg_nwell_floats = int(( p1_nwell_floats + p2_nwell_floats) / 2);
                //if  the  average  parent  frequency  is  low,  float  one  more  than  the  avg  floats
                if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 < TARGET_FREQ)
                {
1310                        while( child_nwell_floats - avg_nwell_floats < 0) // this  will  make a  diff  of  0,  and  leave  it  if  it  comes  with
                                    more
                            {
                                    int index = random() % child.nwell_size();
                                    if(child.NwellFloats[index] == One)
                                    {
1315                                        child.NwellFloats[index] = Zero;
                                            child_nwell_floats  += 1;
                                    }
                            }
                }
1320

                //if  the  average  parent  frequency  is  high  enoug,  float  no  more  than  the  avg  floats
                if((parents[parent_id].Frequency + parents[parent_id+1].Frequency) / 2 > TARGET_FREQ)
                {
1325                        while( child_nwell_floats - avg_nwell_floats > 0) // this  will  make a  diff  of  0,  and  leave  it  if  it  comes  with
                                    less
                            {
                                    int index = random() % child.nwell_size();
                                    if(child.NwellFloats[index] == Zero)
                                    {
1330                                        child.NwellFloats[index] = One;
                                            child_nwell_floats  -= 1;
                                    }
                            }
                }
1335
                //Round  biases
                child.VDD = (int(child.VDD/BIAS_RES))*BIAS_RES;

                            //Apply  limits  to  biases
1340                if  (child.VDD > MAX_VDD) child.VDD = MAX_VDD;
                    if  (child.VDD < MIN_VDD) child.VDD = MIN_VDD;

                //Evaluate  child
                fitness_function.Evaluate(child,  best_chromosome);
1345
                //add  it  to  the  population
                population.Add(child, REPLACE2);

            }
1350    }

//**************************************************************************************************************************

        else
        {
1355        printf("ReproductionFunction::Reproduce - Bad Operator or run mode\n");
            return;
        }
}
```

```
#include "SelectionFunction.h"



5   SelectionFunction::SelectionFunction(SelectionMode selection_mode)
    {
        Selection_Mode = selection_mode;
    }

10


    SelectionFunction::~SelectionFunction()
    {
    }
15


    void SelectionFunction::Select(const Population& population, int parent_count, Population& parents)
    {
20      int i;
        int index1;
        int index2;
        int index;

25      //srand(time(NULL));
        parents.clear();
```

```
        if(Selection_Mode == TOURNAMENT)
        {
30         for(i = 0; i < parent_count; i++)
           {
              index1 = random() % population.size();
              index2 = random() % population.size();
              if(population[index1].Fitness > population[index2].Fitness)
35               index = index1;
              else
                 index = index2;
              parents.push_back(population[index]);
           }
40
        }
        else
        {
           printf("SelectionFunction::Select - Invalid selection mode\n");
45      }
     }
```

Appendix A

gIWABB source code

Header file

```
//================================================================
//aNWF.h Header file for aggressive nwell floating
//Author: Justin Gregg
//History:
//Created 2003.05.28
//================================================================

#ifndef aNWF_H
#define aNWF_H

#include <stdlib.h>
#include <stdio.h>
#include <string>
#include <fstream>
#include <vector>
#include <math.h>
#include <time.h>
#undef system

using namespace std;

//=============== constants ======================================
const int chrom_length = 32;

float vdd_min = 0.6;
float vdd_max = 1.6;
float nwell_bias_min = 0.6;
float nwell_bias_max = 1.6;
float pwell_bias_min = -0.5;
float pwell_bias_max = 0.5;

float avg_power=4.4314e-4;
float avg_freq =2.8808e8;

char* case_name = "add";
char* spice_path = "/net/etltrk6/mnt/a/iabb/adder32_V5/spice";
char* runinput_path = "/net/etltrk6/mnt/a/iabb/adder32_V5/spice/runinputs.vdiv3_0x1r0";
char* tail_filename = "tail.runinput.vdiv3";
char* run_ispice = ". /net/etltrk6/mnt/a/iabb/source_path/shbp; ispice -j -d -c %s%i /net/etltrk6/mnt/a/iabb/adder32_V5"; //need
      case_name, case_num
char* run_hpspice = ". /net/etltrk6/mnt/a/iabb/source_path/shbp; cd %s/case.%s%i; hpspice -p raw alias -s -nowindows -c \"include %s/
      measure\"";

//=============== structures =====================================
typedef struct ind_st //an individual structure
{
        int chip;
        float pwell_bias, nwell_bias, vdd_bias;
        int chrom[chrom_length]; // 1 is connected 0 is floated
        double fitness, frequency, power;
        double dPower[chrom_length], dFrequency[chrom_length]; //change in f and p for floating each well
} Ind;

//=============== other common parameters ========================
const int num_of_yields = 5;
float target_frequency, target_power;

int start_chip, final_chip, chip_num, yield_num;
float bias_res;
Ind best_ind;
int run_mode, case_num, max_step;

vector<Ind> ind_cache_vec; //cache of evaluated inds
int spice_evals, good_chips;

//=============== files ==========================================
FILE* best_file;
FILE* cache_file;
FILE* eval_file;

//=============== functions ======================================
void evaluate(Ind& ind); //writes the cserun.runinput file, runs ispice and hpspice. Fills ind's p, f, fitness, quad
int is_equal(Ind ind_a, Ind ind_b); //determines if two inds are the same
Ind new_ind();
```

```
int  best_well_to_float (Ind& ind);
int  best_well_to_connect(Ind& ind);

void CON();
void NWF_DWB();
void NWF_PWB();
void NWF_NWB();
void DWB();
void VDD();
void NWF_VDD();
void NWF_PWB_VDD();
void PWB_VDD();
void NWB_VDD();
void NWB();
void NWF();
void NWF_NWB_VDD();

void write_ind(Ind ind, FILE* file_ptr);
void write_eval(int this_chip_good, FILE* file_ptr);
#endif
```

Main code

```
//================================================================
//aNWF.c Main file for aggressive nwell floating
//Author: Justin Gregg
//History:
//Created 2003.05.28
//================================================================

#include "gIWABB.h"
#include <assert.h>

//============= main ============================================

int main(int argc, char* argv[])
{
        if(argc < 7)
        {
                printf("Missing arguments.\nUsage: command start_chip final_chip run_mode bias_res yield# case# (delay power max_step)\n"
                        );
                printf("This program does agressive nwell floating search.\n");
                printf("if yield#=0 then provided delay and power targets are used.\n");
                printf("if max_step is given, it is used as max step\n");
                printf("Valid run_modes:\n");
                printf(" 0\t\t CON\n 1\t\tNWF+DWB\n 2\t\tNWF+NWB\n 3\t\tNWF+PWB\n 4\t\taDWB\n 5\t\tVDD\n 6\t\tNWF+
                        PWB+VDD\n 7\t\tPWB+VDD\n 8\t\tNWF+NWB+VDD\n 9\t\tNWB+VDD\n 11\t\tNWF+VDD\n\n");
                exit(1);
        }

        srandom(time(NULL));

        float target_frequency_list [] = {0.98, 1.00, 1.01, 1.02, 1.03, 1.2};
        float target_power_list [] = {1.00, 1.00, 1.00, 1.00, 1.00, 1.05};

        start_chip = atoi(argv[1]); assert(start_chip >= 0);
        final_chip = atoi(argv[2]); assert(final_chip >= 0);
        run_mode = atoi(argv[3]); assert(run_mode >=0 && run_mode <=11);
        bias_res = atof(argv[4]); assert(bias_res >=0);
        yield_num = atoi(argv[5]);
        case_num = atoi(argv[6]);

        if(yield_num == 0){
                target_frequency = 1/atof(argv[7]);
                target_power = atof(argv[8]);
        }
        else{
                target_frequency = target_frequency_list[yield_num-1];
                target_power = target_power_list[yield_num-1];
        }

        if(argc == 10) max_step = atoi(argv[0]);
        else max_step = 200;

        best_ind = new_ind();

        char temp_filename[256];
        switch (run_mode)
        {
                case 0:
                        sprintf(temp_filename,"cache.CON.y%i.txt", yield_num);
                        cache_file = fopen(temp_filename, "w");
                        if(! cache_file) {printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        printf("Starting CON.\n");
                        CON();
                        break;

                case 1:
                        sprintf(temp_filename,"best.NWF_DWB.y%i.txt", yield_num);
                        best_file = fopen(temp_filename, "w");
                        if(! best_file) {printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        sprintf(temp_filename,"cache.NWF_DWB.y%i.txt", yield_num);
                        cache_file = fopen(temp_filename, "w");
                        if(! cache_file) {printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        sprintf(temp_filename,"eval.NWF_DWB.y%i.txt", yield_num);
                        eval_file = fopen(temp_filename, "w");
                        if(! eval_file) {printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        printf("Starting NWF_DWB.\n");
                        NWF_DWB();
                        break;

                case 2:
                        sprintf(temp_filename,"best.NWF_NWB.y%i.txt", yield_num);
                        best_file = fopen(temp_filename, "w");
```

```
80                  if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"cache.NWF_NWB.y%i.txt", yield_num);
                    cache_file = fopen(temp_filename, "w");
                    if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"eval.NWF_NWB.y%i.txt", yield_num);
85                  eval_file = fopen(temp_filename, "w");
                    if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    printf("Starting NWF_NWB.\n");
                    NWF_NWB();
                    break;
90
            case 3:
                    sprintf(temp_filename,"best.NWF_PWB.y%i.txt", yield_num);
                    best_file = fopen(temp_filename, "w");
                    if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
95                  sprintf(temp_filename,"cache.NWF_PWB.y%i.txt", yield_num);
                    cache_file = fopen(temp_filename, "w");
                    if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"eval.NWF_PWB.y%i.txt", yield_num);
                    eval_file = fopen(temp_filename, "w");
100                 if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    printf("Starting NWF_PWB.\n");
                    NWF_PWB();
                    break;

105         case 4:
                    sprintf(temp_filename,"best.DWB.y%i.txt", yield_num);
                    best_file = fopen(temp_filename, "w");
                    if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"cache.DWB.y%i.txt", yield_num);
110                 cache_file = fopen(temp_filename, "w");
                    if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"eval.DWB.y%i.txt", yield_num);
                    eval_file = fopen(temp_filename, "w");
                    if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
115                 printf("Starting DWB.\n");
                    DWB();
                    break;

            case 5:
120                 sprintf(temp_filename,"best.VDD.y%i.txt", yield_num);
                    best_file = fopen(temp_filename, "w");
                    if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"cache.VDD.y%i.txt", yield_num);
                    cache_file = fopen(temp_filename, "w");
125                 if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"eval.VDD.y%i.txt", yield_num);
                    eval_file = fopen(temp_filename, "w");
                    if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    printf("Starting VDD.\n");
130                 VDD();
                    break;

            case 6:
                    sprintf(temp_filename,"best.NWF_PWB_VDD.y%i.txt", yield_num);
135                 best_file = fopen(temp_filename, "w");
                    if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"cache.NWF_PWB_VDD.y%i.txt", yield_num);
                    cache_file = fopen(temp_filename, "w");
                    if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
140                 sprintf(temp_filename,"eval.NWF_PWB_VDD.y%i.txt", yield_num);
                    eval_file = fopen(temp_filename, "w");
                    if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    printf("Starting NWF_PWB_VDD.\n");
                    NWF_PWB_VDD();
145                 break;

            case 7:
                    sprintf(temp_filename,"best.PWB_VDD.y%i.txt", yield_num);
                    best_file = fopen(temp_filename, "w");
150                 if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"cache.PWB_VDD.y%i.txt", yield_num);
                    cache_file = fopen(temp_filename, "w");
                    if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"eval.PWB_VDD.y%i.txt", yield_num);
155                 eval_file = fopen(temp_filename, "w");
                    if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    printf("Starting PWB_VDD.\n");
                    PWB_VDD();
                    break;
160
            case 8:
                    sprintf(temp_filename,"best.NWF_NWB_VDD.y%i.txt", yield_num);
                    best_file = fopen(temp_filename, "w");
                    if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
165                 sprintf(temp_filename,"cache.NWF_NWB_VDD.y%i.txt", yield_num);
                    cache_file = fopen(temp_filename, "w");
                    if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"eval.NWF_NWB_VDD.y%i.txt", yield_num);
                    eval_file = fopen(temp_filename, "w");
170                 if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    printf("Starting NWF_NWB_VDD.\n");
                    NWF_NWB_VDD();
                    break;

175         case 9:
                    sprintf(temp_filename,"best.NWB_VDD.y%i.txt", yield_num);
                    best_file = fopen(temp_filename, "w");
                    if (! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"cache.NWB_VDD.y%i.txt", yield_num);
180                 cache_file = fopen(temp_filename, "w");
                    if (! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                    sprintf(temp_filename,"eval.NWB_VDD.y%i.txt", yield_num);
                    eval_file = fopen(temp_filename, "w");
                    if (! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
185                 printf("Starting NWB_VDD.\n");
                    NWB_VDD();
```

```
                        break;

/*              case 10:
190                     sprintf ( temp_filename ," best .NWB.y%i.txt",  yield_num );
                        best_file  = fopen( temp_filename , "w");
                        if (!  best_file  ) { printf ("%s  couldn 't  be  opened\n",  temp_filename );  exit (1);}
                        sprintf ( temp_filename ," cache .NWB.y%i.txt",  yield_num );
                        cache_file  = fopen( temp_filename , "w");
195                     if (!  cache_file  ) { printf ("%s  couldn 't  be  opened\n",  temp_filename );  exit (1);}
                        sprintf ( temp_filename ," eval .NWB.y%i.txt",  yield_num );
                        eval_file  = fopen( temp_filename , "w");
                        if (!  eval_file  ) { printf ("%s  couldn 't  be  opened\n",  temp_filename );  exit (1);}
                        printf (" Starting  NWB.\n");
200                     NWB();
                        break ;
*/
                case 10:
                        // sprintf ( temp_filename ," best .NWF.y%i.txt",  yield_num );
205                     //  best_file  = fopen( temp_filename , "w");
                        // if (!  best_file  ) { printf ("%s  couldn 't  be  opened\n",  temp_filename );  exit (1);}
                        // sprintf ( temp_filename ," cache .NWF.y%i.txt",  yield_num );
                        ////  cache_file  = fopen( temp_filename , "w");
                        // if (!  cache_file  ) { printf ("%s  couldn 't  be  opened\n",  temp_filename );  exit (1);}
210                     // sprintf ( temp_filename ," eval .NWF.y%i.txt",  yield_num );
                        //  eval_file  = fopen( temp_filename , "w");
                        // if (!  eval_file  ) { printf ("%s  couldn 't  be  opened\n",  temp_filename );  exit (1);}
                        printf("Starting NWF.\n");
                        NWF();
215                     break;



                case 11:
220                     sprintf(temp_filename,"best.NWF_VDD.y%i.txt", yield_num);
                        best_file = fopen(temp_filename, "w");
                        if(! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        sprintf(temp_filename,"cache.NWF_VDD.y%i.txt", yield_num);
                        cache_file = fopen(temp_filename, "w");
225                     if(! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        sprintf(temp_filename,"eval.NWF_VDD.y%i.txt", yield_num);
                        eval_file = fopen(temp_filename, "w");
                        if(! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        printf("Starting NWF_VDD.\n");
230                     NWF_VDD();
                        break;

                default:
                        printf("Invalid runmode\n\n");
235                     exit(1);
                        break;

        }
}
240

//============================ Evaluation and spice functions ================================
void evaluate(Ind& ind)
{
245     //check  biasing  limits
        if(ind.vdd_bias < vdd_min) ind.vdd_bias = vdd_min;
        if(ind.vdd_bias > vdd_max) ind.vdd_bias = vdd_max;
        if(ind.nwell_bias < nwell_bias_min) ind.nwell_bias = nwell_bias_min;
        if(ind.nwell_bias > nwell_bias_max) ind.nwell_bias = nwell_bias_max;
250     if(ind.pwell_bias < pwell_bias_min) ind.pwell_bias = pwell_bias_min;
        if(ind.pwell_bias > pwell_bias_max) ind.pwell_bias = pwell_bias_max;

        //check  cache  for  ind
        for (int i=0; i< ind_cache_vec.size(); i++)
255     {
                if(is_equal(ind_cache_vec[i], ind) == 1)
                {
                        ind = ind_cache_vec[i];
                        printf("Cache hit\n");
260                     return;
                }
        }

        spice_evals++;
265
        //write  connect  file
        char temp_filename[256];
        char command[2048];

270     sprintf(temp_filename, "%s/case.%s%i/temp.connect", spice_path, case_name, case_num);
        FILE* f = fopen(temp_filename, "w");
        if(!f)
        {
                printf("%s could not be opened.\n", temp_filename);
275             exit(1);
        }

        // define  the  node  names  (numbers)  for  all  the  control  inputs
        int vdiv_node_name[]=
280         {2790, 2823, 2835, 2838, 2841, 2844, 2847, 2850, 2853,
             2760, 2763, 2766, 2769, 2772, 2775, 2778, 2781, 2784, 2787,
             2793, 2796, 2799, 2802, 2805, 2808, 2811, 2814, 2817, 2820,
             2826, 2829, 2832};

285     int pullup_node_name[]=
            {2789, 2822, 2834, 2837, 2840, 2843, 2846, 2849, 2852,
             2759, 2762, 2765, 2768, 2771, 2774, 2777, 2780, 2783, 2786,
             2792, 2795, 2798, 2801, 2804, 2807, 2810, 2813, 2816, 2819,
             2825, 2828, 2831};
290
        int nwell_bias_node_name = 2854;
        int vdiv_vdd_node_name = 2855;
        int vdd_node_name = 1;
```

```
        int substrate_node_name= 2757;
295
        // print  nwell  voltage
        fprintf(f, "40  1\n");
        fprintf(f, "%i 0 %f 0 0\n", nwell_bias_node_name, ind.nwell_bias);
        fprintf(f, "103\n");
300
        // printf  vdd  node
        fprintf(f, "40  1\n");
        fprintf(f, "%i 0 %f 0 0\n", vdd_node_name, ind.vdd_bias);
        fprintf(f, "103\n");
305
        // printf  pullup  controls
        for(int i = chrom_length-1; i >=0; i--)
        {
                fprintf(f, "40  1\n");
310             fprintf(f, "%i 0 %f 0 0\n", pullup_node_name[i], ind.chrom[i] == 1 ? float(0.0) : ind.vdd_bias);
                fprintf(f, "103\n");
        }

        // printf  vdiv  control  signals
315     for (int i = chrom_length-1; i>=0; i--)
        {
                fprintf(f, "40  1\n");
                fprintf(f, "%i 0 %f 0 0\n", vdiv_node_name[i], ind.chrom[i] == 1 ? ind.vdd_bias : float(0.0));
                fprintf(f, "103\n");
320     }

        // print  substrate  voltage
        fprintf(f, "40  1\n");
        fprintf(f, "%i 0 %f 0 0\n", substrate_node_name, ind.pwell_bias);
325     fprintf(f, "103\n");

        // printf  the  vdiv  node  voltage
        fprintf(f, "40  1\n");
        fprintf(f, "%i 0 %f 0 0\n", vdiv_vdd_node_name, ind.vdd_bias);
330     fprintf(f, "103\n");

        fclose(f);

        sprintf(command, "cat %s/%i.cserun.runinput %s/case.%s%i/temp.connect %s/%s > %s/case.%s%i/cserun.runinput",
335                     runinput_path, ind.chip, spice_path, case_name, case_num, spice_path, tail_filename,
                        spice_path, case_name, case_num);
        system(command);

//      sprintf(command, "rm %s", temp_filename);
340 //  system(command);

        //=============run ispice ============
        sprintf(command, run_ispice, case_name, case_num);
        printf("Running ispice: %s\n", command);
345     system(command);

        //=============run hpspice ===========
        char buffer[4096];
        sprintf(command, run_hpspice, spice_path, case_name, case_num, spice_path);
350     printf("Running hpspice: %s\n", command);
        FILE* p;
        float power=0;
        float delay=0;
        p= popen(command, "r");
355     if(!p)
        {
                printf("pipe open failed in evaluate\n");
                exit(1);
        }
360
        fscanf(p, "%s", buffer);
        fscanf(p, "%s", buffer);
        fscanf(p, "%s", buffer);
        fscanf(p, "%s", buffer);
365     fscanf(p, "%s", buffer);
        power = ind.vdd_bias *fabs(atof(buffer));
        fscanf(p, "%s", buffer);
        power += ind.vdd_bias *fabs(atof(buffer));
        ind.power = power/avg_power;
370     fscanf(p, "%s", buffer);
        delay = atof(buffer);
        fscanf(p, "%s", buffer);
        delay += atof(buffer);
        ind.frequency = (1/(delay/2))/avg_freq;
375     pclose(p);

        // generate  the  fitness  of  the  ind
        if(ind.frequency > target_frequency && ind.power > target_power)
                ind.fitness = -1 * pow((target_power - ind.power) / target_power, 2);
380     else if(ind.frequency < target_frequency && power > target_power)
                ind.fitness = -1 * (pow((target_frequency - ind.frequency)/target_frequency, 2) +
                        pow((target_power - ind.power)/target_power, 2));
        else if(ind.frequency < target_frequency && power < target_power)
                ind.fitness = -1 * pow((target_frequency - ind.frequency)/target_frequency, 2);
385     else if(ind.frequency >= target_frequency && ind.power <= target_power)
                ind.fitness = pow((target_frequency - ind.frequency)/target_frequency, 2) +
                        pow((target_power - ind.power)/target_power, 2);

        //compare  to  best_ind
390     if(ind.fitness >= best_ind.fitness) best_ind = ind;

        //add  to  cache_vec
        ind_cache_vec.push_back(ind);

395     // write  to  cache  file
        write_ind(ind, cache_file);

}

400 //============================ General functions for dealing with inds
```

```
        ============================================
    Ind new_ind() // initializes   a new ind
    {
            Ind temp_ind;
            temp_ind.chip = chip_num;
405         temp_ind.pwell_bias = 0.0;
            temp_ind.nwell_bias = 1.1;
            temp_ind.vdd_bias = 1.1;
            for(int i=0; i< chrom_length; i++)
            {
410                 temp_ind.chrom[i]=1;
                    temp_ind.dPower[i]=0.0;
                    temp_ind.dFrequency[i]=0.0;
            }
            temp_ind.fitness = -777e7;
415         temp_ind.frequency = -777e7;
            temp_ind.power = 777e7;
            return temp_ind;
    }

420 int is_equal(Ind ind_a, Ind ind_b) // checks  if  two  inds  are  identical
    {
            if(ind_a.chip != ind_b.chip) return 0;
            if(ind_a.pwell_bias != ind_b.pwell_bias) return 0;
            if(ind_a.nwell_bias != ind_b.nwell_bias) return 0;
425         if(ind_a.vdd_bias != ind_b.vdd_bias) return 0;
            for(int i=0; i < chrom_length; i++)
            {
                    if(ind_a.chrom[i] != ind_b.chrom[i]) return 0;
            }
430         return 1;
    }

    int  best_well_to_float (Ind& ind) // returns  the  index  of  the  connected  well  with  the  smallest  P-F slope when  floated
    {
435         double min_con_PFslope = 777e7;
            int  well_to_float  = -1;
            for(int i=0; i< chrom_length; i++)
            {
                    if(ind.chrom[i]==0 || ind.dFrequency[i] == 0.0) continue; //skip well  if  it  is  floated  already
440                 double con_PFslope = ind.dPower[i]/ind.dFrequency[i];
                    if(con_PFslope < min_con_PFslope)
                    {
                            min_con_PFslope = con_PFslope;
                             well_to_float  = i;
445                 }
            }
            return well_to_float;
    }

450 int best_well_to_connect(Ind& ind) // returns  the  index  of  the  floated  well  with  the  largest  P-F slope when  connected
    {
            double max_float_PFslope = -777e7;
            int well_to_con = -1;
            for(int i=0; i< chrom_length; i++)
455         {
                    if(ind.chrom[i]==1 || ind.dFrequency[i] == 0.0) continue; // skip well  if  it  is  connected  already
                    double float_PFslope = ind.dPower[i]/ind.dFrequency[i];
                    if(float_PFslope > max_float_PFslope)
                    {
460                         max_float_PFslope = float_PFslope;
                            well_to_con = i;
                    }
            }
            return well_to_con;
465 }


    //========================= write file functions ===================================
    void write_ind(Ind ind, FILE* file_ptr)
470 {
            int  float_count= 0;
            for (int  i=0; i< chrom_length; i++) if(ind.chrom[i]==0) float_count++;
             fprintf ( file_ptr , "%i %i %.9f %.9f %.9e %.3f %.3f %.3f ", ind.chip, float_count , ind.frequency, ind.power, ind. fitness ,
                            ind. pwell_bias , ind. nwell_bias , ind.vdd_bias);
475         for(int i=0; i<chrom_length; i++) fprintf(file_ptr , "%i", ind.chrom[i]);
             fprintf ( file_ptr , "\n");
             fflush ( file_ptr );
    }

480 void write_eval(int this_chip_good , FILE* file_ptr)
    {
             fprintf ( file_ptr , "%i\t%i\t%i\t%i\n", chip_num, spice_evals, this_chip_good, good_chips);
             fflush ( file_ptr );
    }
485

    //======================= the aNWF functions ====================================
    void CON()
    {
490         for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
            {
                    Ind ind = new_ind();
                    evaluate(ind);
            }
495 }

    void NWF_DWB() // does the NWF+DWB mode
    {
            good_chips=0;
500         for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
            {
                     spice_evals  = 0;
                    best_ind = new_ind();

505                 //make new ind  and  evaluate  normal  chip
                    Ind ind = new_ind();
```

```
                evaluate(ind);

                if(best_ind. fitness >=0.0)
510             {
                        printf("Normal chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
515                     continue;
                }

                // evaluate each well floated
                for(int i=0; i<chrom_length; i++)
520             {
                        Ind test_ind = new_ind();
                        test_ind .chrom[i]=0;
                        evaluate( test_ind );
                        if(best_ind. fitness >= 0.0)
525                     {
                                printf("Good config for chip %i found during float test.\n", chip_num);
                                break;
                        }
                        ind.dPower[i] = test_ind .power − ind.power;
530                     ind.dFrequency[i] = test_ind .frequency − ind.frequency;
                }

                if(best_ind. fitness >=0.0)
                {
535                     printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
                        continue;
540             }

                if(ind.frequency < target_frequency && ind.power < target_power)
                {
                        float estimated_f = ind.frequency;
545                     float estimated_p = ind.power;
                        while(estimated_f < target_frequency && estimated_p < target_power)
                        {
                                int new_float = best_well_to_float (ind);
                                if(new_float == −1) break;
550                             ind.chrom[new_float]=0;
                                estimated_f += ind.dFrequency[new_float];
                                estimated_p += ind.dPower[new_float];
                        }
                        evaluate(ind);
555                     if(best_ind. fitness >= 0.0) printf("Good config for chip %i found on first float try\n", chip_num);
                }

                if(best_ind. fitness >=0.0)
                {
560                     printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
                        continue;
565             }

                for(int step= 0; step < max_step; step++)
                {
                        int x = random() % 100;
570                     if(ind.frequency < target_frequency)
                        {
                                if(x < 33) ind.pwell_bias += bias_res; //forward bias nfets more
                                else if(x < 66) ind.nwell_bias −= bias_res; //forward bias pfets more
                                else
575                             {
                                        int new_float = best_well_to_float (ind);
                                        if(new_float != −1) ind.chrom[new_float] = 0; // float the next best well
                                }
                        }
580                     else if(ind.power > target_power)
                        {
                                if(x < 33) ind.pwell_bias −= bias_res; // reverse bias nfets more
                                else if(x < 66) ind.nwell_bias += bias_res; //reverse bias pfets more
                                else
585                             {
                                        int new_con = best_well_to_connect(ind);
                                        if(new_con != −1) ind.chrom[new_con] = 1; // connect the best power saving well
                                }
                        }
590
                        evaluate(ind);
                        if(best_ind. fitness >=0.0)
                        {
                                printf("Good configuration found at step %i.\n", step);
595                             break;
                        }
                }

                if(best_ind. fitness >=0.0)
600             {
                        printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
605                     continue;
                }
                else
                {
                        write_ind(best_ind, best_file );
610                     write_eval (0, eval_file );
                }
        }
}
```

```
615

    void NWF_NWB() // does the NWF+NWB mode
    {
            good_chips=0;
620         for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
            {
                    spice_evals = 0;
                    best_ind = new_ind();

625                 //make new ind and evaluate normal chip
                    Ind ind = new_ind();
                    evaluate(ind);

                    if(best_ind. fitness >=0.0)
630                 {
                            good_chips++;
                            printf("Normal chip %i is good.\n", chip_num);
                            write_ind(best_ind, best_file );
                            write_eval (1, eval_file );
635                         continue;
                    }

                    // evaluate each well floated
                    for(int i=0; i<chrom_length; i++)
640                 {
                            Ind test_ind = new_ind();
                            test_ind .chrom[i]=0;
                            evaluate( test_ind );
                            if(best_ind. fitness >= 0.0)
645                         {
                                    printf("Good config for chip %i found during float test.\n", chip_num);
                                    break;
                            }
                            ind.dPower[i] = test_ind.power − ind.power;
650                         ind.dFrequency[i] = test_ind.frequency − ind.frequency;
                    }

                    if(best_ind. fitness >=0.0)
                    {
655                         good_chips++;
                            printf("Chip %i is good.\n", chip_num);
                            write_ind(best_ind, best_file );
                            write_eval (1, eval_file );
                            continue;
660                 }

                    if(ind.frequency < target_frequency && ind.power < target_power)
                    {
                            float estimated_f = ind.frequency;
665                         float estimated_p = ind.power;
                            while(estimated_f < target_frequency && estimated_p < target_power)
                            {
                                    int new_float = best_well_to_float (ind);
                                    if(new_float == −1) break;
670                                 ind.chrom[new_float]=0;
                                    estimated_f += ind.dFrequency[new_float];
                                    estimated_p += ind.dPower[new_float];
                            }
                            evaluate(ind);
675                         if(best_ind. fitness >= 0.0) printf("Good config for chip %i found on first float try\n", chip_num);
                    }

                    if(best_ind. fitness >=0.0)
                    {
680                         good_chips++;
                            printf("Chip %i is good.\n", chip_num);
                            write_ind(best_ind, best_file );
                            write_eval (1, eval_file );
                            continue;
685                 }

                    for(int step= 0; step < max_step; step++)
                    {
                            int x = random() % 100;
690                         if(ind.frequency < target_frequency)
                            {
                                    if(x < 50) ind.nwell_bias −= bias_res; //forward bias pfets more
                                    else
                                    {
695                                         int new_float = best_well_to_float (ind);
                                            if(new_float != −1) ind.chrom[new_float] = 0; // float the next best well
                                    }
                            }
                            else if(ind.power > target_power)
700                         {
                                    if(x < 50) ind.nwell_bias += bias_res; // reverse bias pfets more
                                    else
                                    {
                                            int new_con = best_well_to_connect(ind);
705                                         if(new_con != −1) ind.chrom[new_con] = 1; // connect the best power saving well
                                    }
                            }

                            evaluate(ind);
710                         if(best_ind. fitness >=0.0)
                            {
                                    printf("Good configuration found at step %i.\n", step);
                                    break;
                            }
715                 }

                    if(best_ind. fitness >=0.0)
                    {
                            good_chips++;
720                         printf("Chip %i is good.\n", chip_num);
```

```
                        write_ind(best_ind, best_file);
                        write_eval(1, eval_file);
                        continue;
                }
725             else
                {
                        write_ind(best_ind, best_file);
                        write_eval(0, eval_file);
                }
730     }
}



735 void NWF_PWB() // does the NWF+PWB mode
{
        good_chips=0;
        for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
        {
740             spice_evals = 0;
                best_ind = new_ind();

                //make new ind and evaluate normal chip
                Ind ind = new_ind();
745             evaluate(ind);

                if(best_ind.fitness >=0.0)
                {
                        good_chips++;
750                     printf("Normal chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file);
                        write_eval(1, eval_file);
                        continue;
                }
755
                // evaluate each well floated
                for(int i=0; i<chrom_length; i++)
                {
                        Ind test_ind = new_ind();
760                     test_ind.chrom[i]=0;
                        evaluate(test_ind);
                        if(best_ind.fitness >= 0.0)
                        {
                                printf("Good config for chip %i found during float test.\n", chip_num);
765                             break;
                        }
                        ind.dPower[i] = test_ind.power - ind.power;
                        ind.dFrequency[i] = test_ind.frequency - ind.frequency;
                }
770
                if(best_ind.fitness >=0.0)
                {
                        good_chips++;
                        printf("Chip %i is good.\n", chip_num);
775                     write_ind(best_ind, best_file);
                        write_eval(1, eval_file);
                        continue;
                }

780             if(ind.frequency < target_frequency && ind.power < target_power)
                {
                        float estimated_f = ind.frequency;
                        float estimated_p = ind.power;
                        while(estimated_f < target_frequency && estimated_p < target_power)
785                     {
                                int new_float = best_well_to_float(ind);
                                if(new_float == -1) break;
                                ind.chrom[new_float]=0;
                                estimated_f += ind.dFrequency[new_float];
790                             estimated_p += ind.dPower[new_float];
                        }
                        evaluate(ind);
                        if(best_ind.fitness >= 0.0) printf("Good config for chip %i found on first float try\n", chip_num);
                }
795
                if(best_ind.fitness >=0.0)
                {
                        good_chips++;
                        printf("Chip %i is good.\n", chip_num);
800                     write_ind(best_ind, best_file);
                        write_eval(1, eval_file);
                        continue;
                }

805             for(int step= 0; step < max_step; step++)
                {
                        int x = random() % 100;
                        if(ind.frequency < target_frequency)
                        {
810                             if(x < 50) ind.pwell_bias += bias_res; //forward bias nfets more
                                else
                                {
                                        int new_float = best_well_to_float(ind);
                                        if(new_float != -1) ind.chrom[new_float] = 0; // float the next best well
815                             }
                        }
                        else if(ind.power > target_power)
                        {
                                if(x < 50) ind.pwell_bias -= bias_res; // reverse bias nfets more
820                             else
                                {
                                        int new_con = best_well_to_connect(ind);
                                        if(new_con != -1) ind.chrom[new_con] = 1; // connect the best power saving well
                                }
825                     }

                        evaluate(ind);
```

```
                        if(best_ind. fitness >=0.0)
                        {
830                             printf("Good configuration found at step %i.\n", step);
                                break;
                        }
                }

835             if(best_ind. fitness >=0.0)
                {
                        good_chips++;
                        printf("Chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file );
840                     write_eval (1, eval_file );
                        continue;
                }
                else
                {
845                     write_ind(best_ind, best_file );
                        write_eval (0, eval_file );
                }
        }
}
850

void DWB() // does the DWB mode
{
        good_chips=0;
855     for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
        {
                spice_evals = 0;
                best_ind = new_ind();

860             //make new ind and evaluate normal chip
                Ind ind = new_ind();
                evaluate(ind);

                if(best_ind. fitness >=0.0)
865             {
                        good_chips++;
                        printf("Normal chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
870                     continue;
                }

                for(int step= 0; step < max_step; step++)
                {
875                     int x = random() % 100;
                        if(ind.frequency < target_frequency)
                        {
                                if(x < 50) ind.pwell_bias += bias_res; //forward bias nfets more
                                else ind. nwell_bias -= bias_res; //forward bias pfets more
880                     }
                        else if(ind.power > target_power)
                        {
                                if(x < 50) ind.pwell_bias -= bias_res; // reverse bias nfets more
                                else ind. nwell_bias += bias_res; // reverse bias pfets more
885                     }

                        evaluate(ind);
                        if(best_ind. fitness >=0.0)
                        {
890                             printf("Good configuration found at step %i.\n", step);
                                break;
                        }
                }

895             if(best_ind. fitness >=0.0)
                {
                        good_chips++;
                        printf("Chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file );
900                     write_eval (1, eval_file );
                        continue;
                }
                else
                {
905                     write_ind(best_ind, best_file );
                        write_eval (0, eval_file );
                }
        }
}
910

void VDD() // does the VDD mode
{
        good_chips=0;
915     for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
        {
                spice_evals = 0;
                best_ind = new_ind();

920             //make new ind and evaluate normal chip
                Ind ind = new_ind();
                evaluate(ind);

                if(best_ind. fitness >=0.0)
925             {
                        good_chips++;
                        printf("Normal chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
930                     continue;
                }

                for(int step= 0; step < max_step; step++)
                {
```

```
935                     int x = random() % 100;
                        if(ind.frequency < target_frequency)
                        {
                                ind.vdd_bias += bias_res;
                        }
940                     else if(ind.power > target_power)
                        {
                                ind.vdd_bias -= bias_res;
                        }

945                     evaluate(ind);
                        if(best_ind. fitness >=0.0)
                        {
                                printf("Good configuration found at step %i.\n", step);
                                break;
950                     }
                }

                if(best_ind. fitness >=0.0)
                {
955                     good_chips++;
                        printf("Chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
                        continue;
960             }
                else
                {
                        write_ind(best_ind, best_file );
                        write_eval (0, eval_file );
965             }
        }
}


970 void NWF() // does the NWF mode
{
        good_chips=0;
        for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
        {
975                     char temp_filename[256];

                        sprintf (temp_filename,"%i.best.NWF.y%i.txt", chip_num, yield_num);
                        best_file = fopen(temp_filename, "w");
                        if(! best_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
980                     sprintf (temp_filename,"%i.cache.NWF.y%i.txt", chip_num, yield_num);
                        cache_file = fopen(temp_filename, "w");
                        if(! cache_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}
                        sprintf (temp_filename,"%i.eval.NWF.y%i.txt", chip_num, yield_num);
                        eval_file = fopen(temp_filename, "w");
985                     if(! eval_file ) { printf("%s couldn't be opened\n", temp_filename); exit(1);}

                spice_evals = 0;
                best_ind = new_ind();

990             //make new ind and evaluate normal chip
                Ind ind = new_ind();
                evaluate(ind);

                if(best_ind. fitness >=0.0)
995             {
                        good_chips++;
                        printf("Normal chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
1000                    continue;
                }

                // evaluate each well floated
                for(int i=0; i<chrom_length; i++)
1005            {
                        Ind test_ind = new_ind();
                        test_ind .chrom[i]=0;
                        evaluate( test_ind );
                        if(best_ind. fitness >= 0.0)
1010                    {
                                printf("Good config for chip %i found during float test.\n", chip_num);
                                break;
                        }
                        ind.dPower[i] = test_ind.power - ind.power;
1015                    ind.dFrequency[i] = test_ind.frequency - ind.frequency;
                }

                if(best_ind. fitness >=0.0)
                {
1020                    good_chips++;
                        printf("Chip %i is good.\n", chip_num);
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
                        continue;
1025            }

                if(ind.frequency < target_frequency && ind.power < target_power)
                {
                        float estimated_f = ind.frequency;
1030                    float estimated_p = ind.power;
                        while(estimated_f < target_frequency && estimated_p < target_power)
                        {
                                int new_float = best_well_to_float (ind);
                                if(new_float == -1) break;
1035                            ind.chrom[new_float]=0;
                                estimated_f += ind.dFrequency[new_float];
                                estimated_p += ind.dPower[new_float];
                        }
                        evaluate(ind);
1040                    if(best_ind. fitness >= 0.0) printf("Good config for chip %i found on first float try\n", chip_num);
                }
```

```
                    if(best_ind. fitness >=0.0)
                    {
1045                        good_chips++;
                            printf("Chip %i is good.\n", chip_num);
                            write_ind(best_ind, best_file );
                            write_eval (1, eval_file );
                            continue;
1050                }

                    for(int step= 0; step < max_step; step++)
                    {
                            int x = random() % 100;
1055                        if(ind.frequency < target_frequency)
                            {
                                    int new_float = best_well_to_float (ind);
                                    if(new_float != -1) ind.chrom[new_float] = 0; // float the next best well
                            }
1060                        else if(ind.power > target_power)
                            {
                                    int new_con = best_well_to_connect(ind);
                                    if(new_con != -1) ind.chrom[new_con] = 1; // connect the best power saving well
                            }
1065
                            evaluate(ind);
                            if(best_ind. fitness >=0.0)
                            {
                                    printf("Good configuration found at step %i.\n", step);
1070                                break;
                            }
                    }

                    if(best_ind. fitness >=0.0)
1075                {
                            good_chips++;
                            printf("Chip %i is good.\n", chip_num);
                            write_ind(best_ind, best_file );
                            write_eval (1, eval_file );
1080                        continue;
                    }
                    else
                    {
                            write_ind(best_ind, best_file );
1085                        write_eval (0, eval_file );
                    }
            }
    }

1090
    void NWF_VDD() // does the NWF+VDD mode
    {
            good_chips=0;
            for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
1095        {
                    spice_evals = 0;
                    best_ind = new_ind();

                    //make new ind and evaluate normal chip
1100                Ind ind = new_ind();
                    evaluate(ind);

                    if(best_ind. fitness >=0.0)
                    {
1105                        good_chips++;
                            printf("Normal chip %i is good.\n", chip_num);
                            write_ind(best_ind, best_file );
                            write_eval (1, eval_file );
                            continue;
1110                }

                    // evaluate each well floated
                    for(int i=0; i<chrom_length; i++)
                    {
1115                        Ind test_ind = new_ind();
                            test_ind .chrom[i]=0;
                            evaluate( test_ind );
                            if(best_ind. fitness >= 0.0)
                            {
1120                                printf("Good config for chip %i found during float test.\n", chip_num);
                                    break;
                            }
                            ind.dPower[i] = test_ind.power - ind.power;
                            ind.dFrequency[i] = test_ind.frequency - ind.frequency;
1125                }

                    if(best_ind. fitness >=0.0)
                    {
                            good_chips++;
1130                        printf("Chip %i is good.\n", chip_num);
                            write_ind(best_ind, best_file );
                            write_eval (1, eval_file );
                            continue;
                    }
1135
                    if(ind.frequency < target_frequency && ind.power < target_power)
                    {
                            float estimated_f = ind.frequency;
                            float estimated_p = ind.power;
1140                        while(estimated_f < target_frequency && estimated_p < target_power)
                            {
                                    int new_float = best_well_to_float (ind);
                                    if(new_float == -1) break;
                                    ind.chrom[new_float]=0;
1145                                estimated_f += ind.dFrequency[new_float];
                                    estimated_p += ind.dPower[new_float];
                            }
                            evaluate(ind);
```

```
                if(best_ind.fitness >= 0.0) printf("Good config for chip %i found on first float try\n", chip_num);
1150        }

            if(best_ind.fitness >=0.0)
            {
                good_chips++;
1155            printf("Chip %i is good.\n", chip_num);
                write_ind(best_ind, best_file);
                write_eval(1, eval_file);
                continue;
            }
1160
            for(int step= 0; step < max_step; step++)
            {
                int x = random() % 100;
                if(ind.frequency < target_frequency)
1165            {
                    if(x < 50) ind.vdd_bias += bias_res;
                    else
                    {
                        int new_float = best_well_to_float(ind);
1170                    if(new_float != -1) ind.chrom[new_float] = 0; // float the next best well
                    }
                }
                else if(ind.power > target_power)
                {
1175                if(x < 50) ind.vdd_bias -= bias_res;
                    else
                    {
                        int new_con = best_well_to_connect(ind);
                        if(new_con != -1) ind.chrom[new_con] = 1; // connect the best power saving well
1180                }
                }

                evaluate(ind);
                if(best_ind.fitness >=0.0)
1185            {
                    printf("Good configuration found at step %i.\n", step);
                    break;
                }
            }
1190
            if(best_ind.fitness >=0.0)
            {
                good_chips++;
                printf("Chip %i is good.\n", chip_num);
1195            write_ind(best_ind, best_file);
                write_eval(1, eval_file);
                continue;
            }
            else
1200        {
                write_ind(best_ind, best_file);
                write_eval(0, eval_file);
            }
        }
1205 }

void NWF_PWB_VDD() // does the NWF+PWB+VDD mode
{
        good_chips=0;
1210    for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
        {
            spice_evals = 0;
            best_ind = new_ind();

1215        //make new ind and evaluate normal chip
            Ind ind = new_ind();
            evaluate(ind);

            if(best_ind.fitness >=0.0)
1220        {
                printf("Normal chip %i is good.\n", chip_num);
                good_chips++;
                write_ind(best_ind, best_file);
                write_eval(1, eval_file);
1225            continue;
            }

            // evaluate each well floated
            for(int i=0; i<chrom_length; i++)
1230        {
                Ind test_ind = new_ind();
                test_ind.chrom[i]=0;
                evaluate(test_ind);
                if(best_ind.fitness >= 0.0)
1235            {
                    printf("Good config for chip %i found during float test.\n", chip_num);
                    break;
                }
                ind.dPower[i] = test_ind.power - ind.power;
1240            ind.dFrequency[i] = test_ind.frequency - ind.frequency;
            }

            if(best_ind.fitness >=0.0)
            {
1245            printf("Chip %i is good.\n", chip_num);
                good_chips++;
                write_ind(best_ind, best_file);
                write_eval(1, eval_file);
                continue;
1250        }

            if(ind.frequency < target_frequency && ind.power < target_power)
            {
                float estimated_f = ind.frequency;
1255            float estimated_p = ind.power;
```

```
                        while(estimated_f < target_frequency && estimated_p < target_power)
                        {
                                int new_float = best_well_to_float (ind);
                                if(new_float == -1) break;
1260                            ind.chrom[new_float]=0;
                                estimated_f += ind.dFrequency[new_float];
                                estimated_p += ind.dPower[new_float];
                        }
                        evaluate(ind);
1265                    if(best_ind. fitness >= 0.0) printf("Good config for chip %i found on first float try\n", chip_num);
                }

                if(best_ind. fitness >=0.0)
                {
1270                    printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
                        continue;
1275            }

                for(int step= 0; step < max_step; step++)
                {
                        int x = random() % 100;
1280                    if(ind.frequency < target_frequency)
                        {
                                if(x < 33) ind.pwell_bias += bias_res; //forward bias nfets more
                                else if(x < 66) ind.vdd_bias += bias_res; //forward bias pfets more
                                else
1285                            {
                                        int new_float = best_well_to_float (ind);
                                        if(new_float != -1) ind.chrom[new_float] = 0; // float the next best well
                                }
                        }
1290                    else if(ind.power > target_power)
                        {
                                if(x < 33) ind.pwell_bias -= bias_res; // reverse bias nfets more
                                else if(x < 66) ind.vdd_bias -= bias_res; //reverse bias pfets more
                                else
1295                            {
                                        int new_con = best_well_to_connect(ind);
                                        if(new_con != -1) ind.chrom[new_con] = 1; // connect the best power saving well
                                }
                        }
1300
                        evaluate(ind);
                        if(best_ind. fitness >=0.0)
                        {
                                printf("Good configuration found at step %i.\n", step);
1305                            break;
                        }
                }

                if(best_ind. fitness >=0.0)
1310            {
                        printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
1315                    continue;
                }
                else
                {
                        write_ind(best_ind, best_file );
1320                    write_eval (0, eval_file );
                }
        }
}

1325
void PWB_VDD() // does the PWB+VDD mode
{
        good_chips=0;
        for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
1330    {
                spice_evals = 0;
                best_ind = new_ind();

                //make new ind and evaluate normal chip
1335            Ind ind = new_ind();
                evaluate(ind);

                if(best_ind. fitness >=0.0)
                {
1340                    printf("Normal chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file );
                        write_eval (1, eval_file );
                        continue;
1345            }

                for(int step= 0; step < max_step; step++)
                {
                        int x = random() % 100;
1350                    if(ind.frequency < target_frequency)
                        {
                                if(x < 50) ind.pwell_bias += bias_res; //forward bias nfets more
                                else ind.vdd_bias += bias_res; //forward bias pfets more
                        }
1355                    else if(ind.power > target_power)
                        {
                                if(x < 50) ind.pwell_bias -= bias_res; // reverse bias nfets more
                                else ind.vdd_bias -= bias_res; // reverse bias pfets more
                        }
1360
                        evaluate(ind);
                        if(best_ind. fitness >=0.0)
```

```
                        {
                                printf("Good configuration found at step %i.\n", step);
1365                            break;
                        }
                }

                if(best_ind.fitness >=0.0)
1370            {
                        printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file);
                        write_eval(1, eval_file);
1375                    continue;
                }
                else
                {
                        write_ind(best_ind, best_file);
1380                    write_eval(0, eval_file);
                }
        }
}

1385



void NWF_NWB_VDD() // does the NWF+NWB+VDD mode
{
1390    good_chips=0;
        for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
        {
                spice_evals = 0;
                best_ind = new_ind();
1395
                //make new ind and evaluate normal chip
                Ind ind = new_ind();
                evaluate(ind);

1400            if(best_ind.fitness >=0.0)
                {
                        printf("Normal chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file);
1405                    write_eval(1, eval_file);
                        continue;
                }

                // evaluate each well floated
1410            for(int i=0; i<chrom_length; i++)
                {
                        Ind test_ind = new_ind();
                        test_ind.chrom[i]=0;
                        evaluate(test_ind);
1415                    if(best_ind.fitness >= 0.0)
                        {
                                printf("Good config for chip %i found during float test.\n", chip_num);
                                break;
                        }
1420                    ind.dPower[i] = test_ind.power - ind.power;
                        ind.dFrequency[i] = test_ind.frequency - ind.frequency;
                }

                if(best_ind.fitness >=0.0)
1425            {
                        printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file);
                        write_eval(1, eval_file);
1430                    continue;
                }

                if(ind.frequency < target_frequency && ind.power < target_power)
                {
1435                    float estimated_f = ind.frequency;
                        float estimated_p = ind.power;
                        while(estimated_f < target_frequency && estimated_p < target_power)
                        {
                                int new_float = best_well_to_float(ind);
1440                            if(new_float == -1) break;
                                ind.chrom[new_float]=0;
                                estimated_f += ind.dFrequency[new_float];
                                estimated_p += ind.dPower[new_float];
                        }
1445                    evaluate(ind);
                        if(best_ind.fitness >= 0.0) printf("Good config for chip %i found on first float try\n", chip_num);
                }

                if(best_ind.fitness >=0.0)
1450            {
                        printf("Chip %i is good.\n", chip_num);
                        good_chips++;
                        write_ind(best_ind, best_file);
                        write_eval(1, eval_file);
1455                    continue;
                }

                for(int step= 0; step < max_step; step++)
                {
1460                    int x = random() % 100;
                        if(ind.frequency < target_frequency)
                        {
                                if(x < 33) ind.vdd_bias += bias_res;
                                else if(x < 66) ind.nwell_bias -= bias_res; //forward bias pfets more
1465                            else
                                {
                                        int new_float = best_well_to_float(ind);
                                        if(new_float != -1) ind.chrom[new_float] = 0; // float the next best well
                                }
```

```
1470                        }
                            else if(ind.power > target_power)
                            {
                                    if(x < 33) ind.vdd_bias -= bias_res;
                                    else if(x < 66) ind.nwell_bias += bias_res; // reverse bias pfets more
1475                                else
                                    {
                                            int new_con = best_well_to_connect(ind);
                                            if(new_con != -1) ind.chrom[new_con] = 1; // connect the best power saving well
                                    }
1480                        }

                            evaluate(ind);
                            if(best_ind.fitness >=0.0)
                            {
1485                                printf("Good configuration found at step %i.\n", step);
                                    break;
                            }
                    }

1490                if(best_ind.fitness >=0.0)
                    {
                            printf("Chip %i is good.\n", chip_num);
                            good_chips++;
                            write_ind(best_ind, best_file);
1495                        write_eval(1, eval_file);
                            continue;
                    }
                    else
                    {
1500                        write_ind(best_ind, best_file);
                            write_eval(0, eval_file);
                    }
            }
    }
1505



    void NWB_VDD() // does the NWB+VDD mode
1510 {
            good_chips=0;
            for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
            {
                    spice_evals = 0;
1515                best_ind = new_ind();

                    //make new ind and evaluate normal chip
                    Ind ind = new_ind();
                    evaluate(ind);
1520
                    if(best_ind.fitness >=0.0)
                    {
                            printf("Normal chip %i is good.\n", chip_num);
                            good_chips++;
1525                        write_ind(best_ind, best_file);
                            write_eval(1, eval_file);
                            continue;
                    }

1530                for(int step= 0; step < max_step; step++)
                    {
                            int x = random() % 100;
                            if(ind.frequency < target_frequency)
                            {
1535                                if(x < 50) ind.vdd_bias += bias_res;
                                    else ind.nwell_bias -= bias_res; //forward bias pfets more
                            }
                            else if(ind.power > target_power)
                            {
1540                                if(x < 50) ind.vdd_bias -= bias_res;
                                    else ind.nwell_bias += bias_res; // reverse bias pfets more
                            }

                            evaluate(ind);
1545                        if(best_ind.fitness >=0.0)
                            {
                                    printf("Good configuration found at step %i.\n", step);
                                    break;
                            }
1550                }

                    if(best_ind.fitness >=0.0)
                    {
                            printf("Chip %i is good.\n", chip_num);
1555                        good_chips++;
                            write_ind(best_ind, best_file);
                            write_eval(1, eval_file);
                            continue;
                    }
1560                else
                    {
                            write_ind(best_ind, best_file);
                            write_eval(0, eval_file);
                    }
1565        }
    }

    void NWB() // does the NWB mode
    {
1570        good_chips=0;
            for(chip_num=start_chip; chip_num <= final_chip; chip_num++)
            {
                    spice_evals = 0;
                    best_ind = new_ind();
1575
                    //make new ind and evaluate normal chip
```

```
                    Ind ind = new_ind();
                    evaluate(ind);

1580                if(best_ind. fitness >=0.0)
                    {
                            printf("Normal chip %i is good.\n", chip_num);
                            good_chips++;
                            write_ind(best_ind, best_file);
1585                        write_eval (1, eval_file);
                            continue;
                    }

                    for(int step= 0; step < max_step; step++)
1590                {
                            int x = random() % 100;
                            if(ind.frequency < target_frequency)
                            {
                                    ind. nwell_bias -= bias_res; //forward bias pfets more
1595                        }
                            else if(ind.power > target_power)
                            {
                                    ind. nwell_bias += bias_res; // reverse bias pfets more
                            }
1600
                            evaluate(ind);
                            if(best_ind. fitness >=0.0)
                            {
                                    printf("Good configuration found at step %i.\n", step);
1605                                break;
                            }
                    }

                    if(best_ind. fitness >=0.0)
1610                {
                            printf("Chip %i is good.\n", chip_num);
                            good_chips++;
                            write_ind(best_ind, best_file);
                            write_eval (1, eval_file);
1615                        continue;
                    }
                    else
                    {
                            write_ind(best_ind, best_file);
1620                        write_eval (0, eval_file);
                    }
            }
    }
```

Appendix B

moIWABB source code

Header files

```
/*========================================================================
  PISA  (www.tik.ee.ethz.ch/pisa/)
  ========================================================================
  Computer Engineering (TIK)
5 ETH Zurich
  ========================================================================
  SPEA2 - Strength Pareto EA 2

   Implementation in C for the selector side.
10
  Header file.

   file: spea2.h
   author: Marco Laumanns, laumanns@tik.ee.ethz.ch
15 last change: 31.10.02

     Modified by JLG 14.07.03
  ========================================================================
*/
20
#ifndef SPEA2_H
#define SPEA2_H

/*-----------------------| specify Operating System |------------------*/
25 /* necessary for wait() */

/* #define PISA_WIN */
#define PISA_UNIX

30 /*----------------------------| macro |-----------------------------------*/

#define PISA_ERROR(x) fprintf(stderr, "\nError: " x "\n"), fflush(stderr), exit(EXIT_FAILURE)

/*---------------------------| constants |--------------------------------*/
35 #define FILE_NAME_LENGTH 128 /* maximal length of filenames */
#define CFG_ENTRY_LENGTH 128 /* maximal length of entries in cfg file */
#define PISA_MAXDOUBLE 1E99 /* Internal maximal value for double */

#define chrom_length 32 /*Max chromosome length*/
40 #define vdd_min 0.6
#define vdd_max 1.6
#define nwell_bias_min 0.6
#define nwell_bias_max 1.6
#define pwell_bias_min -0.5
45 #define pwell_bias_max 0.5

/*----------------------------| structs |--------------------------------*/

typedef struct ind_st /* an individual */
50 {
     int index;
     float pwell_bias, nwell_bias, vdd;
     int genes[chrom_length];
     double *f; /* objective vector */
55   double fitness;
} ind;

typedef struct pop_st /* a population */
{
60   int size;
     int maxsize;
     ind **ind;
} pop;

65 /* population containers */
pop *pp_all;
pop *pp_new;
pop *pp_sel;
//pop *pp_cache;
70
/* common parameters */
int popsize; /* number of individuals in initial population */
int parentpop_size;     /* number of individuals selected as parents */
int childpop_size; /* number of offspring individuals */
75 int dim;    /* number of objectives */
```

```
float bias_res;
int starting_chip, final_chip, chip_num, case_num;
int max_generations, gen, bias_gen, runmode, overall_runmode;

80 /* local parameters from paramfile */
int seed; /* seed for random number generator */
int tournament; /* parameter for tournament selection */


85 /*-------------| functions for control flow (in spea2.c) |------------*/

void write_flag(char *filename, int flag);
int read_flag(char *filename);
//void wait(double sec);
90
/*---------| initialization function (in spea2_functions.c) |---------*/

void initialize();

95 /*--------| memory allocation functions (in spea2_functions.c) |------*/

void* chk_malloc(size_t size);
pop* create_pop(int size, int dim);
ind* create_ind(int dim);
100
void free_memory(void);
void free_pop(pop *pp);
void free_ind(ind *p_ind);

105 /*-----| functions implementing the selection (spea2_functions.c) |---*/
void bias_search1();
void bias_search2();
void copy_arc();
void recover_arc();
110
void selection();
void mergeOffspring();
void calcFitnesses();
void calcDistances();
115 int getNN(int index, int k);
double getNNd(int index, int k);
void environmentalSelection();
void truncate_nondominated();
void truncate_dominated();
120 void matingSelection();

void select_initial();
void select_normal();
int dominates(ind *p_ind_a, ind *p_ind_b);
125 int is_equal(ind *p_ind_a, ind *p_ind_b);
double calcDistance(ind *p_ind_a, ind *p_ind_b);
int irand(int range);
FILE* cache_file;
/*--------------------| data exchange functions |------------------------*/
130
/* in spea2_functions.c */

void crossover();
void mutate();
135 void init_first_pop(void);
void eval_new(void);
void write_sel(void);
void write_arc(void);

140 /* in spea2_io.c */

int eval_pop(pop *pp, int size, int dim);
void write_pop(char *filename, pop *pp, int size);

145 #endif /* SPEA2_H */
```

Main code

```
/*========================================================================
  PISA (www.tik.ee.ethz.ch/pisa/)
  ========================================================================
  Computer Engineering (TIK)
5 ETH Zurich
  ========================================================================
  SPEA2 - Strength Pareto EA 2

  Implements most functions.
10
  file: spea2_functions.c
  author: Marco Laumanns, laumanns@tik.ee.ethz.ch
  last change: 31.10.02

15   Modified by JLG 14.07.03
  ========================================================================
*/


20 #include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <math.h>
25 #include <time.h>

#include "spea2i.h"

/* other variables */
30 char cfgfile[FILE_NAME_LENGTH]; /* 'cfg' file (common parameters) */
char inifile[FILE_NAME_LENGTH]; /* 'ini' file (initial population) */
char selfile[FILE_NAME_LENGTH]; /* 'sel' file (parents) */
```

```
    char arcfile [FILE_NAME_LENGTH]; /* 'arc' file (archive) */
    char varfile [FILE_NAME_LENGTH]; /* 'var' file ( offspring ) */
35

    /* population containers */
    //pop * pp_all ;
    //pop *pp_new;
40  //pop * pp_sel ;
    pop *pp_all_copy;

    /* SPEA2 internal global variables */
    int * fitness_bucket ;
45  int *fitness_bucket_mod;
    int *copies;
    int *old_index;
    double **dist;
    int **NN;
50
    void write_new();

    /*-----------------------| initialization |------------------------------*/

55  void initialize ()
    /* Performs the necessary initialization to start in state 0. */
    {

        srand(time(NULL)); /* seeding random number generator */
60
        /* create individual and archive pop */
        pp_all = create_pop(popsize + childpop_size*2, dim);
        pp_sel = create_pop(parentpop_size, dim);
    //      pp_cache = create_pop (10000, dim);
65  }


    /*-------------------| memory allocation functions |---------------------*/

70  void* chk_malloc(size_t size)
    /* Wrapper function for malloc () . Checks for failed allocations . */
    {
        void *return_value = malloc(size);
        if(return_value == NULL)
75          PISA_ERROR("Selector: Out of memory.");
        return (return_value);
    }


80  pop* create_pop(int maxsize, int dim)
    /* Allocates memory for a population . */
    {
        int i;
        pop *pp;
85
        assert(dim >= 0);
        assert(maxsize >= 0);

        pp = (pop*) chk_malloc(sizeof(pop));
90      pp->size = 0;
        pp->maxsize = maxsize;
        pp->ind = (ind**) chk_malloc(maxsize * sizeof(ind*));

        for (i = 0; i < maxsize; i++)
95          pp->ind[i] = NULL;

        return (pp);
    }

100
    ind* create_ind(int dim)
    /* Allocates memory for one individual . */
    {
        ind *p_ind;
105     int i;

        assert(dim >= 0);

        p_ind = (ind*) chk_malloc(sizeof(ind));
110
        p_ind->index = -1;
        p_ind->fitness = -1;

        switch(runmode)
115     {
            case 1: //NWF+DWB
                        for(i=0; i < chrom_length; i++)
                        {
                                if(rand()%33 <= 4) p_ind->genes[i]=0;
120                             else p_ind->genes[i]=1;
                        }
                        p_ind->vdd=1.1;
                        p_ind->nwell_bias = nwell_bias_min + rand() % ((int)((nwell_bias_max - nwell_bias_min)/bias_res)+1) * bias_res;
                        p_ind->pwell_bias = pwell_bias_min + rand() % ((int)((pwell_bias_max - pwell_bias_min)/bias_res)+1) * bias_res;
125         break;
            case 2: //NWF+NWB
                        for(i=0; i < chrom_length; i++)
                        {
                                if(rand()%33 <= 4) p_ind->genes[i]=0;
130                             else p_ind->genes[i]=1;
                        }
                        p_ind->vdd=1.1;
                        p_ind->nwell_bias = nwell_bias_min + rand() % ((int)((nwell_bias_max - nwell_bias_min)/bias_res)+1) * bias_res;
                        p_ind->pwell_bias = 0.0;
135         break;
            case 3: //NWF+PWB
                        for(i=0; i < chrom_length; i++)
                        {
                                if(rand()%33 <= 4) p_ind->genes[i]=0;
```

```
140                         else p_ind->genes[i]=1;
                    }
                    p_ind->vdd=1.1;
                    p_ind->nwell_bias = 1.1;
                    p_ind->pwell_bias = pwell_bias_min + rand() % ((int)((pwell_bias_max - pwell_bias_min)/bias_res)+1) * bias_res;
145         break;
            case 4: //DWB
                    for(i=0; i < chrom_length; i++)
                    {
                            p_ind->genes[i]=1;
150                 }
                    p_ind->vdd=1.1;
                    p_ind->nwell_bias = nwell_bias_min + rand() % ((int)((nwell_bias_max - nwell_bias_min)/bias_res)+1) * bias_res;
                    p_ind->pwell_bias = pwell_bias_min + rand() % ((int)((pwell_bias_max - pwell_bias_min)/bias_res)+1) * bias_res;
            break;
155         case 5: //VDD
                    for(i=0; i < chrom_length; i++)
                    {
                            p_ind->genes[i]=1;
                    }
160                 p_ind->vdd= vdd_min + rand() % ((int)((vdd_max - vdd_min)/bias_res)+1) * bias_res;
                    p_ind->nwell_bias = p_ind->vdd;
                    p_ind->pwell_bias = 0.0;
            break;
            case 6: //NWF+PWB+VDD
165                 for(i=0; i < chrom_length; i++)
                    {
                            if(rand()%33 <= 4) p_ind->genes[i]=0;
                            else p_ind->genes[i]=1;
                    }
170                 p_ind->vdd= vdd_min + rand() % ((int)((vdd_max - vdd_min)/bias_res)+1) * bias_res;
                    p_ind->nwell_bias = p_ind->vdd;
                    p_ind->pwell_bias = pwell_bias_min + rand() % ((int)((pwell_bias_max - pwell_bias_min)/bias_res)+1) * bias_res;
            break;
            case 7: //PWB+VDD
175                 for(i=0; i < chrom_length; i++)
                    {
                            p_ind->genes[i]=1;
                    }
                    p_ind->vdd= vdd_min + rand() % ((int)((vdd_max - vdd_min)/bias_res)+1) * bias_res;
180                 p_ind->nwell_bias = p_ind->vdd;
                    p_ind->pwell_bias = pwell_bias_min + rand() % ((int)((pwell_bias_max - pwell_bias_min)/bias_res)+1) * bias_res;
            break;
            case 8: //NWF+NWB+VDD
                    for(i=0; i < chrom_length; i++)
185                 {
                            if(rand()%33 <= 4) p_ind->genes[i]=0;
                            else p_ind->genes[i]=1;
                    }
                    p_ind->vdd= vdd_min + rand() % ((int)((vdd_max - vdd_min)/bias_res)+1) * bias_res;
190                 p_ind->nwell_bias = nwell_bias_min + rand() % ((int)((nwell_bias_max - nwell_bias_min)/bias_res)+1) * bias_res;
                    p_ind->pwell_bias = 0.0;
            break;
            case 9: //NWB+VDD
                    for(i=0; i < chrom_length; i++)
195                 {
                            p_ind->genes[i]=1;
                    }
                    p_ind->vdd= vdd_min + rand() % ((int)((vdd_max - vdd_min)/bias_res)+1) * bias_res;
                    p_ind->nwell_bias = nwell_bias_min + rand() % ((int)((nwell_bias_max - nwell_bias_min)/bias_res)+1) * bias_res;
200                 p_ind->pwell_bias = 0.0;
            break;
            case 10: //NWF
                    for(i=0; i<chrom_length; i++)
                    {
205                         if(rand()%33 <= 4) p_ind->genes[i]=0;
                            else p_ind->genes[i]=1;
                    }
                    p_ind->nwell_bias = 1.1;
                    p_ind->vdd = 1.1;
210                 p_ind->pwell_bias = 0.0;
            break;
            case 11: //NWF+VDD
                    for(i=0; i < chrom_length; i++)
                    {
215                         if(rand()%33 <= 4) p_ind->genes[i]=0;
                            else p_ind->genes[i]=1;
                    }
                    p_ind->vdd= vdd_min + rand() % ((int)((vdd_max - vdd_min)/bias_res)+1) * bias_res;
                    p_ind->nwell_bias = p_ind->vdd;
220                 p_ind->pwell_bias = 0.0;
            break;
            }


225     p_ind->f = (double*) chk_malloc(dim * sizeof(double));
        return (p_ind);
    }


230 void free_memory()
    /* Frees all memory. */
    {
        free_pop(pp_sel);
        free_pop(pp_all);
235     free_pop(pp_all_copy);
    //      free_pop(pp_cache);
    }


240 void free_pop(pop *pp)
    /* Frees memory for given population. */
    {
        assert(pp != NULL);

245     free(pp->ind);
        free(pp);
```

```
}

250 void free_ind(ind *p_ind)
/* Frees memory for given individual . */
{
     assert(p_ind != NULL);

255      free(p_ind->f);
     free(p_ind);
}


260
/*------------------------| selection functions|---------------------------*/

void selection()
{
265      int i;
     int size;

     /* Join offspring individuals from variator to population */
printf("in selction call mergeOffspring()\n");
270      mergeOffspring();

     size = pp_all->size;

     /* Create internal data structures for selection process */
275      /* Vectors */
     fitness_bucket = (int*) chk_malloc(size * size * sizeof(int));
     fitness_bucket_mod = (int*) chk_malloc(size * sizeof(int));
     copies = (int*) chk_malloc(size * sizeof(int));
     old_index = (int*) chk_malloc(size * sizeof(int));
280
     /* Matrices */
     dist = (double**) chk_malloc(size * sizeof(double*));
     NN = (int**) chk_malloc(size * sizeof(int*));
     for (i = 0; i < size; i++)
285      {
          dist[i] = (double*) chk_malloc(size * sizeof(double));
          NN[i] = (int*) chk_malloc(size * sizeof(int));
     }

290      /* Calculates SPEA2 fitness values for all individuals */
printf("in selction call calcFitness()\n");
     calcFitnesses();

     /* Calculates distance matrix dist[][] */
295 printf("in selction call calcDistances()\n");
     calcDistances();

     /* Performs environmental selection
          (truncates 'pp_all' to size 'popsize') */
300 printf("in selction call environmentalSelection()\n");
     environmentalSelection();

     /* Performs mating selection
          (fills mating pool / offspring population pp_sel */
305 printf("in selction call matingSelection()\n");
     matingSelection();

     /* Frees memory of internal data structures */
     free(fitness_bucket);
310      free(fitness_bucket_mod);
     free(copies);
     free(old_index);
     for (i = 0; i < size; i++)
     {
315           free(dist[i]);
          free(NN[i]);
     }
     free(dist);
     free(NN);
320
     return;
}


325 void mergeOffspring()
{
     int i;

     assert(pp_all->size + pp_new->size <= pp_all->maxsize);
330
     for (i = 0; i < pp_new->size; i++)
     {
          pp_all->ind[pp_all->size + i] = pp_new->ind[i];
     }
335
     pp_all->size += pp_new->size;

     free_pop(pp_new);
}
340

void calcFitnesses()
{
     int i, j;
345      int size;
     int *strength;

     size = pp_all->size;
     strength = (int*) chk_malloc(size * sizeof(int));
350
     /* initialize fitness and strength values */
     for (i = 0; i < size; i++)
     {
```

```
            pp_all->ind[i]->fitness = 0;
355         strength[i] = 0;
            fitness_bucket[i] = 0;
            fitness_bucket_mod[i] = 0;
            for (j = 0; j < size; j++)
            {
360             fitness_bucket[i * size + j] = 0;
            }
        }

        /* calculate strength values */
365     for (i = 0; i < size; i++)
        {
            for (j = 0; j < size; j++)
            {
                if (dominates(pp_all->ind[i], pp_all->ind[j]))
370             {
                    strength[i]++;
                }
            }
        }
375
        /* Fitness values = sum of strength values of dominators */
        for (i = 0; i < size; i++)
        {
            int sum = 0;
380         for (j = 0; j < size; j++)
            {
                if (dominates(pp_all->ind[j], pp_all->ind[i]))
                {
                    sum += strength[j];
385             }
            }
            pp_all->ind[i]->fitness = sum;
            fitness_bucket[sum]++;
            fitness_bucket_mod[(sum / size)]++;
390     }

        free(strength);
        return;
    }
395

    void calcDistances()
    {
        int i, j;
400     int size = pp_all->size;

        /* initialize copies[] vector and NN[][] matrix */
        for (i = 0; i < size; i++)
        {
405         copies[i] = 1;
            for (j = 0; j < size; j++)
            {
                NN[i][j] = -1;
            }
410
        }

        /* calculate distances */
        for (i = 0; i < size; i++)
415     {
            NN[i][0] = i;
            for (j = i + 1; j < size; j++)
            {
                dist[i][j] = calcDistance(pp_all->ind[i], pp_all->ind[j]);
420             assert(dist[i][j] < PISA_MAXDOUBLE);
                dist[j][i] = dist[i][j];
                if (dist[i][j] == 0)
                {
                    NN[i][copies[i]] = j;
425                 NN[j][copies[j]] = i;
                    copies[i]++;
                    copies[j]++;
                }
            }
430         dist[i][i] = 0;
        }
    }


435 int getNN(int index, int k)
    /* lazy evaluation of the k-th nearest neighbor
       pre-condition: (k-1)-th nearest neigbor is known already */
    {
        assert(index >= 0);
440     assert(k >= 0);
        assert(copies[index] > 0);

        if (NN[index][k] < 0)
        {
445         int i;
            double min_dist = PISA_MAXDOUBLE;
            int min_index = -1;
            int prev_min_index = NN[index][k-1];
            double prev_min_dist = dist[index][prev_min_index];
450         assert(prev_min_dist >= 0);

            for (i = 0; i < pp_all->size; i++)
            {
                double my_dist = dist[index][i];
455
                if (my_dist < min_dist && index != i)
                {
                    if (my_dist > prev_min_dist ||
                        (my_dist == prev_min_dist && i > prev_min_index))
460                 {
```

```
                    min_dist = my_dist;
                    min_index = i;
                }
            }
465     }

        NN[index][k] = min_index;
    }

470 return (NN[index][k]);
}


double getNNd(int index, int k)
475 /* Returns the distance to the k-th nearest neigbor
     if this individual is still in the population.
    For for already deleted individuals, returns -1 */
{
    int neighbor_index = getNN(index, k);
480
    if (copies[neighbor_index] == 0)
        return (-1);
    else
        return (dist[index][neighbor_index]);
485 }


void environmentalSelection()
{
490     int i;
    int new_size = 0;

    if (fitness_bucket[0] > popsize)
    {
495 printf("in environmentalSelection() calling truncate_nondominated()\n");
        truncate_nondominated();
    }
    else if (pp_all->size > popsize)
    {
500 printf("in environmentalSelection() calling truncate_dominated()\n");
        truncate_dominated();
    }

    /* Move remaining individuals to top of array in 'pp_all' */
505 printf("in environmentalSelection() reording pp_all\n");
    for (i = 0; i < pp_all->size; i++)
    {
        ind* temp_ind = pp_all->ind[i];
        if (temp_ind != NULL)
510     {
            assert(copies[i] > 0);
            pp_all->ind[i] = NULL;
            pp_all->ind[new_size] = temp_ind;
            old_index[new_size] = i;
515         new_size++;
        }
    }
    assert(new_size <= popsize);
    pp_all->size = new_size;
520
    return;
}


525 void truncate_nondominated()
/* truncate from nondominated individuals (if too many) */
{
    int i;

530 /* delete all dominated individuals */
    for (i = 0; i < pp_all->size; i++)
    {
        if (pp_all->ind[i]->fitness > 0)
        {
535         free_ind(pp_all->ind[i]);
            pp_all->ind[i] = NULL;
            copies[i] = 0;
        }
    }
540
    /* truncate from non-dominated individuals */
    while (fitness_bucket[0] > popsize)
    {
        int *marked;
545     int max_copies = 0;
        int count = 0;
        int delete_index;

        marked = (int*) chk_malloc(pp_all->size * sizeof(int));
550
        /* compute inds with maximal copies */
        for (i = 0; i < pp_all->size; i++)
        {
            if (copies[i] > max_copies)
555         {
                count = 0;
                max_copies = copies[i];
            }
            if (copies[i] == max_copies)
560         {
                marked[count] = i;
                count++;
            }
        }
565
        assert(count >= max_copies);
```

```
        if (count > max_copies)
        {
570         int *neighbor;
            neighbor = (int*) chk_malloc(count * sizeof(int));
            for (i = 0; i < count; i++)
            {
                neighbor[i] = 1;  /* pointers to next neighbor */
575         }

            while (count > max_copies)
            {
                double min_dist = PISA_MAXDOUBLE;
580             int count2 = 0;

                for (i = 0; i < count; i++)
                {
                    double my_dist = -1;
585                 while (my_dist == -1 && neighbor[i] < pp_all->size)
                    {
                        my_dist = getNNd(marked[i],neighbor[i]);
                        neighbor[i]++;
                    }
590
                    if (my_dist < min_dist)
                    {
                        count2 = 0;
                        min_dist = my_dist;
595                 }
                    if (my_dist == min_dist)
                    {
                        marked[count2] = marked[i];
                        neighbor[count2] = neighbor[i];
600                     count2++;
                    }
                }
                count = count2;
                if (min_dist == -1) /* all have equal distances */
605             {
                    break;
                }
            }
            free(neighbor);
610     }

        /* remove individual from population */
        delete_index = marked[irand(count)];
        free_ind(pp_all->ind[delete_index]);
615     pp_all->ind[delete_index] = NULL;
        for (i = 0; i < count; i++)
        {
            if (dist[delete_index][marked[i]] == 0)
            {
620             copies[marked[i]]--;
            }
        }
        copies[delete_index] = 0; /* Indicates that this index is empty */
        fitness_bucket[0]--;
625     fitness_bucket_mod[0]--;
        free(marked);
    }

    return;
630 }


void truncate_dominated()
/* truncate from dominated individuals */
635 {
    int i, j;
    int size;
    int num = 0;

640 size = pp_all->size;

    i = -1;
    while (num < popsize)
    {
645     i++;
        num += fitness_bucket_mod[i];
    }

    j = i * size;
650 num = num - fitness_bucket_mod[i] + fitness_bucket[j];
    while (num < popsize)
    {
        j++;
        num += fitness_bucket[j];
655 }

    if (num == popsize)
    {
        for (i = 0; i < size; i++)
660     {
            if (pp_all->ind[i]->fitness > j)
            {
                free_ind(pp_all->ind[i]);
                pp_all->ind[i] = NULL;
665         }
        }
    }
    else /* if not all fit into the next generation */
    {
670     int k;
        int free_spaces;
        int fill_level = 0;
        int *best;
```

```
675         free_spaces = popsize − (num − fitness_bucket[j]);
            best = (int∗) chk_malloc(free_spaces ∗ sizeof(int));
            for (i = 0; i < size; i++)
            {
                if ( pp_all−>ind[i]−>fitness > j)
680             {
                    free_ind (pp_all−>ind[i]);
                    pp_all−>ind[i] = NULL;
                }
                else if ( pp_all−>ind[i]−>fitness == j)
685             {
                    if ( fill_level < free_spaces)
                    {
                        best[ fill_level ] = i;
                        fill_level ++;
690                     for (k = fill_level − 1; k > 0; k−−)
                        {
                            int temp;
                            if (getNNd(best[k], 1) <= getNNd(best[k − 1], 1))
                            {
695                             break;
                            }
                            temp = best[k];
                            best[k] = best[k−1];
                            best[k−1] = temp;
700                     }
                    }
                    else
                    {
                        if (getNNd(i, 1) <= getNNd(best[free_spaces − 1], 1))
705                     {
                            free_ind (pp_all−>ind[i]);
                            pp_all−>ind[i] = NULL;
                        }
                        else
710                     {
                            free_ind (pp_all−>ind[best[free_spaces − 1]]);
                            pp_all−>ind[best[free_spaces − 1]] = NULL;
                            best[free_spaces − 1] = i;
                            for (k = fill_level − 1; k > 0; k−−)
715                         {
                                int temp;
                                if (getNNd(best[k], 1) <= getNNd(best[k − 1], 1))
                                {
                                    break;
720                             }
                                temp = best[k];
                                best[k] = best[k−1];
                                best[k−1] = temp;
                            }
725                     }
                    }
                }
            }
        }
730     return;
    }


    void matingSelection()
735 /* Fills mating pool 'pp_sel' */
    {
        int i, j;

        for (i = 0; i < parentpop_size; i++)
740     {
            int winner = irand(pp_all−>size);

            for (j = 1; j < tournament; j++)
            {
745             int opponent = irand(pp_all−>size);
                if ( pp_all−>ind[opponent]−>fitness
                    < pp_all−>ind[winner]−>fitness || winner == opponent)
                {
                    winner = opponent;
750             }
                else if ( pp_all−>ind[opponent]−>fitness
                         == pp_all−>ind[winner]−>fitness)
                {
                    if ( dist [old_index[opponent]][getNN(old_index[opponent], 1)] >
755                     dist [old_index[winner]][getNN(old_index[winner], 1)])
                    {
                        winner = opponent;
                    }
                }
760         }
            pp_sel−>ind[i] = pp_all−>ind[winner];
        }
        pp_sel−>size = parentpop_size;
    }
765

    void select_initial ()
    /* Performs initial selection. */
    {
770     selection ();
    }


    void select_normal()
775 /* Performs normal selection.*/
    {
        selection ();
    }

780
    int dominates(ind ∗p_ind_a, ind ∗p_ind_b)
```

```
    /* Determines if one individual dominates another.
       Minimizing fitness values. */
    {
785     int i;
        int a_is_worse = 0;
        int equal = 1;

         for (i = 0; i < dim && !a_is_worse; i++)
790      {
              a_is_worse = p_ind_a->f[i] > p_ind_b->f[i];
               equal = (p_ind_a->f[i] == p_ind_b->f[i]) && equal;
         }

795      return (!equal && !a_is_worse);
    }


    int is_equal(ind *p_ind_a, ind *p_ind_b)
800 /* Determines if two individuals are equal in all objective values.*/
    {
         int i;
         int equal = 1;

805      for (i = 0; i < dim; i++)
              equal = (p_ind_a->f[i] == p_ind_b->f[i]) && equal;

         return (equal);
    }
810

    double calcDistance(ind *p_ind_a, ind *p_ind_b)
    {
        int i;
815     double distance = 0;

        if ( is_equal(p_ind_a, p_ind_b) == 1)
        {
            return (0);
820     }

        for (i = 0; i < dim; i++){
            // printf ("dim: %d, a: %f, b: %f\n", dim, p_ind_a->f[i], p_ind_b->f[i]);
            distance += pow(p_ind_a->f[i]-p_ind_b->f[i],2);
825         // printf (" Distance: %f\n", distance);
        }

        return (sqrt(distance));
    }
830

    int irand(int range)
    /* Generate a random integer. */
    {
835     int j;
        j=(int) ((double)range * (double) rand() / (RAND_MAX+1.0));
        return (j);
    }

840
    /*--------------------| data exchange functions |------------------------*/

    void init_first_pop()
    {
845     int i;
        pp_new = create_pop(popsize, dim);

        for (i = 0; i < popsize; i++)
            pp_new->ind[i] = create_ind(dim);
850     pp_new->size = popsize;

        eval_pop(pp_new, popsize, dim);
    }

855 void crossover() // crossover the selected parents (pp_sel) to make a child population (pp_new)
    {
            int i, j;
            float par1, par2, smaller, larger;

860 //          free_pop(pp_new);
            pp_new = create_pop(childpop_size, dim);

            for(i = 0; i < childpop_size; i=i+2)
            {
865                 pp_new->ind[i] = create_ind(dim);
                    pp_new->ind[i+1] = create_ind(dim);


                    switch(runmode)
870                 {
                            case 1: //NWF+DWB
                                    for(j = 0; j < chrom_length; j++)
                                    {
                                            if(rand()%11 <= 5) //cross over prob ~.5
875                                         {
                                                    pp_new->ind[i]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                                    pp_new->ind[i+1]->genes[j] = pp_sel->ind[i]->genes[j];
                                            }
                                            else
880                                         {
                                            pp_new->ind[i]->genes[j] = pp_sel->ind[i]->genes[j];
                                            pp_new->ind[i+1]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                            }
                                    }
885
                                    // real-variable intermediate crossover
                                    // nwell_bias
                                    par1 = pp_sel->ind[i]->nwell_bias;
```

```
                                par2 = pp_sel->ind[i+1]->nwell_bias;
890
                                smaller = par1;
                                larger = par2;
                                if (par1 > par2) {smaller = par2; larger = par1;}

895                             pp_new->ind[i]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
                                pp_new->ind[i+1]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;

                                // pwell_bias
                                par1 = pp_sel->ind[i]->pwell_bias;
900                             par2 = pp_sel->ind[i+1]->pwell_bias;

                                smaller = par1;
                                larger = par2;
                                if (par1 > par2) {smaller = par2; larger = par1;}
905
                                pp_new->ind[i]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
                                pp_new->ind[i+1]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
                        break;
                        case 2: //NWF+NWB
910                             for(j = 0; j < chrom_length; j++)
                                {
                                        if(rand()%11 <= 5) //cross over prob ~.5
                                        {
                                                pp_new->ind[i]->genes[j] = pp_sel->ind[i+1]->genes[j];
915                                             pp_new->ind[i+1]->genes[j] = pp_sel->ind[i]->genes[j];
                                        }
                                        else
                                        {
                                        pp_new->ind[i]->genes[j] = pp_sel->ind[i]->genes[j];
920                                     pp_new->ind[i+1]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                        }
                                }

                                // real - variable   intermediate   crossover
925                             // nwell_bias
                                par1 = pp_sel->ind[i]->nwell_bias;
                                par2 = pp_sel->ind[i+1]->nwell_bias;

                                smaller = par1;
930                             larger = par2;
                                if (par1 > par2) {smaller = par2; larger = par1;}

                                pp_new->ind[i]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
                                pp_new->ind[i+1]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
935                     break;
                        case 3: //NWF+PWB
                                for(j = 0; j < chrom_length; j++)
                                {
                                        if(rand()%11 <= 5) //cross over prob ~.5
940                                     {
                                                pp_new->ind[i]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                                pp_new->ind[i+1]->genes[j] = pp_sel->ind[i]->genes[j];
                                        }
                                        else
945                                     {
                                        pp_new->ind[i]->genes[j] = pp_sel->ind[i]->genes[j];
                                        pp_new->ind[i+1]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                        }
                                }
950
                                // real - variable   intermediate   crossover
                                // pwell_bias
                                par1 = pp_sel->ind[i]->pwell_bias;
                                par2 = pp_sel->ind[i+1]->pwell_bias;
955
                                smaller = par1;
                                larger = par2;
                                if (par1 > par2) {smaller = par2; larger = par1;}

960                             pp_new->ind[i]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
                                pp_new->ind[i+1]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
                                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                                pp_new->ind[i+1]->nwell_bias = pp_new->ind[i+1]->vdd;
                        break;
965                     case 4: //DWB
                                // real - variable   intermediate   crossover
                                // nwell_bias
                                par1 = pp_sel->ind[i]->nwell_bias;
                                par2 = pp_sel->ind[i+1]->nwell_bias;
970
                                smaller = par1;
                                larger = par2;
                                if (par1 > par2) {smaller = par2; larger = par1;}

975                             pp_new->ind[i]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;
                                pp_new->ind[i+1]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                        bias_res;

                                // pwell_bias
                                par1 = pp_sel->ind[i]->pwell_bias;
980                             par2 = pp_sel->ind[i+1]->pwell_bias;

                                smaller = par1;
                                larger = par2;
                                if (par1 > par2) {smaller = par2; larger = par1;}
985
```

```
                pp_new->ind[i]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                        bias_res;
                pp_new->ind[i+1]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                        bias_res;
        break;
        case 5: //VDD
990
                // real - variable   intermediate   crossover
                //VDD
                par1 = pp_sel->ind[i]->vdd;
                par2 = pp_sel->ind[i+1]->vdd;
995
                smaller = par1;
                larger = par2;
                if (par1 > par2) {smaller = par2; larger = par1;}

1000            pp_new->ind[i]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                pp_new->ind[i+1]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                pp_new->ind[i+1]->nwell_bias = pp_new->ind[i+1]->vdd;
        break;
1005    case 6: //NWF+PWB+VDD
                for(j = 0; j < chrom_length; j++)
                {
                        if(rand()%11 <= 5) //cross over prob ~.5
                        {
1010                            pp_new->ind[i]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                pp_new->ind[i+1]->genes[j] = pp_sel->ind[i]->genes[j];
                        }
                        else
                        {
1015                    pp_new->ind[i]->genes[j] = pp_sel->ind[i]->genes[j];
                        pp_new->ind[i+1]->genes[j] = pp_sel->ind[i+1]->genes[j];
                        }
                }

1020            // pwell_bias
                par1 = pp_sel->ind[i]->pwell_bias;
                par2 = pp_sel->ind[i+1]->pwell_bias;

                smaller = par1;
1025            larger = par2;
                if (par1 > par2) {smaller = par2; larger = par1;}

                pp_new->ind[i]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                        bias_res;
                pp_new->ind[i+1]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                        bias_res;
1030
                //VDD
                par1 = pp_sel->ind[i]->vdd;
                par2 = pp_sel->ind[i+1]->vdd;

1035            smaller = par1;
                larger = par2;
                if (par1 > par2) {smaller = par2; larger = par1;}

                pp_new->ind[i]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
1040            pp_new->ind[i+1]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                pp_new->ind[i+1]->nwell_bias = pp_new->ind[i+1]->vdd;
        break;
        case 7: //PWB+VDD
1045            // pwell_bias
                par1 = pp_sel->ind[i]->pwell_bias;
                par2 = pp_sel->ind[i+1]->pwell_bias;

                smaller = par1;
1050            larger = par2;
                if (par1 > par2) {smaller = par2; larger = par1;}

                pp_new->ind[i]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                        bias_res;
                pp_new->ind[i+1]->pwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                        bias_res;
1055
                //VDD
                par1 = pp_sel->ind[i]->vdd;
                par2 = pp_sel->ind[i+1]->vdd;

1060            smaller = par1;
                larger = par2;
                if (par1 > par2) {smaller = par2; larger = par1;}

                pp_new->ind[i]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
1065            pp_new->ind[i+1]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                pp_new->ind[i+1]->nwell_bias = pp_new->ind[i+1]->vdd;
        break;
        case 8: //NWF+NWB+VDD
1070            for(j = 0; j < chrom_length; j++)
                {
                        if(rand()%11 <= 5) //cross over prob ~.5
                        {
                                pp_new->ind[i]->genes[j] = pp_sel->ind[i+1]->genes[j];
1075                            pp_new->ind[i+1]->genes[j] = pp_sel->ind[i]->genes[j];
                        }
                        else
                        {
                        pp_new->ind[i]->genes[j] = pp_sel->ind[i]->genes[j];
1080                    pp_new->ind[i+1]->genes[j] = pp_sel->ind[i+1]->genes[j];
                        }
                }

                // nwell_bias
1085            par1 = pp_sel->ind[i]->nwell_bias;
                par2 = pp_sel->ind[i+1]->nwell_bias;
```

```
                                        smaller = par1;
                                        larger = par2;
1090                                    if (par1 > par2) {smaller = par2; larger = par1;}

                                        pp_new->ind[i]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                            bias_res;
                                        pp_new->ind[i+1]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                            bias_res;
                                        //VDD
1095                                    par1 = pp_sel->ind[i]->vdd;
                                        par2 = pp_sel->ind[i+1]->vdd;

                                        smaller = par1;
                                        larger = par2;
1100                                    if (par1 > par2) {smaller = par2; larger = par1;}

                                        pp_new->ind[i]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                                        pp_new->ind[i+1]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                                break;
1105                            case 9: //NWB+VDD
                                        // nwell_bias
                                        par1 = pp_sel->ind[i]->nwell_bias;
                                        par2 = pp_sel->ind[i+1]->nwell_bias;

1110                                    smaller = par1;
                                        larger = par2;
                                        if (par1 > par2) {smaller = par2; larger = par1;}

                                        pp_new->ind[i]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                            bias_res;
1115                                    pp_new->ind[i+1]->nwell_bias = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) *
                                            bias_res;
                                        //VDD
                                        par1 = pp_sel->ind[i]->vdd;
                                        par2 = pp_sel->ind[i+1]->vdd;

1120                                    smaller = par1;
                                        larger = par2;
                                        if (par1 > par2) {smaller = par2; larger = par1;}

                                        pp_new->ind[i]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
1125                                    pp_new->ind[i+1]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                                break;
                                case 10: //NWF
                                        for(j = 0; j < chrom_length; j++)
                                        {
1130                                            if(rand()%11 <= 5) //cross over prob ~.5
                                                {
                                                        pp_new->ind[i]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                                        pp_new->ind[i+1]->genes[j] = pp_sel->ind[i]->genes[j];
                                                }
1135                                            else
                                                {
                                                pp_new->ind[i]->genes[j] = pp_sel->ind[i]->genes[j];
                                                pp_new->ind[i+1]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                                }
1140                                    }
                                        pp_new->ind[i]->vdd = 1.1;
                                        pp_new->ind[i+1]->vdd = 1.1;
                                        pp_new->ind[i]->nwell_bias = 1.1;
                                        pp_new->ind[i+1]->nwell_bias = 1.1;
1145                                    pp_new->ind[i]->pwell_bias = 0.0;
                                        pp_new->ind[i+1]->pwell_bias = 0.0;
                                break;
                                case 11: //NWF+VDD
                                        for(j = 0; j < chrom_length; j++)
1150                                    {
                                                if(rand()%11 <= 5) //cross over prob ~.5
                                                {
                                                        pp_new->ind[i]->genes[j] = pp_sel->ind[i+1]->genes[j];
                                                        pp_new->ind[i+1]->genes[j] = pp_sel->ind[i]->genes[j];
1155                                            }
                                                else
                                                {
                                                pp_new->ind[i]->genes[j] = pp_sel->ind[i]->genes[j];
                                                pp_new->ind[i+1]->genes[j] = pp_sel->ind[i+1]->genes[j];
1160                                            }
                                        }
                                        // real-variable intermediate crossover
                                        //VDD
                                        par1 = pp_sel->ind[i]->vdd;
1165                                    par2 = pp_sel->ind[i+1]->vdd;

                                        smaller = par1;
                                        larger = par2;
                                        if (par1 > par2) {smaller = par2; larger = par1;}
1170
                                        pp_new->ind[i]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;
                                        pp_new->ind[i+1]->vdd = smaller - bias_res + rand() % ((int)((larger - smaller)/bias_res)+3) * bias_res;

                                        pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
1175                                    pp_new->ind[i+1]->nwell_bias = pp_new->ind[i+1]->vdd;
                                break;
                        }

        }
1180    pp_new->size = childpop_size;
}

void mutate() //mutate the new child population
{
1185    int i, j;

        for(i=0; i < childpop_size; i++)
        {
```

```
1190            switch(runmode)
                {
                        case 1: //NWF+DWB
                                for(j=0; j < chrom_length; j++)
                                {
1195                                    if (rand() % (chrom_length + 1) <=1) //2/chrom_length mutation
                                        {
                                                if (pp_new->ind[i]->genes[j] == 0) pp_new->ind[i]->genes[j] = 1;
                                                else pp_new->ind[i]->genes[j] = 0;
                                        }
1200                            }

                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->nwell_bias -= bias_res;
1205                            }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->nwell_bias += bias_res;
                                }
1210
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->pwell_bias -= bias_res;
                                }
1215                            if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->pwell_bias += bias_res;
                                }
                        break;
1220                    case 2: //NWF+NWB
                                for(j=0; j < chrom_length; j++)
                                {
                                        if (rand() % (chrom_length + 1) <= 1) //2/chrom_length mutation
                                        {
1225                                            if (pp_new->ind[i]->genes[j] == 0) pp_new->ind[i]->genes[j] = 1;
                                                else pp_new->ind[i]->genes[j] = 0;
                                        }
                                }

1230                            if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->nwell_bias -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
1235                            {
                                        pp_new->ind[i]->nwell_bias += bias_res;
                                }
                        break;
                        case 3: //NWF+PWB
1240                            for(j=0; j < chrom_length; j++)
                                {
                                        if (rand() % (chrom_length + 1) <= 1) //2/chrom_length mutation
                                        {
                                                if (pp_new->ind[i]->genes[j] == 0) pp_new->ind[i]->genes[j] = 1;
1245                                            else pp_new->ind[i]->genes[j] = 0;
                                        }
                                }

                                if (rand() % 100 < 5) // 5% real mutation rate
1250                            {
                                        pp_new->ind[i]->pwell_bias -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1255                                    pp_new->ind[i]->pwell_bias += bias_res;
                                }
                                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                        break;
                        case 4: //DWB
1260                            if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->nwell_bias -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
1265                            {
                                        pp_new->ind[i]->nwell_bias += bias_res;
                                }

                                if (rand() % 100 < 5) // 5% real mutation rate
1270                            {
                                        pp_new->ind[i]->pwell_bias -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1275                                    pp_new->ind[i]->pwell_bias += bias_res;
                                }
                        break;
                        case 5: //VDD
                                if (rand() % 100 < 5) // 5% real mutation rate
1280                            {
                                        pp_new->ind[i]->vdd -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1285                                    pp_new->ind[i]->vdd += bias_res;
                                }
                                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                        break;
                        case 6: //NWF+PWB+VDD
1290                            for(j=0; j < chrom_length; j++)
                                {
                                        if (rand() % (chrom_length + 1) <= 1) //2/chrom_length mutation
                                        {
                                                if (pp_new->ind[i]->genes[j] == 0) pp_new->ind[i]->genes[j] = 1;
1295                                            else pp_new->ind[i]->genes[j] = 0;
                                        }
```

```
                                }

                                if (rand() % 100 < 5) // 5% real mutation rate
1300                            {
                                        pp_new->ind[i]->pwell_bias -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1305                                    pp_new->ind[i]->pwell_bias += bias_res;
                                }

                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1310                                    pp_new->ind[i]->vdd -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->vdd += bias_res;
1315                            }
                                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                        break;
                        case 7: //PWB+VDD
                                if (rand() % 100 < 5) // 5% real mutation rate
1320                            {
                                        pp_new->ind[i]->pwell_bias -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1325                                    pp_new->ind[i]->pwell_bias += bias_res;
                                }

                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1330                                    pp_new->ind[i]->vdd -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->vdd += bias_res;
1335                            }
                                pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                        break;
                        case 8: //NWF+NWB+VDD
                                for(j=0; j < chrom_length; j++)
1340                            {
                                        if (rand() % (chrom_length + 1) <= 1) //2/chrom_length mutation
                                        {
                                                if (pp_new->ind[i]->genes[j] == 0) pp_new->ind[i]->genes[j] = 1;
                                                else pp_new->ind[i]->genes[j] = 0;
1345                                    }
                                }

                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
1350                                    pp_new->ind[i]->nwell_bias -= bias_res;
                                }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->nwell_bias += bias_res;
1355                            }

                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->vdd -= bias_res;
1360                            }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->vdd += bias_res;
                                }
1365                    break;
                        case 9: //NWB+VDD
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->nwell_bias -= bias_res;
1370                            }
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->nwell_bias += bias_res;
                                }
1375
                                if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->vdd -= bias_res;
                                }
1380                            if (rand() % 100 < 5) // 5% real mutation rate
                                {
                                        pp_new->ind[i]->vdd += bias_res;
                                }
                        break;
1385                    case 10: //NWF
                                for(j=0; j < chrom_length; j++)
                                {
                                        if (rand() % (chrom_length + 1) <= 1) //2/chrom_length mutation
                                        {
1390                                            if (pp_new->ind[i]->genes[j] == 0) pp_new->ind[i]->genes[j] = 1;
                                                else pp_new->ind[i]->genes[j] = 0;
                                        }
                                }
                                pp_new->ind[i]->vdd = 1.1;
1395                            pp_new->ind[i]->nwell_bias = 1.1;
                                pp_new->ind[i]->pwell_bias = 0.0;
                        break;
                        case 11: //NWF+VDD

1400                            for(j=0; j < chrom_length; j++)
                                {
                                        if (rand() % (chrom_length + 1) <= 1) //2/chrom_length mutation
                                        {
```

```
                                                    if (pp_new->ind[i]->genes[j] == 0) pp_new->ind[i]->genes[j] = 1;
1405                                                else pp_new->ind[i]->genes[j] = 0;
                                            }
                                    }

                                    if (rand() % 100 < 5) // 5% real mutation rate
1410                                {
                                            pp_new->ind[i]->vdd -= bias_res;
                                    }
                                    if (rand() % 100 < 5) // 5% real mutation rate
                                    {
1415                                        pp_new->ind[i]->vdd += bias_res;
                                    }

                                    pp_new->ind[i]->nwell_bias = pp_new->ind[i]->vdd;
                            break;
1420                    }
            }
    }

    void bias_search1()
1425 {
            int size, i;
    printf("running bias_search1()\n");

            size = pp_all->size;
1430
    // printf (" freeing pp_new\n");
    //         free_pop (pp_new);
    printf("creating pp_new\n");
            pp_new = create_pop(2*size, dim);
1435
            // create 2 new inds in pp_new based on pp_sel ind with biasing in +/- direction
            for(i=0; i<size; i++)
            {
    printf("creating pp_new-ind[%i]\n",i);
1440                pp_new->ind[i] = create_ind(dim);
    printf("creating pp_new-ind[%i]\n",i+size);
                    pp_new->ind[i+size] = create_ind(dim);
            }

1445        for(i=0; i<size; i++)
            {
                    int j;
    printf("copying pp_sel[%i] to pp_new[%i]\n", i, i);
                    pp_new->ind[i]->pwell_bias = pp_all->ind[i]->pwell_bias;
1450                pp_new->ind[i]->nwell_bias = pp_all->ind[i]->nwell_bias;
                    pp_new->ind[i]->vdd = pp_all->ind[i]->vdd;
                    for(j=0; j<chrom_length; j++) pp_new->ind[i]->genes[j] = pp_all->ind[i]->genes[j];
    printf("copying pp_sel[%i] to pp_new[%i]\n", i+size, i+size);
                    pp_new->ind[i+size]->pwell_bias = pp_all->ind[i]->pwell_bias;
1455                pp_new->ind[i+size]->nwell_bias = pp_all->ind[i]->nwell_bias;
                    pp_new->ind[i+size]->vdd = pp_all->ind[i]->vdd;
                    for(j=0; j<chrom_length; j++) pp_new->ind[i+size]->genes[j] = pp_all->ind[i]->genes[j];

    printf("changing pp_new[%i] and pp_new[%i] biasing\n", i, i+size);
1460 printf("pp_new->ind[%i]->biases = %f %f %f\n", i, pp_new->ind[i]->pwell_bias, pp_new->ind[i]->nwell_bias, pp_new->ind[i]->vdd);
    printf("pp_new->ind[%i]->biases = %f %f %f\n", i+size, pp_new->ind[i+size]->pwell_bias, pp_new->ind[i+size]->nwell_bias, pp_new->
        ind[i+size]->vdd);

                    switch(runmode)
                    {
1465                        case 1: //NWF+DWB
                                    pp_new->ind[i]->nwell_bias += bias_res;
                                    pp_new->ind[i+size]->nwell_bias -= bias_res;
                            break;
                            case 2: //NWF+NWB
1470                                pp_new->ind[i]->nwell_bias += bias_res;
                                    pp_new->ind[i+size]->nwell_bias -= bias_res;
                            break;
                            case 3: //NWF+PWB
                                    pp_new->ind[i]->pwell_bias += bias_res;
1475                                pp_new->ind[i+size]->pwell_bias -= bias_res;
                            break;
                            case 6: //NWF+PWB+VDD
                                    pp_new->ind[i]->pwell_bias += bias_res;
                                    pp_new->ind[i+size]->pwell_bias -= bias_res;
1480                        break;
                            case 8: //NWF+NWB+VDD
                                    pp_new->ind[i]->nwell_bias += bias_res;
                                    pp_new->ind[i+size]->nwell_bias -= bias_res;
                            break;
1485                        case 11: //NWF+VDD
                                    pp_new->ind[i]->vdd += bias_res;
                                    pp_new->ind[i+size]->vdd -= bias_res;
                            break;
                            default:
1490                                printf("Bad runmode in bias_search1()!\n");
                    }
    printf("pp_new->ind[%i]->biases = %f %f %f\n", i, pp_new->ind[i]->pwell_bias, pp_new->ind[i]->nwell_bias, pp_new->ind[i]->vdd);
    printf("pp_new->ind[%i]->biases = %f %f %f\n", i+size, pp_new->ind[i+size]->pwell_bias, pp_new->ind[i+size]->nwell_bias, pp_new->
        ind[i+size]->vdd);
            }
1495
            //update pp_new size
    printf("setting pp_new size to %i\n", 2*size);
            pp_new->size = 2 * size;

1500        // evaluate pp_new ( eval_new () )
    printf("evaluating pp_new\n");
            eval_new();

    write_new();
1505
            // select inds from pp_all ( selection () )
    printf("doing selection\n");
            selection ();
```

```
     }
1510
     void bias_search2()
     {
             int size, i;
     printf("running bias_search2()\n");
1515
             size = pp_all->size;

     // printf (" freeing  pp_new\n");
     //         free_pop (pp_new);
1520 printf("creating pp_new\n");
             pp_new = create_pop(2*size, dim);

             // create 2 new inds in pp_new based on pp_sel ind with biasing in +/- direction
             for(i=0; i<size; i++)
1525         {
     printf("creating pp_new-ind[%i]\n",i);
                     pp_new->ind[i] = create_ind(dim);
     printf("creating pp_new-ind[%i]\n",i+size);
                     pp_new->ind[i+size] = create_ind(dim);
1530         }

             for(i=0; i<size; i++)
             {
                     int j;
1535 printf("copying pp_sel[%i] to  pp_new[%i]\n", i, i);
                     pp_new->ind[i]->pwell_bias = pp_all->ind[i]->pwell_bias;
                     pp_new->ind[i]->nwell_bias = pp_all->ind[i]->nwell_bias;
                     pp_new->ind[i]->vdd = pp_all->ind[i]->vdd;
                     for(j=0; j<chrom_length; j++) pp_new->ind[i]->genes[j] = pp_all->ind[i]->genes[j];
1540 printf("copying pp_sel[%i] to  pp_new[%i]\n", i+size, i+size);
                     pp_new->ind[i+size]->pwell_bias = pp_all->ind[i]->pwell_bias;
                     pp_new->ind[i+size]->nwell_bias = pp_all->ind[i]->nwell_bias;
                     pp_new->ind[i+size]->vdd = pp_all->ind[i]->vdd;
                     for(j=0; j<chrom_length; j++) pp_new->ind[i+size]->genes[j] = pp_all->ind[i]->genes[j];
1545
     printf("changing pp_new[%i] and pp_new[%i] biasing\n", i, i+size);
     printf("pp_new->ind[%i]->biases = %f %f %f\n", i, pp_new->ind[i]->pwell_bias, pp_new->ind[i]->nwell_bias, pp_new->ind[i]->vdd);
     printf("pp_new->ind[%i]->biases = %f %f %f\n", i+size, pp_new->ind[i+size]->pwell_bias, pp_new->ind[i+size]->nwell_bias, pp_new->
            ind[i+size]->vdd);

1550                 switch(runmode)
                     {
                             case 1: //NWF+DWB
                                     pp_new->ind[i]->pwell_bias += bias_res;
                                     pp_new->ind[i+size]->pwell_bias -= bias_res;
1555                         break;
                             case 6: //NWF+PWB+VDD
                                     pp_new->ind[i]->vdd += bias_res;
                                     pp_new->ind[i+size]->vdd -= bias_res;
                             break;
1560                         case 8: //NWF+NWB+VDD
                                     pp_new->ind[i]->vdd += bias_res;
                                     pp_new->ind[i+size]->vdd -= bias_res;
                             break;
                             default:
1565                                 printf("Bad runmode in bias_search1()!\n");
                     }
     printf("pp_new->ind[%i]->biases = %f %f %f\n", i, pp_new->ind[i]->pwell_bias, pp_new->ind[i]->nwell_bias, pp_new->ind[i]->vdd);
     printf("pp_new->ind[%i]->biases = %f %f %f\n", i+size, pp_new->ind[i+size]->pwell_bias, pp_new->ind[i+size]->nwell_bias, pp_new->
            ind[i+size]->vdd);
             }
1570
             //update pp_new size
     printf("setting pp_new size to %i\n", 2*size);
             pp_new->size = 2 * size;

1575         // evaluate  pp_new ( eval_new () )
     printf("evaluating pp_new\n");
             eval_new();

     write_new();
1580
             // select  inds from pp_all ( selection () )
     printf("doing selection\n");
             selection();
     }
1585

     void copy_arc()
     {
             int i, size;
1590         printf("In copy_arc(): copying pp_all\n");
             size = pp_all->size;
             pp_all_copy = create_pop(size, dim);
             for(i=0; i<size; i++) pp_all_copy->ind[i] = create_ind(dim);

1595         for(i=0; i<size; i++)
             {
                     int j;
                     printf("copying pp_all[%i] to  pp_all_copy[%i]\n", i, i);
                     pp_all_copy->ind[i]->pwell_bias = pp_all->ind[i]->pwell_bias;
1600                 pp_all_copy->ind[i]->nwell_bias = pp_all->ind[i]->nwell_bias;
                     pp_all_copy->ind[i]->vdd = pp_all->ind[i]->vdd;
                     for(j=0; j<chrom_length; j++) pp_all_copy->ind[i]->genes[j] = pp_all->ind[i]->genes[j];
             }
             pp_all_copy->size = size;
1605         printf("Leaving copy_arc()\n");
     }

     void recover_arc()
     {
1610         int size, i;
             printf("In recover_arc(): copying pp_all_copy()\n");
             size = pp_all_copy->size;
             for(i=0; i<size; i++)
```

```
          {
1615              int j;
                  printf("copying pp_all_copy[%i] to pp_all[%i]\n", i, i);
                  pp_all->ind[i]->pwell_bias = pp_all_copy->ind[i]->pwell_bias;
                  pp_all->ind[i]->nwell_bias = pp_all_copy->ind[i]->nwell_bias;
                  pp_all->ind[i]->vdd = pp_all_copy->ind[i]->vdd;
1620              for(j=0; j<chrom_length; j++) pp_all->ind[i]->genes[j] = pp_all_copy->ind[i]->genes[j];
          }
          pp_all->size = size;
          printf("Leaving recover_arc()\n");
      }
1625

      void eval_new()
      {
          eval_pop(pp_new, pp_new->size, dim);
1630  }


      void write_sel()
      {
1635      char filename[128];
          sprintf(filename, "%d.selected.txt", chip_num);
          write_pop(filename, pp_sel, parentpop_size);
      }

1640  void write_new()
      {
          char filename[128];
          sprintf(filename, "m%i/%d.new.txt", runmode, chip_num);
          write_pop(filename, pp_new, pp_new->size);
1645  }

      void write_arc()
      {
           char filename[128];
1650       sprintf(filename, "m%i/%d.archive.txt", runmode, chip_num);
           write_pop(filename, pp_all, pp_all->size);
      }
```

```
      /*========================================================================
        PISA (www.tik.ee.ethz.ch/pisa/)
        ========================================================================
        Computer Engineering (TIK)
 5      ETH Zurich
        ========================================================================
        SPEA2 - Strength Pareto EA 2

        Implements data exchange trough files.
10
         file: spea2_io.c
         author: Marco Laumanns, laumanns@tik.ee.ethz.ch
         last change: 31.10.02

15          Modified by JLG 14.07.03
        ========================================================================
      */


20    #include <stdlib.h>
      #include <stdio.h>
      #include <string.h>
      #include <assert.h>
      #include <math.h>
25
      #include "spea2i.h"
      void WriteCircuit(pop *pop_ptr, int ind_num);
      //int check_cache(ind *ind1);
      // inds_equal(ind *ind1, ind *ind2);
30
      char* create_file_command = "cat /net/etltrk6/mnt/a/iabb/MOadder32_V5/spice/runinputs.vdiv3_0x1r0/%i.cserun.runinput %s /net/etltrk6/
             mnt/a/iabb/MOadder32_V5/spice/tail.runinput.vdiv3 > %s/spice/case.%s%i/cserun.runinput";
      char* spice_path = "/net/etltrk6/mnt/a/iabb/MOadder32_V5";
      char* case_name = "moadd";

35    float avg_power=4.4314e-4;
      float avg_delay=1/(2.8808e+8);

      /*int check_cache(ind *ind1)
      {
40              int i;
                printf("Checking cache ");
                for(i=0; i < pp_cache->size; i++)
                {
                        printf("%i ", i);
45                      if( inds_equal(ind1, pp_cache->ind[i])==1)
                        {
                                ind1->f[0] = pp_cache->ind[i]->f[0];
                                ind1->f[1] = pp_cache->ind[i]->f[1];
                                printf("\n");
50                              return(1);
                        }
                }
                printf("\n");
                return(0);
55    }

      int inds_equal(ind *ind1, ind *ind2)
      {
                int k;
60              if(ind1->pwell_bias != ind2->pwell_bias) return(0);
                if(ind1->nwell_bias != ind2->nwell_bias) return(0);
                if(ind1->vdd != ind2->vdd) return(0);
                for(k=0; k<chrom_length; k++)
                {
```

```
65                      if (ind1->genes[k] != ind2->genes[k]) return (0);
            }
            return (1);
    }
    */
70
    int eval_pop(pop *pp, int size, int dim)
    /* Evals individuals from pop */
    {
        int j, i;
75      FILE* p;
        char command[4096], buffer[4096];
        float d1, d2;

        assert(dim >= 0);
80      assert(pp != NULL);

        for (j = 0; j < size; j++)
        {
            if(pp->ind[j]->nwell_bias < nwell_bias_min) pp->ind[j]->nwell_bias = nwell_bias_min;
85          if(pp->ind[j]->nwell_bias > nwell_bias_max) pp->ind[j]->nwell_bias = nwell_bias_max;
            if(pp->ind[j]->pwell_bias < pwell_bias_min) pp->ind[j]->pwell_bias = pwell_bias_min;
            if(pp->ind[j]->pwell_bias > pwell_bias_max) pp->ind[j]->pwell_bias = pwell_bias_max;
            if(pp->ind[j]->vdd < vdd_min) pp->ind[j]->vdd = vdd_min;
            if(pp->ind[j]->vdd > vdd_max) pp->ind[j]->vdd = vdd_max;
90

    /*      if ( check_cache (pp->ind[j]) ==1)
            {
                    FILE*   hitlist_file ;
95                  sprintf (command, "%i. hitlist . txt ", chip_num);
                    hitlist_file =fopen(command, "a");
                    assert ( hitlist_file != NULL);

                    printf ("Cache hit %i %f %f %f %f %f ", chip_num, pp->ind[j]->f[0], pp->ind[j]->f[1], pp->ind[j]->pwell_bias, pp->ind[j]->
                        nwell_bias, pp->ind[j]->vdd);
100                 for (i =0; i < chrom_length ; i++) printf ("%i", pp->ind[j]->genes[i]);
                    printf ("\n");

                    fprintf ( hitlist_file , "%i %f %f %f %f %f ", chip_num, pp->ind[j]->f[0], pp->ind[j]->f[1], pp->ind[j]->pwell_bias, pp->ind[j
                        ]->nwell_bias, pp->ind[j]->vdd);
                    for (i =0; i < chrom_length ; i++) fprintf ( hitlist_file , "%i ", pp->ind[j]->genes[i]);
105                 fprintf ( hitlist_file , "\n");

                    fclose ( hitlist_file );
                    continue ;
            }
110 */
            WriteCircuit(pp, j);

                    sprintf(command, ". /net/etltrk6/mnt/a/iabb/source_path/shbp; ispice -j -d -c %s%i %s", case_name, case_num, spice_path
                        );
                    system(command);
115
                    sprintf(command, ". /net/etltrk6/mnt/a/iabb/source_path/shbp; cd %s/spice/case.%s%i; hpspice -p raw alias -s -
                        nowindows -c \"set noformat\" -c \"include %s/spice/measure\"", spice_path, case_name, case_num, spice_path);
                    p = popen(command, "r");

                    if(!p){
120                         printf("pipe failed for delay1\n");
                            exit(0);
                    }
                    fscanf(p, "%s",buffer);
                    fscanf(p, "%s",buffer);
125                 fscanf(p, "%s",buffer);
                    fscanf(p, "%s",buffer);
                    fscanf(p, "%s",buffer);
                    pp->ind[j]->f[1]= pp->ind[j]->vdd * fabs(atof(buffer));
                    fscanf(p, "%s",buffer);
130                 pp->ind[j]->f[1] += pp->ind[j]->vdd * fabs(atof(buffer));
                    pp->ind[j]->f[1] = pp->ind[j]->f[1] / avg_power;
                    if(pp->ind[j]->f[1] > 500 || pp->ind[j]->f[1] < 0.0) pp->ind[j]->f[1]=500;

                    fscanf(p, "%s",buffer);
135                 d1 = atof(buffer);
                    fscanf(p, "%s",buffer);
                    d2 = atof(buffer);
                    pclose(p);
                    pp->ind[j]->f[0] = (d1 + d2)/(2*avg_delay);
140                 if(pp->ind[j]->f[0] > 500 || pp->ind[j]->f[0] < 0.0) pp->ind[j]->f[0]=500;

                    sprintf(command, "m%i/%i.cache.txt", runmode, chip_num);
                    cache_file = fopen(command,"a");
                    assert( cache_file != NULL);
145
                    fprintf( cache_file , "%f %f %f %f %f ", pp->ind[j]->f[0], pp->ind[j]->f[1], pp->ind[j]->pwell_bias, pp->ind[j]->
                        nwell_bias, pp->ind[j]->vdd);
                    for(i=0; i<chrom_length; i++) fprintf(cache_file, "%i", pp->ind[j]->genes[i]);
                    fprintf( cache_file , "\n");
                    fclose( cache_file );
150
    /*              pp_cache->ind[pp_cache->size]->f[0] = pp->ind[j]->f[0];
                    pp_cache->ind[pp_cache->size]->f[1] = pp->ind[j]->f[1];
                    pp_cache->ind[pp_cache->size]->pwell_bias = pp->ind[j]->pwell_bias;
                    pp_cache->ind[pp_cache->size]->nwell_bias = pp->ind[j]->nwell_bias;
155                 pp_cache->ind[pp_cache->size]->vdd = pp->ind[j]->vdd;
                    for (i=0; i< chrom_length ; i++) pp_cache->ind[pp_cache->size]->genes[i] = pp->ind[j]->genes[i];
                    pp_cache->size = pp_cache->size + 1;
    */
            }
160
            return (0);  /* signalling that reading was successful */

    }

165 void write_pop(char* filename, pop* pp, int size)
    /* Writes a pop or sample to a given filename . */
```

```
{
        int i, j;
        FILE *fp;
170
        assert(0 <= size <= pp->size);

        fp = fopen(filename, "w");
        assert(fp != NULL);
175
        fprintf(fp, "%i.%i\n", gen, bias_gen); /* generation number */

        for (i = 0; i < size; i++)
        {
180             fprintf(fp, "%f %f %f %f %f ", pp->ind[i]->f[0], pp->ind[i]->f[1], pp->ind[i]->pwell_bias, pp->ind[i]->nwell_bias, pp->ind[
                    i]->vdd);
                    for(j = 0; j < chrom_length; j++) fprintf(fp, "%d", pp->ind[i]->genes[j]);
            fprintf(fp, "\n");
        }

185     fclose(fp);
}

void WriteCircuit(pop *pop_ptr, int ind_num)
{
190
/*
genes[0-31]=adder floats
xbin[0]= floats
xbin[1]=VDD
195 xbin[2]=NWB
xbin[3]=PWB
*/

            char fname[4096];
200         char command[4096];

            FILE* f;

            /* define the node names (numbers) for all the control inputs */
205         int *gene_ptr, i,
            vdiv_node_name[]=
             {2790, 2823, 2835, 2838, 2841, 2844, 2847, 2850, 2853,
             2760, 2763, 2766, 2769, 2772, 2775, 2778, 2781, 2784, 2787,
             2793, 2796, 2799, 2802, 2805, 2808, 2811, 2814, 2817, 2820,
210         2826, 2829, 2832},

            pullup_node_name[]=
             {2789, 2822, 2834, 2837, 2840, 2843, 2846, 2849, 2852,
             2759, 2762, 2765, 2768, 2771, 2774, 2777, 2780, 2783, 2786,
215          2792, 2795, 2798, 2801, 2804, 2807, 2810, 2813, 2816, 2819,
            2825, 2828, 2831},

            nwell_bias_node_name = 2854,
            vdiv_vdd_node_name = 2855,
220         vdd_node_name = 1,
            substrate_node_name= 2757;

            gene_ptr = &(pop_ptr->ind[ind_num]->genes[0]);

225         sprintf(fname,"%s/spice/case.%s%d/tmp.connect", spice_path, case_name, case_num);
            f = fopen(fname,"w");

            if(!f){
                    printf("\n\nCould not open file: %s\n",fname);
230                 exit(1);
            }

            // print the nwell voltage
            fprintf(f, "40 1\n");
235         fprintf(f, "%i 0 %f 0 0\n", nwell_bias_node_name, pop_ptr->ind[ind_num]->nwell_bias);
            fprintf(f, "103\n");

            /* print VDD value in the runinput form*/
            fprintf(f, "40 1\n");
240         fprintf(f, "%i 0 %f 0 0\n", vdd_node_name, pop_ptr->ind[ind_num]->vdd);
            fprintf(f, "103\n");

            /* print the pullup_control signals in the runinput form */
            for(i = chrom_length-1; i >= 0; i--)
245         {
                    fprintf(f, "40 1\n");
                    fprintf(f, "%i 0 %f 0 0\n", pullup_node_name[i], *gene_ptr == 1 ? 0.0 : pop_ptr->ind[ind_num]->vdd);
                    fprintf(f, "103\n");
                    gene_ptr++;
250         }

            gene_ptr = &(pop_ptr->ind[ind_num]->genes[0]);

            /* print the vdiv control signals */
255         for(i = chrom_length-1; i >= 0; i--)
            {
                    fprintf(f, "40 1\n");
                    fprintf(f, "%i 0 %f 0 0\n", vdiv_node_name[i], *gene_ptr == 1 ? pop_ptr->ind[ind_num]->vdd : 0.0);
                    fprintf(f, "103\n");
260                 gene_ptr++;
            }

            // print the substrate voltage
            fprintf(f, "40 1\n");
265         fprintf(f, "%i 0 %f 0 0\n", substrate_node_name, pop_ptr->ind[ind_num]->pwell_bias);
            fprintf(f, "103\n");

            // print the vdiv vdd node voltage
            fprintf(f, "40 1\n");
270         fprintf(f, "%i 0 %f 0 0\n", vdiv_vdd_node_name, pop_ptr->ind[ind_num]->vdd);
            fprintf(f, "103\n");
```

```
            fclose(f);

275         sprintf(command, create_file_command, chip_num, fname, spice_path, case_name, case_num);
        system(command);

    }
```

```
    /*========================================================================
      PISA  (www.tik.ee.ethz.ch/pisa/)
      ========================================================================
      Computer Engineering (TIK)
5     ETH Zurich
      ========================================================================
      SPEA2 - Strength Pareto EA 2

       Implementation in C for the selector side.
10
       Implements Petri net.

       file: spea2.c
       author: Marco Laumanns, laumanns@tik.ee.ethz.ch
15     last change: 31.10.02

         Modified by JLG 14.07.03
      ========================================================================
    */
20
    /* CAUTION: <unistd.h> is not standard C
       It is used only for sleep() and usleep() in wait().
       In Windows use Sleep() in <windows.h> or implement busy waiting.
    */
25
    #include <stdlib.h>
    #include <stdio.h>
    #include <assert.h>
    #include <math.h>
30
    #include "spea2i.h"

    #ifdef PISA_UNIX
    #include <unistd.h>
35  #endif

    #ifdef PISA_WIN
    #include <windows.h>
    #endif
40
    /*-------------------------------| main() |--------------------------------*/

    int main(int argc, char* argv[])
    {
45      /* command line parameters */
        popsize = 30;
        assert(popsize > 0);

        parentpop_size = 30;
50      assert(parentpop_size > 0);

        childpop_size = parentpop_size;
        assert(childpop_size > 0);

55      starting_chip = atoi(argv[1]);
        assert(starting_chip > 0);

        final_chip = atoi(argv[2]);
        assert(final_chip > 0);
60      assert(starting_chip <= final_chip);

        bias_res = atof(argv[3]);
        assert(bias_res > 0.0);

65  //      overall_runmode = atoi(argv[4]);
        runmode = 10;
    // assert (runmode == 1 || runmode == 2 || runmode == 3 ||runmode ==6||runmode ==8||runmode==10||runmode==11);
            case_num = atoi(argv[4]);
            assert(case_num >= 0);
70
        tournament = 2;
        max_generations = 50;
        gen=0;

75      dim = 2;

          for(chip_num = starting_chip; chip_num <= final_chip; chip_num++){

                  initialize();
80                init_first_pop();    /* make and eval first pop*/
                  bias_gen = 0;

                  for(gen = 1; gen <= max_generations; gen++){
    printf("***********************************GENERATION %i ***********************\n", gen);
85                        selection(); /* do selection */
                          write_arc();       /* write arc file (all individuals
                                                  that could ever be used again) */

                          // crossover, mutate
90                        crossover();
                          mutate();

                          eval_new(); /* eval children */
                  }
95
                  selection();
                  write_arc();
     //             write_sel();
```

```
     //                    runmode =  overall_runmode ;
100
                           copy_arc();

                           runmode = 1; //NWF+DWB
                           for(bias_gen=1; bias_gen<=4; bias_gen++)
105                        {
                                   bias_search1();
                                   write_arc();
                                   bias_search2();
                                   write_arc();
110                        }

                           recover_arc();

                           runmode = 2; //NWF+NWB
115                        for(bias_gen=1; bias_gen<=4; bias_gen++)
                           {
                                   bias_search1();
                                   write_arc();
                           }
120
                           recover_arc();

                           runmode = 3; //NWF+PWB
                           for(bias_gen=1; bias_gen<=4; bias_gen++)
125                        {
                                   bias_search1();
                                   write_arc();
                           }

130                        recover_arc();

                           runmode = 6; //NWF+PWB+VDD
                           for(bias_gen=1; bias_gen<=4; bias_gen++)
                           {
135                                bias_search1();
                                   write_arc();
                                   bias_search2();
                                   write_arc();
                           }
140
                           recover_arc();

                           runmode = 8; //NWF+NWB+VDD
                           for(bias_gen=1; bias_gen<=4; bias_gen++)
145                        {
                                   bias_search1();
                                   write_arc();
                                   bias_search2();
                                   write_arc();
150                        }

                           recover_arc();

                           runmode = 11; //NWF+VDD
155                        for(bias_gen=1; bias_gen<=4; bias_gen++)
                           {
                                   bias_search1();
                                   write_arc();
                           }
160
                           recover_arc();

                           free_memory();
                           runmode = 10;
165          }

             return (0);
     }

170
     /*--------------------| functions for control flow |----------------------*/

     void write_flag(char* filename, int flag)
     /* Write the state flag to given file. */
175  {
             FILE *fp;

             assert(0 <= flag <= 11);

180          fp = fopen(filename, "w");
             assert(fp != NULL);
             fprintf(fp, "%d", flag);
             fclose(fp);
     }
185

     int read_flag(char* filename)
     /* Read state flag from given file. */
     {
190          int result;
             int flag = -1;
             FILE *fp;
             fp = fopen(filename, "r");
             if (fp != NULL)
195          {
                     result = fscanf(fp, "%d", &flag);
                     fclose(fp);
                     if (result == 1) /* excatly one element read */
                     {
200                          if (flag < 0 || flag > 11)
                                     PISA_ERROR("Selector: Invalid state read from file.");
                     }
             }
             return (flag);
205  }
```

```
     /* void  wait ( double  sec )
     // Makes the  calling  process  sleep .
210  {
             assert ( sec  >= 0);

     # ifdef  PISA_UNIX
             sleep (( unsigned  int )  ( floor ( sec ))) ;
215          usleep (( unsigned  int )  ( sec  −  floor ( sec ))  ∗  1 e6) ;
             //  usleep  can  fail  if  argument  greater  than  1 e6
     # endif

     # ifdef  PISA_WIN
220          Sleep (( unsigned  int )  floor ( sec  ∗  1 e3)) ;
     # endif
     }*/
```

Appendix C

soIWABB and gIWABB analysis tools source code

Main code

This is the soIWABB version. The gIWABB code is very similar.

```
#include <stdio.h>
#include <fstream>
#include <stdlib.h>
#include <math.h>
//#include < string .h>

typedef struct ind_st /* an individual */
{
    int chip_num;
    float delay, power, fitness , pwell_bias , nwell_bias , vdd;
    int genes[32];
} ind;


main(int argc, char** argv)
{
        if(argc == 1){printf("This program makes a yield report for IABB files\n");
                printf("based on the n.best.mode.txt files in mM directories\n");
                printf("Usage: command # (# #...)\n");
                printf("Where # is a mode to be summarized.\n");
                printf("Input: ./mM/n.best.mode.txt for M=8,9,10 and n=1-100\n");
                printf("Output: ./iabb_yields.txt\n");
                exit(1);}
        const int max_modes = argc;

//      ind best_pop [100];
//      for ( int i=0; i<100; i++) best_pop [ i ]. delay = 500;

        char temp_filename[256];
        char buffer[256];


        sprintf(temp_filename, "iabb_yields.txt");
        FILE* yields_file = fopen(temp_filename, "w");

//       float mode_yields [max_modes ][5]={{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0}};
        // initialize yield numbers
        float mode_yields[max_modes-1][5];
        for(int mode_count = 0; mode_count < max_modes-1; mode_count++) //go through modes
                for(int yield=1; yield<=5; yield++) mode_yields[mode_count][yield-1]=0;

        for(int mode_count = 0; mode_count < max_modes-1; mode_count++) //go through modes
        {
                int mode = atoi(argv[mode_count+1]);
                char mode_name[20];
                if(mode == 0) {sprintf(mode_name, "CON");}
                if(mode == 1) {sprintf(mode_name, "NWF_DWB");}
                if(mode == 2) {sprintf(mode_name, "NWF_NWB");}
                if(mode == 3) {sprintf(mode_name, "NWF_PWB");}
                if(mode == 4) {sprintf(mode_name, "DWB");}
                if(mode == 5) {sprintf(mode_name, "VDD");}
                if(mode == 6) {sprintf(mode_name, "NWF_PWB_VDD");}
                if(mode == 7) {sprintf(mode_name, "PWB_VDD");}
                if(mode == 8) {sprintf(mode_name, "NWF_NWB_VDD");}
                if(mode == 9) {sprintf(mode_name, "NWB_VDD");}
                if(mode == 10) {sprintf(mode_name, "NWF");}
                if(mode == 11) {sprintf(mode_name, "NWF_VDD");}

                for(int yield=1; yield <= 5; yield++)
                {
                        int missing_chips= 0;
                        for(int chip=1; chip <=100; chip++)
                        {
                                ind temp_ind;

                                sprintf(temp_filename, "m%d/%i.best.%s.y%d.txt", mode, chip, mode_name, yield);
                                FILE* archive_file = fopen(temp_filename, "r");
```

```
                                    if(! archive_file )
                                    {
70                                          printf("\"%s\" not found.\n", temp_filename);
                                            missing_chips++;
                                            continue;
                                    }

75                                  if(fscanf( archive_file , "%s", buffer) == EOF) //chip_num
                                    {
                                            printf("Chip %s not done\n", temp_filename);
                                            missing_chips++;
                                            continue;
80                                  }
                                    temp_ind.chip_num=atoi(buffer);

                                    fscanf( archive_file , "%s", buffer); // nwell_floats  ( trash )

85                                  fscanf( archive_file , "%s", buffer); // pwell_floats  ( trash )

                                    fscanf( archive_file , "%s", buffer); //freq
                                    temp_ind.delay = 1/atof(buffer);

90                                  fscanf( archive_file , "%s", buffer); //power
                                    temp_ind.power = atof(buffer);

                                    if(temp_ind.delay < 0.5 || temp_ind.delay > 2.0 || temp_ind.power < 0.5 ||temp_ind.delay > 2.0)
                                    {
95                                          printf("Bad objectives in %s, Nd=%f Np=%f\n", temp_filename, temp_ind.delay, temp_ind.power);
                                            missing_chips++;
                                            continue;
                                    }

100                                 fscanf( archive_file , "%s", buffer); //  fitness
                                    temp_ind.fitness = atof(buffer);

                                    fscanf( archive_file , "%s", buffer); // pwell  bias
                                    temp_ind.pwell_bias=atof(buffer);
105
                                    fscanf( archive_file , "%s", buffer); // nwell  bias
                                    temp_ind.nwell_bias=atof(buffer);

                                    fscanf( archive_file , "%s", buffer); // vdd
110                                 temp_ind.vdd = atof(buffer);

                                    fscanf( archive_file , "%s", buffer); // nwell  chromo
                                    for(int i=0; i < 32; i++)
                                    {
115                                         if(buffer[i] == '0') temp_ind.genes[i]= 0;
                                            else if(buffer[i]== '1') temp_ind.genes[i]= 1;
                                            else {printf("Translation failed at chip_num %i, yield %i, gene %i\n", temp_ind.chip_num, yield, i);
                                                    exit(1);}
                                    }

120                                 fscanf( archive_file , "%s", buffer); // pwell  chromo ( trash )
                                    fclose( archive_file );

                                    if(temp_ind.fitness >= 0.0)
                                    {
125                                         mode_yields[mode_count][yield-1]++;
                                    }
                            } //end chip   iteration
                            if(100-missing_chips == 0) mode_yields[mode_count][yield-1] = 0;
                            else mode_yields[mode_count][yield-1]=(ceil(mode_yields[mode_count][yield-1]))/(100-missing_chips)*100;
130
                    } //end yield   iteration
                    fprintf( yields_file , "%s ", mode_name);
                    printf("%s ", mode_name);
            }//end mode  iteration
135
            fprintf( yields_file , "\n");
            printf("\n");

            for(int yield=1; yield<=5; yield++)
140         {
                    for(int mode_count = 0; mode_count < max_modes-1; mode_count++) //go through modes
                    {
                            fprintf( yields_file , "%f ", mode_yields[mode_count][yield-1]);
                            printf("%f ", mode_yields[mode_count][yield-1]);
145                 }
                    fprintf( yields_file , "\n");
                    printf("\n");
            }
            fclose( yields_file );
150 }
```

Appendix D

moIWABB analysis tools source code

Main C++ code

```
   #include <stdio.h>
   #include <fstream>
   #include <stdlib.h>
   #include <math.h>
 5 //#include < cstring >

   typedef struct ind_st /* an individual */
   {
       int chip_num;
10     float delay, power, pwell_bias, nwell_bias, vdd;
       int genes[32];
   } ind;


15 main(int argc, char** argv)
   {

           if(argc < 2)
           {
20                 printf("This program determines the yield from spea2 data\n");
                   printf("usage: command power_max m (m m...)\n");
                   printf("where m are the modes to read\n");
                   printf("inputs: m#/#.archive.txt\n");
                   printf("outputs: ./spea2_yields.txt\n");
25                 exit(1);
           }

           float dlimit[5]={1/.98,1/1.0,1/1.01,1/1.02,1/1.03};
           float plimit = atof(argv[1]);
30         // float plimit =1.05;

           int max_modes = argc-2;
           float mode_yields[max_modes][5];
           for(int mode_count = 0; mode_count < max_modes; mode_count++)
35         {
                   for(int yield=1; yield <=5; yield++) mode_yields[mode_count][yield-1]=0;
           }


40         ind mem[30];

           char temp_filename[256];
           char buffer[256];

45         sprintf(temp_filename, "spea2_yield.txt");
           FILE* yields_file = fopen(temp_filename, "w");

           for(int mode_count = 0; mode_count < max_modes; mode_count++)
           {
50                 int mode = atoi(argv[mode_count+2]);
                   int missing_chips= 0;

                   for(int chip_num = 1; chip_num <=100; chip_num++)
                   {
55
                           int best_mem= -1;
                           sprintf(temp_filename, "m%d/%d.archive.txt", mode, chip_num);
                           FILE* archive_file = fopen(temp_filename, "r");
                           if(! archive_file )
60                         {
                                   printf("%s not found.\n", temp_filename);
                                   missing_chips++;
                                   continue;
                           }
65
                           fscanf( archive_file , "%s", buffer); // generation nnumber ( trash )
                           if(atoi(buffer)<=5)
                           {
                                   printf("%s not done.\n", temp_filename);
```

```
70                  missing_chips++;
                    continue;
                }

                for(int ind_num=0; ind_num<30; ind_num++)
75              {
                    fscanf( archive_file , "%s", buffer); // delay
                    mem[ind_num].delay=atof(buffer);
                    fscanf( archive_file , "%s", buffer); //Power
                    mem[ind_num].power=atof(buffer);
80
                    if(mem[ind_num].power < .5 || mem[ind_num].power > 2.0 || mem[ind_num].delay < .5 || mem[ind_num].delay
                        > 2.0)
                    {
                        // printf ("%s ind %i was discarded . d=%f P=%f\n", temp_filename , ind_num , mem[ind_num]. delay , mem[
                            ind_num].power);
                        fscanf( archive_file , "%s", buffer); // pwellbias
85                      fscanf( archive_file , "%s", buffer); // nwellbias
                        fscanf( archive_file , "%s", buffer); //vdd
                        fscanf( archive_file , "%s", buffer); //genes
                        continue;
                    }
90
                    fscanf( archive_file , "%s", buffer); // pwellbias
                    mem[ind_num].pwell_bias=atof(buffer);
                    fscanf( archive_file , "%s", buffer); // nwellbias
                    mem[ind_num].nwell_bias=atof(buffer);
95                  fscanf( archive_file , "%s", buffer); //vdd
                    mem[ind_num].vdd=atof(buffer);
                    fscanf( archive_file , "%s", buffer); //genes
                    for(int i=0; i < 32; i++)
                    {
100                     if(buffer[i] == '0') mem[ind_num].genes[i]= 0;
                        else if(buffer[i]== '1') mem[ind_num].genes[i]= 1;
                        else {printf("Translation failed at chip %i, ind %i, gene %i\n", chip_num, ind_num, i); exit(1);}
                    }

105                 //find fastest member that still meets power limit
                    if(mem[ind_num].power < plimit && best_mem == -1) best_mem = ind_num;
                    if(mem[ind_num].power < plimit && best_mem != -1) {
                        if (mem[ind_num].delay < mem[best_mem].delay)
                        {
110                         best_mem = ind_num;
                            // printf (" chip %i, best_mem=%i\n", chip_num, best_mem);
                        }
                    }
                }
115             fclose ( archive_file );

                //Sum bin yields
                for(int yield=1; yield <=5; yield++)
                {
120                 if(best_mem == -1) mode_yields[mode_count][yield-1] = mode_yields[mode_count][yield-1];
                    else if(mem[best_mem].power < plimit && mem[best_mem].delay <= dlimit[yield-1])
                        mode_yields[mode_count][yield-1]++;
                }
            } //end chips iteration
125
            for(int yield=1; yield <=5; yield++)
            {
                if(100-missing_chips == 0) mode_yields[mode_count][yield-1] = 0;
                else mode_yields[mode_count][yield-1] = ceil(mode_yields[mode_count][yield-1])/(100-missing_chips)*100;
130         }

        } // end modes iteration

        for(int mode_count = 0; mode_count < max_modes; mode_count++)
135     {
            fprintf( yields_file , "m%i ", atoi(argv[mode_count+2]));
            printf("m%i ", atoi(argv[mode_count+2]));
        }
        fprintf( yields_file , "\n");
140     printf("\n");
        for(int yield=1; yield <=5; yield++)
        {
            for(int mode_count = 0; mode_count < max_modes; mode_count++)
            {
145             fprintf( yields_file , "%f ", mode_yields[mode_count][yield-1]);
                printf("%f ", mode_yields[mode_count][yield-1]);
            }
            fprintf( yields_file , "\n");
            printf("\n");
150     }

}
```

Mathematica code

```
Setup
Needs["LinearAlgebra`MatrixManipulation`"];
<<Statistics`DescriptiveStatistics`
Needs["LinearAlgebra`Cholesky`"];
5 Off[General::spell];
Off[General::"spell1"];
On[General::stop];
Needs["Graphics`Graphics`"];
Needs["Graphics`Colors`"];
10 Needs["Graphics`MultipleListPlot`"]
Needs["Graphics`Legend`"];
Needs["Statistics`NonlinearFit`"]
Needs["NumericalMath`InterpolateRoot`"]
READSPEA[]:=Do[k=1;
15      croppeddata={};
        numberofcolumns=6;
        datafromfile=.;
```

```
        While[k<=100,
                filename=StringJoin[ToString[k],".archive.txt"];
20              archstream = OpenRead[filename];
                If[archstream ==$Failed,k++;Continue[];];
                Skip[archstream, Number, 1];
                datafromfile=ReadList[archstream,Table[Number,{numberofcolumns}]];
                Close[archstream];
25              If[Length[datafromfile]>2 ,
                        AppendTo[croppeddata,DeleteCases[Table[

                If[datafromfile[[i,1]] >.2 && datafromfile[[i,1]] < 5.0 &&
                    datafromfile[[i,2]] >.2 && datafromfile[[i,2]] <5.0,
30                              {datafromfile[[i,1]], datafromfile[[i,2]]},{5,0}],
                                {i,1,Length[datafromfile]}],{5,0}]];
                        ];
                If[k==100,Return[croppeddata]];
                k++
35      ],
        {1}
    ]
Directory
SetDirectory[
40    "/Users/jlgregg/Desktop/Other\ HP\ \
 circuits/MOadder32_V5/results_spea2i/res30/m9"];
SPEA2
speadata1=Union[READSPEA[]];
speadata1plot=
45  MultipleListPlot[speadata1,PlotRange->{.9,1.02},
    SymbolShape->
      Table[PlotSymbol[Box,.3,Filled->False],{Length[speadata1]}],
    SymbolStyle->Table[Blue,{Length[speadata1]}]]
Clean the spea data to remove duplicate x values
50 For[chip=1,chip<=Length[speadata1],chip++,
  For[ind=1,ind<=Length[speadata1[[chip]]],ind++,x1=speadata1[[chip,ind,1]];
      For[ind2=ind,ind2<=Length[speadata1[[chip]]],ind2++,
        If[x1==speadata1[[chip,ind2,1]],
            speadata1[[chip,ind2,1]]+=.000001*x1
55          ];
        ];
      ];
  ]
chip=.;ind=.;ind2=.;x1=.
60 intfitplots =.;
xvarindata=
    Table[Table[
        speadata1[[chip,ind,1]],{ind,1,Length[speadata1[[chip]]]}],{chip,1,
        Length[speadata1]}];
65 yvarindata=
    Table[Table[
        speadata1[[chip,ind,2]],{ind,1,Length[speadata1[[chip]]]}],{chip,1,
        Length[speadata1]}];
 intfitcurves =
70  Table[Interpolation[Union[speadata1[[k]]],
        InterpolationOrder -> 1],{k,1,Length[speadata1]}];
 intfitplots =
    Table[{Plot[
          intfitcurves[[k]][x],{x,Min[xvarindata[[k]]],Max[xvarindata[[k]]]},
75        PlotRange->{Min[yvarindata[[k]]],Max[yvarindata[[k]]]},
          AxesOrigin->{1,1},DisplayFunction->Identity],
        ListPlot[speadata1[[k]],
          PlotRange->{Min[yvarindata[[k]]],Max[yvarindata[[k]]]},
          PlotStyle->RGBColor[0,0,1],AxesOrigin->{1,1},
80        DisplayFunction->Identity]},{k,1,Length[speadata1]}];
SpeaData1AndInterpPlot=
  Show[intfitplots,PlotRange->{.92,1.2},
    DisplayFunction->$DisplayFunction]
maxdelay=1/1.03;
85 maxpower=1.0;
delaytable=.;
Table[FindRoot[
      Evaluate[intfitcurves[[k]][x]]==maxpower,{x,Min[xvarindata[[k]]],
        Max[xvarindata[[k]]]}],{k,1,Length[intfitcurves]}];
90 x/.%//Flatten;
delaytable=DeleteCases[%,x];
yield=0;
For[k=1,k<=Length[delaytable],
    If[delaytable[[k]]<maxdelay,yield++];
95  k++
    ];
Print["Good chips: ",yield]
Print["Chips under maxpower: ",Length[delaytable]]
Print["Chips in data: ", Length[intfitcurves]]
100 Print["Overall yield: ", N[yield/Length[intfitcurves]]*100]
Make the graph
Export["speaplot.pdf",
  Show[intfitplots,
    MultipleListPlot[
105   Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
      SymbolShape->
        MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
            Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
      DisplayFunction->Identity],PlotRange->{{.85,1.2},{.9,1.2}},
110 Frame->True,Axes->False,DisplayFunction->$DisplayFunction]]
Testing the FindRoot function
For[k=1,k<=Length[intfitcurves],k++,
  Print[k,
    FindRoot[Evaluate[intfitcurves[[k]][x]]==maxpower,{x,
115     Min[xvarindata[[k]]],Max[xvarindata[[k]]]}]]
  ]
k=81;
speadata1[[k]]
Interpolation[Union[speadata1[[k]]],InterpolationOrder->1]
120 k=.;
Compiling all the graphs
Make m2 graph for compilations
newdelaytable={};
For[k=1,k<=Length[delaytable],k++,
```

```
125       If[delaytable[[k]]>=.85 && delaytable[[k]]<=1.2,
            AppendTo[newdelaytable,delaytable[[k]]]
            ]
          ];
    k=.;
130 delaytable=newdelaytable;
    newdelaytable=.;
    m2plot=Show[intfitplots,
          MultipleListPlot[
            Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
135         SymbolShape->
              MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
                  Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
            DisplayFunction->Identity,PlotRange->{{.85,1.2},{.9,1.2}}],
          PlotRange->{{.85,1.2},{.9,1.2}},Frame->True,Axes->False,
140       FrameTicks->None,FrameStyle->AbsoluteThickness[1.0],
          GridLines->{{Automatic,RGBColor[0,0,0]},{Automatic,RGBColor[0,0,0]}},
          DisplayFunction->Identity]
    Make m4 graph for compilations
    newdelaytable={};
145 For[k=1,k<=Length[delaytable],k++,
          If[delaytable[[k]]>=.85 && delaytable[[k]]<=1.2,
            AppendTo[newdelaytable,delaytable[[k]]]
            ]
          ];
150 k=.;
    delaytable=newdelaytable;
    newdelaytable=.;

    m4plot=Show[intfitplots,
155       MultipleListPlot[
            Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
            SymbolShape->
              MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
                  Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
160         DisplayFunction->Identity,PlotRange->{{.85,1.2},{.9,1.2}}],
          PlotRange->{{.85,1.2},{.9,1.2}},Frame->True,Axes->False,
          FrameTicks->None,FrameStyle->AbsoluteThickness[1.0],
          GridLines->{{Automatic,RGBColor[0,0,0]},{Automatic,RGBColor[0,0,0]}},
          DisplayFunction->Identity]
165 Make m5 graph for compilations
    newdelaytable={};
    For[k=1,k<=Length[delaytable],k++,
          If[delaytable[[k]]>=.85 && delaytable[[k]]<=1.2,
            AppendTo[newdelaytable,delaytable[[k]]]
170         ]
          ];
    k=.;
    delaytable=newdelaytable;
    newdelaytable=.;
175
    m5plot=Show[intfitplots,
          MultipleListPlot[
            Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
            SymbolShape->
180           MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
                  Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
            DisplayFunction->Identity,PlotRange->{{.85,1.2},{.9,1.2}}],
          PlotRange->{{.85,1.2},{.9,1.2}},Frame->True,Axes->False,
          FrameTicks->None,FrameStyle->AbsoluteThickness[1.0],
185       GridLines->{{Automatic,RGBColor[0,0,0]},{Automatic,RGBColor[0,0,0]}},
          DisplayFunction->Identity]
    Make m8 graph for compilations
    newdelaytable={};
    For[k=1,k<=Length[delaytable],k++,
190       If[delaytable[[k]]>=.85 && delaytable[[k]]<=1.2,
            AppendTo[newdelaytable,delaytable[[k]]]
            ]
          ];
    k=.;
195 delaytable=newdelaytable;
    newdelaytable=.;

    m8plot=Show[intfitplots,
          MultipleListPlot[
200         Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
            SymbolShape->
              MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
                  Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
            DisplayFunction->Identity,PlotRange->{{.85,1.2},{.9,1.2}}],
205       PlotRange->{{.85,1.2},{.9,1.2}},Frame->True,Axes->False,
          FrameTicks->None,FrameStyle->AbsoluteThickness[1.0],
          GridLines->{{Automatic,RGBColor[0,0,0]},{Automatic,RGBColor[0,0,0]}},
          DisplayFunction->Identity]
    Make m9 graph for compilations
210 newdelaytable={};
    For[k=1,k<=Length[delaytable],k++,
          If[delaytable[[k]]>=.85 && delaytable[[k]]<=1.2,
            AppendTo[newdelaytable,delaytable[[k]]]
            ]
215       ];
    k=.;
    delaytable=newdelaytable;
    newdelaytable=.;

220 m9plot=Show[intfitplots,
          MultipleListPlot[
            Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
            SymbolShape->
              MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
225               Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
            DisplayFunction->Identity,PlotRange->{{.85,1.2},{.9,1.2}}],
          PlotRange->{{.85,1.2},{.9,1.2}},Frame->True,Axes->False,
          FrameTicks->None,FrameStyle->AbsoluteThickness[1.0],
          GridLines->{{Automatic,RGBColor[0,0,0]},{Automatic,RGBColor[0,0,0]}},
230       DisplayFunction->Identity]
    Make m10 graph for compilations
```

```
     newdelaytable={};
     For[k=1,k<=Length[delaytable],k++,
         If[delaytable[[k]]>=.85 && delaytable[[k]]<=1.2,
235          AppendTo[newdelaytable,delaytable[[k]]]
             ]
         ];
     k=.;
     delaytable=newdelaytable;
240  newdelaytable=.;

     m10plot=Show[intfitplots,
         MultipleListPlot[
             Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
245          SymbolShape->
                 MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
                         Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
             DisplayFunction->Identity,PlotRange->{{.85,1.2},{.9,1.2}}],
         PlotRange->{{.85,1.2},{.9,1.2}},Frame->True,Axes->False,
250      FrameTicks->None,FrameStyle->AbsoluteThickness[1.0],
         GridLines->{{Automatic,RGBColor[0,0,0]},{Automatic,RGBColor[0,0,0]}},
         DisplayFunction->Identity]
     Make m11 graph for compilations
     newdelaytable={};
255  For[k=1,k<=Length[delaytable],k++,
         If[delaytable[[k]]>=.85 && delaytable[[k]]<=1.2,
             AppendTo[newdelaytable,delaytable[[k]]]
             ]
         ];
260  k=.;
     delaytable=newdelaytable;
     newdelaytable=.;

     m11plot=Show[intfitplots,
265      MultipleListPlot[
             Table[{delaytable[[k]], maxpower},{k,1,Length[delaytable]}],
             SymbolShape->
                 MakeSymbol[{AbsoluteThickness[1.5],RGBColor[0,1,0],
                         Line[{{0,2},{0,-2}}],Line[{{-2,0},{2,0}}]}],
270          DisplayFunction->Identity,PlotRange->{{.85,1.2},{.9,1.2}}],
         PlotRange->{{.85,1.2},{.9,1.2}},Frame->True,Axes->False,
         FrameTicks->None,FrameStyle->AbsoluteThickness[1.0],
         GridLines->{{Automatic,RGBColor[0,0,0]},{Automatic,RGBColor[0,0,0]}},
         DisplayFunction->Identity]
275  All together now
     Show[GraphicsArray[{
             {m8plot,m11plot},
             {m2plot,m10plot},
             {m9plot,m4plot},
280          {{},m5plot}}],
       DisplayFunction->$DisplayFunction,GraphicsSpacing->0]
     Export["~/Desktop/spearesults.pdf",Show[GraphicsArray[{
               {m8plot,m11plot},
               {m2plot,m10plot},
285            {m9plot,m4plot},
               {{},m5plot}}],
         DisplayFunction->$DisplayFunction,GraphicsSpacing->0]]
```