National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Optimizing Power Consumption, Area, and Delay in Behavioral Synthesis

by

Raul San Martin, M.A.Sc.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of

Doctor of Philosophy

Department of Electronics
Faculty of Engineering
Carleton University
March 1995

© copyright
1995, Raul San Martin

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

**THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.**

**L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.**

**THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.**

**L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.**

ISBN 0-612-02963-8

Canada

Name

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

0000

U·M·I

SUBJECT TERM

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
 Art History 0377
 Cinema 0900
 Dance 0378
 Fine Arts 0357
 Information Science 0723
 Journalism 0391
 Library Science 0399
 Mass Communications 0708
 Music 0413
 Speech Communication 0459
 Theater 0465

EDUCATION

General 0515
 Administration 0514
 Adult and Continuing 0516
 Agricultural 0517
 Art 0273
 Bilingual and Multicultural 0282
 Business 0688
 Community College 0275
 Curriculum and Instruction 0727
 Early Childhood 0518
 Elementary 0524
 Finance 0277
 Guidance and Counseling 0519
 Health 0680
 Higher 0745
 History of 0520
 Home Economics 0278
 Industrial 0521
 Language and Literature 0279
 Mathematics 0280
 Music 0522
 Philosophy of 0998
 Physical 0523

Psychology 0525
 Reading 0535
 Religious 0527
 Sciences 0714
 Secondary 0533
 Social Sciences 0534
 Sociology of 0340
 Special 0529
 Teacher Training 0530
 Technology 0710
 Tests and Measurements 0288
 Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
 General 0289
 Ancient 0290
 Linguistics 0291
 Modern 0291
 Literature 0401
 General 0294
 Classical 0295
 Comparative 0297
 Medieval 0298
 Modern 0316
 Africa 0591
 American 0305
 Asian 0352
 Canadian (English) 0355
 Canadian (French) 0593
 English 0311
 Germanic 0312
 Latin American 0315
 Middle Eastern 0313
 Romance 0314
 Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
 Religion 0318
 General 0321
 Biblical Studies 0319
 Clergy 0320
 History of 0322
 Philosophy of 0469
 Theology 0323

SOCIAL SCIENCES

American Studies 0323
 Anthropology 0324
 Archaeology 0326
 Cultural 0327
 Physical 0310
 Business Administration 0272
 General 0770
 Accounting 0454
 Banking 0338
 Management 0385
 Marketing 0501
 Canadian Studies 0503
 Economics 0505
 General 0508
 Agricultural 0509
 Commerce Business 0510
 Finance 0511
 History 0358
 Labor 0366
 Theory 0351
 Folklore 0578
 Geography 0366
 Gerontology 0351
 History 0578
 General 0578

Ancient 0579
 Medieval 0581
 Modern 0582
 Black 0328
 African 0331
 Asia: Australia and Oceania 0332
 Canadian 0334
 European 0335
 Latin American 0336
 Middle Eastern 0333
 United States 0337
 History of Science 0585
 Law 0398
 Political Science 0615
 General 0616
 International Law and Relations 0617
 Public Administration 0814
 Recreation 0452
 Social Work 0626
 Sociology 0627
 General 0938
 Criminology and Penology 0631
 Demography 0628
 Ethnic and Racial Studies 0629
 Individual and Family Studies 0630
 Industrial and Labor Relations 0700
 Public and Social Welfare 0344
 Social Structure and Development 0709
 Theory and Methods 0999
 Transportation 0453
 Urban and Regional Planning 0453
 Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
 General 0285
 Agronomy 0475
 Animal Culture and Nutrition 0476
 Animal Pathology 0359
 Food Science and Technology 0478
 Forestry and Wildlife 0479
 Plant Culture 0480
 Plant Pathology 0817
 Plant Physiology 0777
 Range Management 0746
 Wood Technology 0306
 Biology 0287
 General 0308
 Anatomy 0309
 Biostatistics 0379
 Botany 0329
 Cell 0353
 Ecology 0369
 Entomology 0793
 Genetics 0410
 Limnology 0307
 Microbiology 0317
 Molecular 0416
 Neuroscience 0433
 Oceanography 0821
 Physiology 0778
 Radiation 0472
 Veterinary Science 0786
 Zoology 0760
 Biophysics 0786
 General 0760
 Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
 Geochemistry 0996

Geodesy 0370
 Geology 0372
 Geophysics 0373
 Hydrology 0388
 Mineralogy 0411
 Paleobotany 0345
 Paleocology 0426
 Paleontology 0418
 Paleozoology 0985
 Palynology 0427
 Physical Geography 0368
 Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
 Health Sciences 0566
 General 0300
 Audiology 0992
 Chemotherapy 0567
 Dentistry 0350
 Education 0769
 Hospital Management 0758
 Human Development 0982
 Immunology 0564
 Medicine and Surgery 0347
 Mental Health 0569
 Nursing 0570
 Nutrition 0380
 Obstetrics and Gynecology 0354
 Occupational Health and Therapy 0381
 Ophthalmology 0571
 Pathology 0419
 Pharmacology 0572
 Pharmacy 0382
 Physical Therapy 0573
 Public Health 0574
 Radiology 0575
 Recreation 0575

Speech Pathology 0460
 Toxicology 0383
 Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
 Chemistry 0485
 General 0749
 Agricultural 0486
 Analytical 0487
 Biochemistry 0488
 Inorganic 0738
 Nuclear 0490
 Organic 0491
 Pharmaceutical 0494
 Physical 0495
 Polymer 0754
 Radiation 0405
 Mathematics 0605
 Physics 0986
 General 0606
 Acoustics 0608
 Astronomy and Astrophysics 0748
 Atmospheric Science 0607
 Atomic 0798
 Electronics and Electricity 0759
 Elementary Particles and High Energy 0609
 Fluid and Plasma 0610
 Molecular 0752
 Nuclear 0756
 Optics 0611
 Radiation 0463
 Solid State 0346
 Statistics 0984
 Applied Sciences
 Applied Mechanics 0346
 Computer Science 0984

Engineering 0537
 General 0538
 Aerospace 0539
 Agricultural 0540
 Automotive 0541
 Biomedical 0542
 Chemical 0543
 Civil 0544
 Electronics and Electrical 0348
 Heat and Thermodynamics 0545
 Hydraulic 0546
 Industrial 0547
 Marine 0794
 Materials Science 0548
 Mechanical 0743
 Metallurgy 0551
 Mining 0552
 Nuclear 0549
 Packaging 0765
 Petroleum 0554
 Sanitary and Municipal 0790
 System Science 0428
 Geotechnology 0796
 Operations Research 0795
 Plastics Technology 0994
 Textile Technology 0994

PSYCHOLOGY

General 0621
 Behavioral 0384
 Clinical 0622
 Developmental 0620
 Experimental 0623
 Industrial 0624
 Personality 0625
 Physiological 0989
 Psychobiology 0349
 Psychometrics 0632
 Social 0451

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

**OPTIMIZING POWER CONSUMPTION, AREA AND DELAY IN
BEHAVIORAL SYNTHESIS**

submitted by Raul San Martin
in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

Chair, Department of Electronics

Thesis Supervisor

External Examiner

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Faculty of Engineering
Carleton University
March 1995

Abstract

The recent impressive reductions of integrated circuit's (IC) feature sizes, coupled with the continuous increase of the number of components and clock frequencies, clearly indicate the need to move the IC design to higher levels of abstraction, where it is more easily understood and managed. This thesis presents a new methodology for the optimization of power consumption, area, and delay in the behavioral synthesis of ASICs.

Two tools were derived from this methodology: PASSOS and Power-Profiler. Both tools perform global simultaneous scheduling and assignment, by selecting the best hardware module for each operation of a data flow graph, and allocating the operations to precise time slots. PASSOS mixes and matches the best digit-serial arithmetic operators while Power-Profiler chooses from any set of adders and multipliers defined in a library. Both tools use genetic algorithms to perform the optimizations.

The approach described in this thesis can optimize any combination of area, average power, and peak power, subject to any side constraints on area, power, and delay. The peak power consumption problem is tackled through the use of *power profiles*, which are power distribution graphs introduced to show how a component's energy is used on each clock cycle. Furthermore, the approach allows the best supply voltages to be chosen for each timing constraint (including mixed-voltage solutions).

The methodology is applied to well-known examples, and also to very large and complex problems. The results show that the methodology can be effectively and efficiently used to significantly reduce transistor counts, average power consumption, and peak power consumption in the behavioral synthesis of ASICs.

Acknowledgments

I estimate that during the course of this Ph.D. program I have logged tens of thousands hours of CPU time (some of us work in parallel...). This was achieved by "taking over" the department's network at night (to the horror of the owl-students). I am grateful to the Department of Electronics for providing such advanced resources.

Special thanks to my friend Sergio Golombek for the numerous technical discussions, tips and suggestions. They were particularly appreciated coming from someone who designs the real things.

I would like to thank my parents for raising me in an environment conducive to learning, and for setting the example and encouraging the high standards that made this thesis possible.

I am grateful to my wife Fernanda for her continuous support, understanding and patience, for the tennis, cross-country skiing, games, etc., and also for her technical help, and many more little and not so little things, but most of all, for her love and company.

Finally, this thesis would absolutely not have been possible without a person that deservedly receives many teaching excellence awards for his outstanding dedication to teaching and students. John Knight, my supervisor, has always been available for help, always encouraging, seeing the bright side of problems with his encyclopedic knowledge. For the funding, for sending me to interesting conferences, for the technical discussions, for the encouragement, and too many other valuable things, he receives my "supervising excellence" award.

Thank you.

A mis padres, Amelia y Osvaldo

Para Fernanda

TABLE OF CONTENTS

List of Figures	xi
List of Tables	xv
I Introduction	1
1.1 The Need for Behavioral Synthesis	1
1.2 Advantages of Behavioral Synthesis	4
1.3 Thesis Contributions	5
1.3.1 Simultaneous Scheduling and Assignment	7
1.3.2 Digit-Serial Implementations at the Behavioral Level	7
1.3.3 Average Power Consumption Optimization at the Behavioral Level	7
1.3.4 Power Profiles and Peak Power Consumption	7
1.3.5 Supply Voltage Choice at the Behavioral Level	7
1.3.6 Multiobjective, Multiply Constrained Behavioral Optimization	7
1.3.7 Real Parameter Values	8
1.3.8 Larger Applications	8
1.4 Thesis Organization	9
II Behavioral Synthesis	10
2.1 The Assignment Problem	10
2.2 The Scheduling Problem	14
2.3 Joint Assignment and Scheduling	15
2.4 Previous Work	17
III Power Consumption in Behavioral Synthesis and Power Profiles	21
3.1 Why Low Power?	21
3.1.1 Shut Down of Operators	23
3.1.2 Low Supply Voltage	23
3.1.3 Mixed-Voltage Circuits	25

3.1.4	Use of Parallel Operators Only on Time-Critical Data Paths	25
3.1.5	Use Digit-Serial Operators Where Speed is Less Important	26
3.2	Power in Behavioral Synthesis	26
3.2.1	Previous Work	28
3.3	Power Profiles	29
3.4	Module Selection (Assignment) and Average Power	39
3.5	Scheduling, Assignment and Peak Power	43
3.6	Should Silicon Area be Considered?	46
3.7	Defeating the Power Profiles	48
3.8	Concluding Remarks	54
IV	Hardware	55
4.1	Digit-Serial Operations	55
4.1.1	Transistor Counts, Time Delays and Power Consumption	61
4.2	Multiplication and Multipliers	65
4.2.1	Booth Algorithm	65
4.2.2	Bit-Pair Recoding	67
4.2.3	The Quasi-Serial Multiplier	70
4.2.4	Modular Array Multiplication	70
4.2.4.1	A Multiply Module	73
4.2.4.2	Wallace trees	73
4.3	Delays, Transistor Counts and Average Power Consumption	76
V	Genetic Algorithms	79
5.1	Biology and History of Genetic Algorithms	80
5.2	Basics	81
5.2.1	Reproduction	83
5.2.2	Crossover	85
5.2.3	Mutation	87
5.3	Reproduction: Steady State or Generational GA?	87
5.4	Types of Crossover	90

5.5	Fitness Techniques	92
5.6	The Schema Theorem: GA Theory	94
5.7	Parallelization of GAs	95
5.8	Hybrid Genetic Algorithms	96
5.9	Programmability	97
5.10	Another Example	97
5.11	About Genetic Programming	101
VI	Implementation	105
6.1	Encoding	105
6.1.1	Non-Shared Problems	107
6.1.2	Shared Problems	110
6.2	Parameter Setting	114
6.2.1	PC: Probability of Crossover	115
6.2.2	Probability of Mutation	115
6.2.3	Number of Generations	118
6.2.4	Population Size	119
6.3	Initial Populations	119
6.3.1	Extended Initial Populations	119
6.3.2	Population Seeding	122
6.4	Pruning	124
6.5	Steady-State Reproduction	124
6.6	Analysis	124
6.7	Algorithm	128
6.7.1	Where Does Crossover Occur?	129
6.8	Objective Function	129
6.8.1	Calculating Transistor Counts	130
6.8.2	Calculating Average Power	132
6.8.3	Calculating Peak Power	132
6.8.4	Calculating Delay	132
6.8.5	Multiobjective Optimization	132
6.8.6	Penalty Functions	133

6.9	Tools: PASSOS and Power-Profiler	136
6.10	Software Structure	136
6.11	Input Files	138
6.12	Batch Commands	138
6.13	The World-Wide-Web Interface	139
VII	Data Flow Graphs and Digit-Serial Applications	142
7.1	Data Flow Graphs	142
7.2	Assumptions	146
7.3	Results	147
7.4	Analysis	152
VIII	Architectural-Choice Applications	156
8.1	Applications	156
8.2	Single Module Behavioral Synthesis	157
8.3	Different Supply Voltages	157
8.4	Different Power Optimization Goals	160
8.5	Overall Peak Power Profiles	160
8.6	Mixing Supply Voltages	163
8.7	Considering Silicon Area and Supply Voltages	165
8.8	Pareto Fronts	171
8.9	Optimizing Delay With a Power Constraint	173
8.10	CPU Times	174
8.11	Analysis	175
IX	Larger Applications	179
9.1	Unrolling Data Flow Graphs	179
9.2	Quality and Efficiency of Solutions	182
9.3	Some Examples	183
9.4	Analysis	186

X	Conclusion	188
10.1	Contributions	188
10.2	Limitations	190
10.2.1	Limited Modules and Architectures	190
10.2.2	Effects of Clock Frequency	191
10.2.3	Other Circuit Components	191
10.2.4	Other Behavioral Functions	191
10.2.5	Digit-Serial Implementations	192
10.2.6	Lower Supply Voltages	192
10.2.7	Power Consumption	192
10.3	Summary of Important Results and Conclusions	193
10.4	Future Work	195
	Bibliography	197
Appendix A	The World Wide Web Interface	205

List of Figures

FIGURE 1	Optimization issues in the circuit synthesis process.	3
FIGURE 2	Exploitation of parallelism in behavioral synthesis.	6
FIGURE 3	A data flow graph with six additions.	11
FIGURE 4	Non-shared timing diagrams with $T_c = 60$ ns.	12
FIGURE 5	Non-shared timing diagram with a $T_c = 100$ ns.	13
FIGURE 6	A shared implementation with a $T_c = 60$ ns.	15
FIGURE 7	Schedules with: (a) $T_c = 80$ ns; (b) $T_c = 120$ ns.	16
FIGURE 8	Assignment AND scheduling with: (a) $T_c = 80$ ns; (b) $T_c = 120$ ns.	18
FIGURE 9	Instantaneous power supply of a 32-bit CLA.	24
FIGURE 10	Speed and power relative performance of RCA and CLA.	25
FIGURE 11	Behavioral implementation example.	27
FIGURE 12	Inputs to adders and multipliers.	31
FIGURE 13	Energy consumption and delays per 32-bit operation.	32
FIGURE 14	Icc current on the ARR16 multiplier.	33
FIGURE 15	Instantaneous power of RCA 5V.	34
FIGURE 16	Instantaneous power of the CLA adder.	35
FIGURE 17	Instantaneous power of the Booth multiplier.	36
FIGURE 18	Instantaneous power of the QBS multiplier.	37
FIGURE 19	Power profiles: RCA32 and CLA32.	38
FIGURE 20	Power Profiles of the ARR16, ARR32 and QBS32 multipliers.	40
FIGURE 21	Effects of schedules C, D, and E on peak power.	44
FIGURE 22	Effects of schedules F, G, and H on peak power.	45
FIGURE 23	Performance of schedules C, D, E, F, G, and H.	48
FIGURE 24	The effects of supply voltages on total delay.	49
FIGURE 25	Reducing delay at 3.3V with schedules A2 and B.	50
FIGURE 26	Power profiles of ripple/no ripple conditions.	51

FIGURE 27	Peak power with different number of inputs changing.	53
FIGURE 28	1-bit adder block diagrams.	55
FIGURE 29	A two-bit addition.	57
FIGURE 30	Two-bit addition using a one-bit serial adder.	57
FIGURE 31	Implementations of a four-bit addition.	60
FIGURE 32	Qualitative representation of the timing/area relationship.	62
FIGURE 33	Plot of α , β , and η for a 32-bit addition.	63
FIGURE 34	Example of bit-pair recoding.	68
FIGURE 35	Basic hardware for bit-pair recoding multiplication.	69
FIGURE 36	The Quasi-Serial Multiplier block diagram.	71
FIGURE 37	Modules for eight-bit operands multiplication.	72
FIGURE 38	Paper-and-pencil multiplication of AH by BH	73
FIGURE 39	A 4 x 4 NMN (Non-additive multiplier module).	74
FIGURE 40	Wallace trees: (a) W3; (b) W5; (c) W7.	75
FIGURE 41	Arrangement of 4x4 modules for an eight-bit multiplication.	76
FIGURE 42	Arrangements for 4 x 4, 8 x 8, 16 x 16, and 32 x 32 multipliers.	77
FIGURE 43	Plot of function $f(x)$	82
FIGURE 44	Roulette wheels for initial population.	84
FIGURE 45	Generational vs. Steady-State GAs.	89
FIGURE 46	One-point crossover.	90
FIGURE 47	Two-point crossover.	91
FIGURE 48	Uniform crossover.	92
FIGURE 49	Examples of schemata (<i>building blocks</i>) in a chromosome.	94
FIGURE 50	Plot of a double sinc.	98
FIGURE 51	Initial and last populations of double sinc example.	100
FIGURE 52	Evolution of average and maximum fitness values.	101
FIGURE 53	Example of crossover in genetic programming.	104
FIGURE 54	A data flow graph and a typical chromosome.	106
FIGURE 55	Non-shared problem chromosome and representation.	108

FIGURE 56	Table descriptors with an example.	109
FIGURE 57	Encoding in shared problems.	111
FIGURE 58	Scheduling: Example of relative slots.	112
FIGURE 59	Effects of probability of crossover.	116
FIGURE 60	Effects of probability of mutation.	117
FIGURE 61	Effects of number of generations.	118
FIGURE 62	Effects of population size.	120
FIGURE 63	Algorithm for extended initial population.	121
FIGURE 64	Effects of extended initial populations.	121
FIGURE 65	Single and multiple seeds bidimensional analogy.	123
FIGURE 66	Results using steady-state reproduction.	125
FIGURE 67	List of followers and list of paths.	129
FIGURE 68	Algorithms for generational and steady-state GAs.	130
FIGURE 69	Scheduling and peak power profile of an example.	131
FIGURE 71	Area objective function with penalty functions.	134
FIGURE 70	Some typical penalty functions.	135
FIGURE 72	Data flow in PASSOS and Power-Profiler.	137
FIGURE 73	Sample input file.	139
FIGURE 74	The World Wide Web GUI.	141
FIGURE 75	The EWF data flow graph.	143
FIGURE 76	The 16-point FIR filter data flow graph.	144
FIGURE 77	The DCT data flow graph.	145
FIGURE 78	Area and power results of the EWF.	149
FIGURE 79	Results of the DCT example.	150
FIGURE 80	Results of the INU5 example.	151
FIGURE 81	Area and power with single digit operators (EWF).	153
FIGURE 82	Delay minimization with area and power constraints (FIR).	154
FIGURE 83	Area and power savings of EWF, DCT, and INU5.	155
FIGURE 84	Comparison with single module approach.	158

FIGURE 85	Average power of EWF and FIR at 3.3V and 5V.	159
FIGURE 86	Effects of different optimization goals.	161
FIGURE 87	Peak power profiles for a delay of 1,000 ns.	162
FIGURE 88	Results of triple optimization (area/average power/peak power).	163
FIGURE 89	Mixing supply voltages.	164
FIGURE 90	Schedule of the FIR example with a supply voltage of 3.3V.	166
FIGURE 91	Schedule of the FIR example with a mixed supply voltage.	167
FIGURE 92	Considering transistor counts and supply voltages.	168
FIGURE 93	Power optimizations considering area.	169
FIGURE 94	Schedule of the combined area-power optimization.	170
FIGURE 95	The area-average Power Pareto fronts.	172
FIGURE 96	Delay optimization with power constraints.	173
FIGURE 97	The FIR example using only Booth multipliers and CLA adders.	176
FIGURE 98	Schedules of the FIR example for area and peak power optimization.	178
FIGURE 99	INU2: The Elliptic Wave Filter unrolled once.	180
FIGURE 100	Three-stage (INU3) and five-stage (INU5) EWF.	181
FIGURE 101	INU3 power profiles.	184
FIGURE 102	INU5 power profiles.	185
FIGURE 103	Average power as a function of delay.	187
FIGURE 104	GUI for custom data flow graph definition.	206
FIGURE 105	DFG definition corresponding to figure 2.	207

List of Tables

TABLE 1	A set of adders with a varied range of delays and transistor counts.	10
TABLE 2	Comparison between implementations.	17
TABLE 3	Delay and average power of adders and multipliers.	30
TABLE 4	Effects of assignment on average power and energy consumption (5V). ...	41
TABLE 5	Delay normalized to 1,320 ns.	42
TABLE 6	Delays and peak power of several schedules.	46
TABLE 7	Transistor counts of adders and multipliers.	47
TABLE 8	Power comparison among several cases (RCA).	52
TABLE 9	Areas and delays for a two-bit addition.	58
TABLE 10	Delay and area of the four-bit addition implementations.	59
TABLE 11	Delay and area for implementations of 32-bit addition.	61
TABLE 12	Addition: Transistor counts, delays and power consumption.	62
TABLE 13	Good and bad cases of Booth's technique.	67
TABLE 14	Bit-pair recoding table.	68
TABLE 15	Splitting multiplier and multiplicand into 2 numbers each.	72
TABLE 16	Characteristics of digit-serial operators (5V).	77
TABLE 17	Initial population and fitness values.	83
TABLE 18	The mating pool.	85
TABLE 19	Mates, crossover sites and generation 1.	86
TABLE 20	Fitness techniques.	93
TABLE 21	Operator types for digit-serial choice.	107
TABLE 22	Architectural choices (mixed-voltage).	107
TABLE 23	Scheduling information of an example chromosome.	111
TABLE 24	Minima, maxima, number of slots and bits needed.	114
TABLE 25	Effects of extended initialization buffer sizes.	122
TABLE 26	Results with fixed CPU times (30 seconds and 90 seconds).	126

TABLE 27	Effects of large constants on fitness proportions.	136
TABLE 28	Characteristics of typical applications.	142
TABLE 29	Best solutions in the literature.	147
TABLE 30	Typical results: assignments of FIR, EWF, DCT.	156
TABLE 31	Assignments for the FIR example with a delay of 400 ns.	165
TABLE 32	CPU times (seconds) on a SparcStation 5.	174
TABLE 33	Average cpu times for all timing constraints (minutes).	174
TABLE 34	Characteristics of unrolled filters.	182
TABLE 35	Results for relaxed timing constraint (5V).	182
TABLE 36	Results for tight timing constraint (5V).	182

I Introduction

This thesis deals with the optimization of silicon area, power consumption and delay in the behavioral synthesis of application-specific integrated circuits (ASICs).

1.1 The Need for Behavioral Synthesis

While current state-of-the-art integrated circuits (ICs) contain millions of components, the number of components will most likely increase geometrically in the future. As with every increase in the number of components comes an increase in chip complexity, there is a clear need to move the design to higher levels of abstraction, where its functionality is much easier to understand. Tools that work at higher levels should simply accept a behavioral description and generate an IC that meets all functional and performance requirements. Such tools take care of the millions of design details in a fast, automatic, and error-free manner. They can potentially generate excellent designs by taking advantage of particular algorithmic characteristics of applications (e.g., system parallelism).

Consequently, the behavioral and/or high-level synthesis market has a very large expected commercial market value [1]. Nevertheless, the field still faces enormous challenges due to the very complex nature of the synthesis optimization problems involved [2]. These problems usually include the generation of area, power, and speed efficient circuits. The efficiency achieved determines the degree of acceptance of the synthesis methodologies and tools by design engineers. As integrated circuits are everyday becoming larger and more complicated to design, it is very difficult to automatically generate a circuit which is near optimal in silicon area and/or power consumption, and that at the same time is fast enough to meet demanding performance requirements. The technical difficulties are so many that a very efficient,

complete, and general purpose circuit synthesizer, or *silicon compiler*, is still many years ahead.

Behavioral synthesis may be defined as the process of automatically generating a representation of an integrated circuit from a behavioral or functional specification [3]. As such, it is responsible for the translation from one level of design (generally, an algorithmic specification of the behavior of the circuit) into another (generally, a structural level or Register-Transfer level).

Design levels can be as abstract as product specifications or as concrete as layout mask geometries. As the name implies, high-level synthesis corresponds to the higher stages of the typical IC design process, represented by the chart on figure 1. This figure shows every design stage in chronological order. The design starts with a behavioral specification and ends with the layout representation. At the right side of each stage the most common optimization issues are listed. The boundaries between stages are somewhat fuzzy and the optimization problems may be defined as belonging to one stage or another.

At the higher representation levels, functional languages, finite state machines representations (FSM) or other types of representations may be used. State minimization of FSMs, for example, is a well developed topic. Other design issues must be resolved at this level because they may affect the entire design process. For instance, the required precision of the operators is a topic that has not been well explored in the literature, yet it may cause unnecessary use of silicon area, power and/or time for unnecessary precision.

Once a data flow graph is obtained, some major issues arise: scheduling of operations; assignment and allocation to hardware operators, buses and multiplexers; pipelining of operators; architectural choices (bit-serial or word parallel) and clock speed.

Next, a register-transfer (RT) level representation is used to optimize storage allocation, memory synthesis, timing and clocking schemes. At this stage and at the next stage (the logical level), a choice for libraries of components and technology mapping must be made to optimize the circuit so that, for instance, area is minimized while timing constraints are satisfied. Logic minimization may now be performed to reduce the number of logical gates. Other issues are fanout compensation and buffer insertion, which bring us to the electrical level, or

FIGURE 1

Optimization issues in the circuit synthesis process.

transistor level. At this stage, optimal transistor sizing, power consumption, clock distribution for skew minimization or compensation are some of the problems to solve.

Next, at the layout level other very large problems arise. These include problems such as placement of components and routing within a chip. The goals are to minimize area, optimize speed or throughput and minimize electrical noise, etc. Finally, at the production level, although not strictly part of the design cycle, some issues as yield maximization and maximum testability interact closely to higher levels of design, as they may be strongly affected by early decisions in the design process.

There are many other important aspects to consider which interact with most or all design levels. One such issue is reliability. A good overall circuit reliability cannot be obtained only through the use of highly reliable individual components; it is mainly the result of good design practices and methodologies.

Traditionally, high-level synthesis is followed by logic synthesis, which transforms the RT design into combinatorial logic and maps this logic into cells. Logic synthesis is then followed by layout synthesis, which generates the physical geometry of the circuit. While both logic and layout tools have been available for many years, general purpose high-level tools are still in their early years. The automation of the entire design process is highly desirable and is the subject of active research. Many components of a circuit synthesizer or silicon compiler exist or are currently being developed. Such a synthesizer will design integrated circuits which truly satisfy their specification requirements, are free of errors, and will do so in a fraction of the time that is being taken today by human designers.

1.2 Advantages of Behavioral Synthesis

In addition to facilitating design exploration by rapidly generating many different designs, behavioral synthesis also allows exploitation of an application's inherent parallelism, by working from the algorithmic level. This parallelism allows speedup of operations even when using slower hardware. In other words, it is possible to meet an application's timing constraints while reducing circuit costs (in terms of silicon area and power consumption).

A data-flow graph inherently shows what operations may be done in parallel. For example, in figure 2, both multiplications may be done either in series (schedule 1) or in parallel (schedule 2). Since behavioral synthesis works from the data-flow graph, it allows exploitation of parallelism that would be much harder or impossible at lower levels of design. The data-flow graph also shows what operations are critical to finishing the complete calculation on time, and which can be time-spread without affecting the overall completion time. In figure 2, all operations are critical.

The synthesis system must design a fixed and given algorithm but it is free to select:

- the times at which operations start and end (scheduling).
- the type of operator (carry-save, Booth, etc.).
- the basic width of each operator considering consistency with the previous item (digit-serial implementations).
- the supply voltage, or if desired, voltages.
- trade-offs between using fast operators which are turned off when idle and adding extra slower operators.
- clock period(s)
- etc.

The optimization conditions are usually general such as “complete the algorithm in 1,000 ns, keep the area within the pad frame, and minimize power.” Also, the conditions are likely to change, as the designer may want to flatten the peak power drawn to reduce V_{CC} droop, current surges, or EMI.

1.3 Thesis Contributions

The main idea behind this thesis is the optimization of silicon area, average power consumption, peak power consumption, and delay in the behavioral synthesis of ASICs through simultaneous scheduling and assignment of the operations of a data flow graph.

The thesis contains a number of contributions, as described in the following subsections.

Inputs: X, Y, Z, W;
 Outputs: Q
 $K1 = X + Y$;
 $K2 = K1 * Z$;
 $K3 = K1 * W$;
 $Q = K2 - K3$;

(a) Behavioral description

(b) Data flow graph

Schedule 1(c) Serial implementation
with faster hardware *Schedule 2*(d) Parallel implementation
with slower hardware

FIGURE 2

Exploitation of parallelism in behavioral synthesis.
 Schedule 2 exploits the data flow graph's parallelism to
 reduce the overall delay.

1.3.1 Simultaneous Scheduling and Assignment

The method introduced by this thesis combines assignment and scheduling, optimizing both processes globally and simultaneously. Note that by assignment it is meant the selection of units (hardware) to perform each operation of a data flow graph. The methodology chooses the best unit for each operation and schedules the operations to optimize the use of resources.

1.3.2 Digit-Serial Implementations at the Behavioral Level

The method presented here is also capable of choosing the best digit size of each arithmetic operator that will implement an operation of a data flow graph. The method can also combine (mix) bit-serial, digit-serial and parallel operators.

1.3.3 Average Power Consumption Optimization at the Behavioral Level

The method presented is one of the first behavioral systems that can estimate and minimize average power consumption by choosing the best hardware for each operation and simultaneously scheduling the operations. The papers that were published as a product of this research are among the first reported in the area (if not the first).

1.3.4 Power Profiles and Peak Power Consumption

In addition to average power consumption, the method can also estimate and minimize peak power consumption. For this novel contribution, the concept of **Power Profiles** is introduced.

1.3.5 Supply Voltage Choice at the Behavioral Level

The method chooses the best supply voltage (e.g., 5V or 3.3V) to satisfy system specifications while minimizing the resources needed. It is also capable of combining different supply voltages (mixed-voltage designs).

1.3.6 Multiobjective, Multiply Constrained Behavioral Optimization

The method and tools introduced can optimize the following independent parameters:

- silicon area (in terms of transistor counts)

- average power consumption
- peak power consumption
- delay

The same parameters can also be used as constraints (e.g., minimize area with a maximum delay and maximum peak power).

Furthermore, the following three parameters can be jointly optimized (i.e., simultaneously minimized) by means of a user-defined weighted formula:

- silicon area
- average power consumption
- peak power consumption

This weighted optimization can also be done with any of the parameters (delay, average power consumption, peak power consumption, silicon area) as constraints.

1.3.7 Real Parameter Values

The method and tools developed use real values for area, power, and delay, thereby eliminating restrictions imposed by some of the more traditional methods used by the high-level synthesis community. By doing this, the complexity of the problem is greatly increased, but more realistic assumptions about the design space are used.

1.3.8 Larger Applications

In addition to using real values for the circuit parameters, the methodology and tools presented here are applied to very large problems represented by data flow graphs with hundreds of operations. These are far larger than any other data flow graphs used by the behavioral synthesis community. The results obtained are quite good both in terms of the quality of the solutions obtained and the CPU times needed.

1.4 Thesis Organization

This thesis is organized as follows:

Chapter 2 presents background material on behavioral synthesis issues (assignment, scheduling, etc.).

Chapter 3 explains how average and peak power consumption can be estimated at the behavioral level, and considers the influence of silicon area on power consumption. This chapter also introduces the concept of *power profiles*.

Chapter 4 presents the hardware architectures of the adders and multipliers used throughout this thesis.

Chapter 5 presents background information on genetic algorithms.

Chapter 6 shows details of the software implementation.

Chapter 7 presents the data flow graphs used as application examples, and presents results of the digit-serial methodology applied to those problems. This methodology chooses and mixes the best digit sizes (from bit-serial to word-parallel) to implement each addition and multiplication of a data flow graph. Chapter 8 shows results of the architectural-choice methodology on the same problems. This methodology actually chooses and mixes different types of architectures to implement the additions and multiplications of a data flow graph. Chapter 9 expands the problems by unrolling the data flow graphs, generating much larger and more complex problems. The INU3 and INU5 problems are introduced by unrolling the well-known elliptic wave filter example.

Chapter 10 presents the conclusions of this thesis.

Finally, appendix A contains some details of the World-Wide-Web implementation of the tools developed.

II Behavioral Synthesis

The purpose of this chapter is to present the major behavioral synthesis concepts used throughout this thesis.

To explain these concepts, the set of adders shown in table 1 will be used. These adders have delays ranging from 20 to 50 ns, with corresponding transistors counts ranging from 10,000 to 2,000.

TABLE 1 A set of adders with a varied range of delays and transistor counts.

	Adder A1	Adder A2	Adder A3	Adder A4
Graphical Representation
Delay (ns)	20	30	40	50
Area (transistors)	10,000	5,000	3,000	2,000

2.1 The Assignment Problem

Consider the data flow graph shown in figure 3. It contains six additions (0 to 5). It is assumed that each addition will be computed by a separate hardware adder, that is, each adder will only be used to perform one of the additions. In other words, adders (the operators) will not be *shared* among the additions (the operations).

FIGURE 3

A data flow graph with six additions.

Therefore, there will be six adders to implement the operations of the data flow graph. Let us also assume that a timing constraint T_c is given, i.e., the total delay allowed for the six additions is T_c . Figure 4 (a) shows a timing diagram of the data flow graph implemented by six adders (A, B, C, D, E, and F). Adder A implements operation 0, adder B implements operation 1 and so on. For a timing constraint of 60 ns, operations 0, 2, 3, 4, and 5 must be performed by the fastest adders, i.e., those of type A1 (each with a delay of 20 ns). Operation 1, however, can be implemented by a slower adder since the total time available for this operation is $60 - 20 = 40$ ns. By looking at table 1, either an adder of type A2 or A3 can be used. To minimize area, the smallest adder (of type A3) should be chosen. Figure 4 (b) shows the final timing diagram.

Consider now a larger timing constraint: 100 ns. In this case, even the critical path may be implemented by slower operators and still satisfy the constraint. A couple of implementations are shown in figure 5. The first one (a), tries to minimize area by using the smallest possible adders (type A4) for operations 1, 3, 4, and 5. Operation 0 is implemented by an adder of type A1 while operation 2 is implemented by an adder of type A2. This combination uses a total of 23,000 transistors. A more efficient implementation is shown in figure 5 (b). Operation 0 is implemented by an adder of type A2, operation 1 by an adder of type A4, operation 2 by an

(a) Adder B may use up to 40 ns for operation 1

(b) The lowest cost adder is selected (delay of 40 ns)

FIGURE 4

Non-shared timing diagrams with $T_c = 60$ ns.

Adders are A, B, C, D, E, and F. Operations are from (0) to (5).

adder of type A2, and operations 3, 4, and 5 by adders of type A3. The total number of transistors of this combination is 21,000.

Therefore, the problem involves finding the best combination of all six adders to minimize area (or power). This is the *assignment* problem: assigning operators to operations so that timing constraints are satisfied while area or power consumption are minimized. The above examples could be solved by hand because their size is very small. Most problems are much

(a) Implementation with 23,000 transistors

(b) Implementation with 21,000 transistors

FIGURE 5

Non-shared timing diagram with a $T_c = 100$ ns. Implementation (b) is more efficient with a better use of the overall available time.

larger and may have hundreds or even thousands of operations. They may also include other types of operators, such as subtractors, comparators, multipliers, dividers, or more sophisticated arithmetic units.

2.2 The Scheduling Problem

The implementation presented in the previous section uses six adders, one for each operation in the data flow graph. Silicon area can be significantly reduced by sharing adders among operations. For example, the same adder i could be used to implement operations 0 and 4 in the data flow graph. This is possible because these operations are mutually exclusive in time; they can never be performed simultaneously due to the precedence relationship that exists between them. It is also possible for the same adder i to implement operations 0, 2, and 4 for the same reasons. Furthermore, it is even possible for a single adder to implement all operations of the data flow graph, provided that the timing constraint is relaxed enough and that a proper *schedule* is found.

This approach may use significantly less area than the case described in the previous section with the same latency (delay between the start of the first operation and the end of the last one). The disadvantage is that it achieves slower throughputs. For instance, by using different physical operators for operations 0, 2, and 4, the non-shared implementation can accept data at a much higher rate. When adder A of figures 4 and 5 completes an addition, adder C can start its own addition and adder A can start processing new data right away. The same applies for the other adders. This is not the case for the shared implementation, since the same physical adder may be used for operations 0 and 2. Once operation 0 is completed, the adder is now used for operation 2 and cannot start another operation 0.

Figure 6 shows a shared implementation of the same example used previously with a timing constraint of 60 ns. This configuration has the same latency (60 ns), yet uses only three adders (A, B, and C). Adder A implements operations 0, 2, and 3; adder B implements operations 1 and 4; adder C implements operation 5.

Therefore, operations 0 and 1 are scheduled to slot i , operation 2 to slot j , and operations 3, 4, and 5 are scheduled to slot k . This is the *scheduling* problem.

FIGURE 6

A shared implementation with a $T_c = 60$ ns.
By sharing the adders among the operations, only 3
adders are needed (A, B, and C).

Consider now another example with a timing constraint of 80 ns. The same schedule shown in figure 6 can be used to satisfy this timing constraint. However, this would be an inefficient schedule, since it would waste resources by using an unnecessary number of adders. A better schedule is shown in figure 7 (a). By scheduling operation 5 (for example) to slot l , only two adders (A and B) are needed. Adder A performs operations 0, 2, 4, and 5, while adder B performs operations 1 and 3.

Figure 7 (b) shows a schedule for a timing constraint of 120 ns. This schedule allows an implementation with only one adder (adder A). Operation 0 is scheduled to slot i , operation 1 to slot j , and so on until operation 5 is scheduled to slot n .

2.3 Joint Assignment and Scheduling

The previous section showed how sharing and scheduling can reduce the number of operators used while still satisfying a timing constraint. This section shows how the combination of assignment and scheduling can further reduce silicon area.

(a) $T_C = 80$ ns: only two adders are needed (A and B); uses 20,000 transistors

(b) $T_C = 120$ ns: only one adder is needed (A); uses 10,000 transistors

FIGURE 7

Schedules with: (a) $T_C = 80$ ns; (b) $T_C = 120$ ns.

Consider the last two schedules used in the previous section (for timing constraints of 80 ns and 120 ns). Figure 8 shows new schedules for these timing constraints, once again using the same representation of adders of table 1. These schedules show operations scheduled to each clock step (assumed to be 10 ns).

By using a combination of different types of adders, significantly smaller transistor counts and power consumptions can be achieved. Table 2 compares both approaches (only scheduling, as in the previous section, and joint scheduling/assignment). For the timing constraint of 80 ns, there is a reduction of 35% on the number of transistors. For the 120 ns timing constraint, the reduction is 30%. Note that the original implementation in this case was very efficient using only one operator of type A1. The assigned and scheduled implementation uses two operators (one of type A2 and one of type A4) but has an overall lower transistor count.

TABLE 2

Comparison between implementations.

	T _c = 80 ns		T _c = 120 ns	
	Scheduled	Assigned AND scheduled	Scheduled	Assigned AND scheduled
Transistor count	20,000	13,000	10,000	7,000

The above sections presented the basic ideas of assignment and scheduling by applying them to a small data flow graph with only six additions. The same ideas can be extended to other operations (e.g., multiplication) and much larger applications, as will be seen in the following chapters.

2.4 Previous Work

Hundreds of high-level systems have been developed over the last twenty years. It is not the purpose of this section to provide a complete survey of these techniques, but merely to point out some of the most well-known and successful approaches. The reader is directed to [4] for a much more complete and detailed survey.

(a) $T_c = 80$ ns: using two *different* adders (A and B) the transistor count is 13,000

(b) $T_c = 120$ ns: using two *different* adders (A and B) the transistor count is 7,000

FIGURE 8

Assignment AND scheduling with: (a) $T_c = 80$ ns; (b) $T_c = 120$ ns.

Traditionally, high-level synthesis has been subdivided into three types of problems: Scheduling, Partitioning, and Allocation.

Many heuristic methods have been developed for scheduling: list scheduling methods [4, 5, 6, 7], Zone scheduling [8], integer programming methods [9, 10], simulated evolution [11], Force-Directed Scheduling [12], simulated annealing [13], genetic algorithms [14, 15], etc., including incredibly simple techniques such as ALAP (as late as possible), and ASAP (as soon as possible) [1,4]. To simplify the problem, most of these techniques have traditionally made unrealistic assumptions, such as a multiplier's delay being two clock-steps and an adder's delay being one clock-step. Some of them also assume that only one type of hardware can implement each type of arithmetic operation (i.e., addition and multiplication).

Partitioning is the process of dividing the structure of the problem into smaller subproblems. This may be required to satisfy system constraints, such as maximum chip size. It is also a way to overcome the huge complexity of high-level synthesis so that efficient optimizations can be done on smaller problems (that are easier to manage). Since the second requirement is due to the tools' inability to tackle larger problems, it carries the price that local optimizations may find solutions very different from the global optima. Traditional methods for structural partitioning are random selection, cluster growth, hierarchical clustering, min-cut techniques, and even simulated annealing [1, 4, 16].

Allocation is closely related to scheduling, yet operates on a different dimension. Scheduling defines the system's operations regarding timing, while allocation defines the system's operations regarding physical resources. Allocation can be divided naturally into three categories: register allocation, interconnect allocation, and functional unit allocation. Unit selection is the process of choosing the number and type of each hardware component, and is referred to in this thesis as assignment. While unit selection is usually defined as part of allocation, unit binding is sometimes also included. It is the process of defining which specific instances of the components will be used (when there are more than one of the same type). As such, it considers mapping each operation into each specific resource, including variables into registers, registers into muxes, etc.

In many previous synthesis systems a single type of unit or very reduced set of units, such as adders and multipliers, have been used as this greatly reduces the complexity of the allocation problem (although with possibly grave consequences). Sometimes researchers have also been trying to prove that their method is more efficient than others because it uses one or two fewer registers or muxes. This is somewhat irrelevant compared to the major components of a chip, particularly multipliers, which are much slower and larger than most other components. This problem was mentioned at the latest High-Level Synthesis Symposium [19].

There are three major types of allocation algorithms: heuristic methods, linear programming methods and graph-based methods. Heuristics usually pick one component at a time (either an operation or a variable) and assign it to a hardware element. Examples are systems using graph grammars [20], expert systems [21], weight-directed approaches [22], and the left-edge algorithm [23]. Common graph techniques for allocation include clique-partitioning, or node coloring of a dual graph [24]. Linear programming methods have also been used but have traditionally been limited by the computational complexity of the problems [10, 25].

The problems tackled in this thesis involve extremely large data flow graphs (with hundreds of operations in some cases), and selection from several types of multicycle hardware components (with up to 64 cycles per operation). To overcome the limitations of many of the previously mentioned methods, genetic algorithms are used for global simultaneous scheduling and assignment in these very large and complex problems. As will be seen in later chapters, the approach has been very successful.

III Power Consumption in Behavioral Synthesis and Power Profiles

3.1 Why Low Power?

Designing low power ASICS is receiving increasing attention as chip densities and operating frequencies continue to increase, and as portable communications and personal assistant devices proliferate. The advantages of low-power circuits are many and apply not only to portable devices:

- Smaller and lighter products
- Increased reliability
- Lower heat dissipation
- Cheaper IC packages
- Longer battery life

Lower power peaks also improve supply currents, as smaller peak currents are used and voltages levels are more stable. They also lower electromagnetic interference and improve reliability by reducing electromigration effects [26, 27, 28].

The ASIC designer can tackle power consumption at several levels. These levels, along with a few examples, may be partitioned as follows.

The processing/transistor-design/environment level [29, 30].

- Reduce the supply-voltage.
- Reduce the geometry.
- Modify the threshold voltage and the subthreshold slope.
- Reduce the temperature.

The gate-design level [31, 32, 33]:

- Choose static or dynamic CMOS.
- Use reversible logic such as adiabatic gates.
- Order the transistor turn-on sequence in multiple-input gates.

The logic level [33, 34, 35]:

- Reduce hazards and other nonproductive transitions.
- Reduce the number of transitions within the operation.
- Resize the gates (i.e. reduce the gate size of less speed-critical gates).
- Use asynchronous circuits.

The register-transfer design level [36]:

- Choose adjacent state assignments for frequent state transitions.

The behavioral level [37, 38, 39, 40, 41]:

- Shut down (do not change data or clock signals) operators which are unused over a given period.
- Choose the lowest supply voltage that can meet the speed requirements.
- Mix adders and multipliers operating at different supply voltages (e.g., 3.3V and 5V).
- Choose fast, power-hungry operators only on data paths that are time critical. For example, array multipliers are quite fast but consume large amounts of power.
- Choose less-parallel arithmetic (i.e. bit or digit-serial arithmetic) where speed is less important.

The algorithmic level [38, 42]:

- Choose a low-power algorithm (for example, vector quantization uses considerable less power than MPEG coding for digital video compression; different vector quantization algorithms also consume quite different amounts of energy).

The behavioral level as applied to ASICs, not processors, is the concern of this chapter. Behavioral synthesis attacks the design problem from higher levels of abstraction. Among

other advantages, it allows exploitation of the parallelism inherent in the algorithms of many ASICs applications (notably, DSP). This parallelism allows proper design choices to compensate for slower and less power-hungry operators. This is much harder or impossible to do at lower levels of design.

This thesis also introduces **Power-Profiler**, a behavioral synthesis tool for power minimization that can attack both average and peak power problems, while also considering silicon area. It reduces power by simultaneously *assigning* modules (operators) and *scheduling* the operations of a behavioral description or data flow graph.

The following items enlarge on the examples in the behavioral level.

3.1.1 Shut Down of Operators

The major power consumption in CMOS gates occurs when transistors switch the output capacitance from V_{CC} to ground and vice-versa. The power transients produced are clearly seen on figure 9, which shows the instantaneous power supply ($V_{CC} \times I_{CC}$) of a 32-bit carry lookahead adder. Typically, depending on the clock frequency, millions of these instantaneous power transients occur per second. Therefore, it is quite costly to continuously operate large components such as adders and multipliers. A practical way to turn off such components for short intervals is to disable the clock. Then the internal nodes remain at static voltage levels, and do not consume power. The same figure also shows the instantaneous power of the same carry lookahead adder when the clock is (carefully) gated. In this case, power is reduced to insignificant levels.

3.1.2 Low Supply Voltage

The power consumption of a CMOS circuit decreases approximately quadratically with voltage. The reduction of operating speed, on the other hand, is approximately linear (at least for technologies over $0.1 \mu\text{m}$). This is clearly shown in figure 10, which shows the speed performance and power consumptions of 32-bit ripple-carry and carry lookahead adders as the voltage is decreased. The numbers are relative to the performance at 5V.

FIGURE 9

Instantaneous power supply of a 32-bit CLA. The power consumption drops to insignificant levels when the clock is disabled (disabling circuitry represented symbolically by an AND gate).

FIGURE 10

Speed and power relative performance of RCA and CLA. Values are divided by the values at 5V. "Speed" is the inverse of delay.

3.1.3 Mixed-Voltage Circuits

Dual voltages on one IC are attractive enough so that they are being considered commercially [43, 44]. Although such circuitry is viable, careful consideration must be given to lower-level problems such as latchup.

3.1.4 Use of Parallel Operators Only on Time-Critical Data Paths

In low-voltage circuits, it may become necessary to compensate for the loss in speed by increasing system parallelism. For example, consider a circuit that implements the behavioral description given in figure 11(a). The operations are shown as a data-flow graph in (b). (c) shows the traditional implementation using one multiplier operating at 5V. (d) shows a parallel implementation with two multipliers operating at 3.3V. The power consumption of

the parallel option is lower than the one of the traditional approach, while its total delay is smaller (calculations will be shown in more detail later in this chapter). As mentioned earlier, this happens because the rate of power decrease is higher than the rate of delay increase when the supply voltage is lowered.

In this example, however, the extra silicon area used by the second multiplier must also be taken into consideration. A good behavioral synthesis tool should choose the best architecture for each type of operation (addition and multiplication) and should schedule the operations to minimize power while also considering area and delay constraints.

3.1.5 Use Digit-Serial Operators Where Speed is Less Important

Bit-serial and digit-serial adders and multipliers can be both area and power efficient. Their overall delay is larger than their parallel counterparts, but they can still be used in paths that are not time critical. They can greatly reduce chip area and power, while still maintaining overall timing performance [39].

Note that power minimization done at one level can usually be accommodated within optimizations done at another level. Thus, a different process, gate design or multiplier design can easily be accommodated in the behavioral synthesis system. The designer merely runs the Spice models to characterize the power and timing behavior of the new components.

3.2 Power in Behavioral Synthesis

Behavioral synthesis works from the data-flow graph. A data-flow graph inherently shows which operations may be done in parallel and which operations are or are not critical to finishing the complete calculation on time. For example, in figure 11, both multiplications may be done in parallel. A behavioral synthesis system must design a fixed and given algorithm but is free to schedule each operation, to choose the type of operator (ripple-carry or carry lookahead adder, etc.), to select supply voltages, etc. This design space freedom allows the designer to efficiently and rapidly explore many potential design alternatives to minimize power. Also, the conditions are likely to change, as the designer may want to flatten the peak

Inputs: A, B, C, D ;
 Outputs: Y, Z
 $X1 = A + B$;
 $X2 = X1 * C$;
 $X3 = X1 * D$;
 $Y = X2 + X3$;

(a) Behavioral description

(b) Data flow graph

Schedule A

(c) Serial implementation at 5V

Schedule B

(d) Parallel implementation at 3.3V

FIGURE 11

Behavioral implementation example.
 Schedule A shows a serial implementation at 5V while
 schedule B shows a parallel implementation at 3.3V.

power drawn to reduce V_{CC} drop, current surges, electromigration, or EMI. These changes can be easily handled at the behavioral level.

The tools presented in this thesis combine using of lower supply voltages, disabling the clock of idle components and increasing system parallelism to minimize average and peak power consumption while satisfying timing and area constraints.

3.2.1 Previous Work

The introduction of power consumption constraints at the design level is relatively new. Behavioral synthesis for power consumption is even newer, with little literature available.

At the 1994 Low-Power Workshop, Mehra and Rabaey presented a system that uses analytical and stochastic techniques to estimate power from a behavioral-level description. The system achieves errors of about 20% on the estimates compared with an architectural-level simulator, and then uses the results to perform design exploration [38]. Olson and King presented a paper that uses genetic algorithms to do state assignment for reduced-power finite state machines [36].

A research team at Philips has recently reported an asynchronous approach to reduce power consumption. It relies on high-level tools to avoid some of the difficulties in designing asynchronous circuits (hazards, reliability, sensitivity to variations in supply voltage, testability, etc.). Power is reduced but the area overhead is about 100% [35].

Power consumption issues in the Hyper-LP system were also recently reported in [41]. This system considers transformations such as parallelization and pipelining, while considering dropping the supply voltage. A method based on simulated annealing is then used to perform optimization search.

In our previous work, we considered digit-serial arithmetic and inherent parallelism to speed-up circuit implementations so that the timing constraints are met while minimizing power and area; we also considered shutting off idle components to obtain better power utilization [37, 39].

3.3 Power Profiles

Several architectures of adders and multipliers [45, 46] were designed and simulated in order to obtain realistic values for their delay, area, and power characteristics:

- Ripple-carry adder (RCA)
- Carry lookahead adder (CLA)
- Modified Booth multiplier (BOOTH)
- Quasi Bit-Serial multiplier (QBS)
- 32-bit Array multiplier (ARR32)
- 16-bit Digit-serial array multiplier (ARR16)

All these operators perform 32-bit operations, including the 16-bit digit-serial array multiplier, which is used four times to compute the product of two 32-bit numbers. The implementation of these architectures is discussed in chapter IV. Each of them has a unique set of values for power consumption, delay, and silicon area.

Table 3 shows the set of delay, average power and energy values using a system clock of 100 MHz (i.e., clock cycles of 10 ns). The “delay” rows correspond to the intrinsic delay of each operator in the case of RCA, CLA, and ARR32 and of the stages of QBS, BOOTH and ARR16. Total delay corresponds to the delay for a complete 32-bit operation (i.e., a 32-bit addition or 32-bit multiplication).

All input registers have been included in each operator.

Energy is the average power multiplied by total delay. The average power consumption values correspond to the circuits operating at the clock frequency. Thus, some of the implementations are more efficient than others in terms of the percentage of the clock cycle that is actually used for a computation. For example, the QBS multiplier at 3.3V needs 8.8 ns per cycle (utilization of 88% of the 10 ns clock cycle). However, at 5V the utilization drops to 57% as only 5.7 ns are usefully used within each cycle. This merely indicates that the practical characteristics of each operator depend on the clock frequency used. For instance, the net effect on the QBS multiplier is that the overall delay of the multiplier will be 640 ns under both supply voltages. This may not be true with other clock frequencies (e.g., with any clock

period between 5.7 ns and 8.8 ns the 3.3V implementation would require two clock cycles per stage, while the 5V implementation would require only one).

TABLE 3 Delay and average power of adders and multipliers.

		Supply Voltage	
		3.3 V	5 V
RCA	Delay (ns)	25.0	17.1
	Total Delay (ns)	30.0	20.0
	Average Power (mW)	5.4	22.7
	Energy (nJ)	162.0	454.0
CLA	Delay (ns)	11.8	7.0
	Total Delay (ns)	20.0	10.0
	Average Power (mW)	10.5	37.3
	Energy (nJ)	210.0	373.0
QBS	Delay per Stage (ns)	8.8	5.7
	Total Delay (ns)	640.0	640.0
	Average Power (mW)	11.0	28.5
	Energy (nJ)	7,040.0	18,240.0
BOOTH	Delay per Stage (ns)	11.8	8.0
	Total Delay (ns)	320.0	160.0
	Average Power (mW)	12.7	84.0
	Energy (nJ)	4,064.0	13,440.0
ARR 16	Delay per Stage (ns)	72.0	39.2
	Total Delay (ns)	330.0	170.0
	Average Power (mW)	24.6	101.3
	Energy (nJ)	8,118.0	17,221.0
ARR 32	Delay (ns)	158.5	98.5
	Total Delay (ns)	160.0	100.0
	Average Power (mW)	143.1	295.6
	Energy (nJ)	22,896.0	29,560.0

The delay and power values are obtained from HSpice simulations using Northern Telecom's *typical* 0.8 μm transistor models. We refer to the power consumption values in the table as average values since power is data dependent. These values are obtained by applying square

FIGURE 12

Inputs to adders and multipliers.

waves of different frequencies at the inputs in such a manner that a balanced mix of 1s and 0s is applied at any given time, as shown in figure 12. The pattern is repeated on the other inputs.

This input pattern is, in a sense, random, and is applied to both inputs of the operators. More accurate estimates may be obtained if fixed coefficients are used in one of the inputs (this is usually the case with digital filters), in some particular cases. Alternatively, probabilistic or statistical estimates may also be used [47, 48] or the effects of input signal correlations (as in the most significant bits of signed numbers) may be considered [49].

Figure 13 shows the energy consumption and delay for each type of operator implementing a complete 32-bit operation. Although they may appear to be close to zero, the values for the adders are those of table 3. It is clear, however, that the adders have delays and energy values significantly smaller than those of the multipliers.

Figure 14 shows the I_{CC} current of a 16-bit array multiplier operating at 5V with values obtained directly from Spice. The more negative the current, the higher its absolute value is. Since instantaneous power is directly proportional to this current, it can be seen that most of

3.3 Power Profiles

FIGURE 13

Energy consumption and delays per 32-bit operation.

FIGURE 14

 I_{cc} current on the ARR16 multiplier.

the power is dissipated soon after new inputs are clocked. Although specific values may change with the input data, this characteristic is typical and applies to other adders and multipliers. At 3.3V, the general shape of the curves is similar, with lower power values and larger delays.

Figure 15 shows the power of an RCA adder operating at 5V. Note that there are two glitches (or spikes) in the curve. The first one corresponds to the inputs switching (i.e., becoming ready for the next cycle but without actually being clocked). The second spike corresponds to the inputs being clocked and the adder starting the addition (this is the power consumed by the addition itself).

FIGURE 15

Instantaneous power of RCA 5V.

A similar result is shown on figure 16 for the CLA adder at 5V and at 3.3V. Notice that at 5V the power peak is higher than the RCA's peak and that the duration is shorter, as expected. On the 3.3V adder the power peak is substantially reduced while the duration is increased.

Figure 17 shows the peak power of the Booth multiplier during several clock cycles and zoomed on a single cycle.

Figure 18 shows the peak power of the QBS multiplier during several clock cycles and zoomed on a few cycles.

Notice that the shape of the curves is somewhat similar for all the operators soon after the inputs are clocked. There is always a quick burst of energy, which reflects all the initial switching in the circuit.

FIGURE 16

Instantaneous power of the CLA adder.

FIGURE 17

Instantaneous power of the Booth multiplier.

FIGURE 18

Instantaneous power of the QBS multiplier.

FIGURE 19

Power profiles: RCA32 and CLA32.

It is possible to plot a profile of the power dissipation of each operation by simulating each operator and then averaging the power consumption at each clock cycle¹. For example, the RCA needs 2 clock cycles to perform a 32-bit addition at 5V. An average value is obtained for a “typical” first clock cycle by simulating a large number of additions and then averaging the power dissipated during the first cycle of each addition. Similarly, an average is obtained for a “typical” second cycle by averaging the values of the second cycle of each addition. Figure 19 shows the typical power profile, or dynamic power distribution of the adders. Fig-

1. This is done by using the *measure* command of hspice to measure RMS power at each clock cycle.

ure 20 shows the power profiles of the multipliers. Although with each specific set of inputs the exact values will be different, the overall shape of the curves remains similar.

These power profiles reflect the behavior of each architecture. The Booth multiplier uses regular cycles to compute a multiplication. This results in almost homogenous consumption throughout their execution times. The QBS multiplier has an interesting profile with a small peak centered on the middle of the multiplication and decaying towards the end. The RCA and the 32-bit array multiplier do not behave in regular cycles and most switching happens at the start of the computation. This is also reflected on their power profiles. The 16-bit digit-serial array multiplier is another interesting combination, as it is used four times to compute a 32-bit multiplication and needs a final clock cycle for adding partial results (at 5V). The 3.3V Booth multiplier needs two clock cycles for each stage, since it cannot operate at 100 MHz. The first cycle consumes more power than the second, as no power is used after the computation is completed.

3.4 Module Selection (Assignment) and Average Power

Module selection, i.e. the *assignment* of operations to functional units, can have dramatic effects on power consumption. To show this we will consider the operations of the data flow graph of figure 11 under several different assignments.

To simplify the calculations, let us assume that the same type of adder implements operations 0 and 3 and that the same type of multiplier implements operations 1 and 2. If t_{ADD} is the delay of an addition and t_{MUL} is the delay of a multiplication, then the overall execution times of the data flow graph implemented with this assignment under schedules A and B are:

$$t_A = 2 \times t_{ADD} + 2 \times t_{MUL}$$

and

$$t_B = 2 \times t_{ADD} + t_{MUL}$$

FIGURE 20

Power Profiles of the ARR16, ARR32 and QBS32 multipliers.

P_{ADD} and P_{MUL} are the average power consumptions for the adders and multipliers. Then, average power consumption values under schedules A and B are calculated by:

$$P_A = \frac{2 \times P_{ADD} \times t_{ADD} + 2 \times P_{MUL} \times t_{MUL}}{t_A}$$

$$P_B = \frac{2 \times P_{ADD} \times t_{ADD} + 2 \times P_{MUL} \times t_{MUL}}{t_B}$$

Consider a supply voltage of 5V, a carry lookahead adder used for the additions, and a QBS multiplier used for the multiplications. Using the values of table 3, $t_A = 1,300$ ns and $t_B = 660$ ns, while $P_A = 28.6$ mW and $P_B = 56.4$ mW. These values assume that a new computation is started as soon as the previous one is completed (i.e., 100% duty cycle). If ripple-carry adders and bit-serial multipliers or array multipliers are used instead, the values change completely. Table 4 shows the delays, power and energy values for a few combinations.

TABLE 4 Effects of assignment on average power and energy consumption (5V).

		Adders	
		RCA	CLA
Multipliers	QBS	$t_A=1320$ ns $P_A=28.3$ mW $E_A=37388$ nJ	$t_A=1300$ ns $P_A=28.6$ mW $E_A=37226$ nJ
		$t_B=680$ ns $P_B=55.0$ mW $E_B=37388$ nJ	$t_B=660$ ns $P_B=56.4$ mW $E_B=37226$ nJ
	BOOTH	$t_A=360$ ns $P_A=77.2$ mW $E_A=27788$ nJ	$t_A=340$ ns $P_A=81.3$ mW $E_A=27626$ nJ
		$t_B=200$ ns $P_B=138.9$ mW $E_B=27788$ nJ	$t_B=180$ ns $P_B=153.5$ mW $E_B=27626$ nJ
	ARRAY 32	$t_A=240$ ns $P_A=250.1$ mW $E_A=60028$ nJ	$t_A=220$ ns $P_A=272.1$ mW $E_A=59866$ nJ
		$t_B=140$ ns $P_B=428.8$ mW $E_B=60028$ nJ	$t_B=120$ ns $P_B=498.9$ mW $E_B=59866$ nJ

This shows how different assignments can affect power and energy consumption. If the supply voltage were 3.3V, yet another completely different set of values would be obtained.

The power values of table 4 cannot be compared directly unless the same delay is assumed. The assignment problem is generally stated as: *given a maximum delay t_i , find the best assignment that minimizes average power consumption.* Consider both cases using RCA adders and QBS and Booth multipliers under schedule A. It is misleading to assume that the combination using the QBS multiplier consumes less power because the delay of both approaches is different.

If the maximum delay allowed is, for instance, 1,320 ns, then the power of the Booth combination drops to 21.1 mW, lower than the QBS combination. This occurs because the adders and multipliers remain idle the rest of the time without using any significant power. Similarly, the power of the array implementation drops to 45.5 mW. However, if the delay allowed is 140 ns, the only feasible solutions use array multipliers. Table 5 shows the average power consumption of all combinations using a delay of 1,320 ns. Since the delay is large enough, both schedules give the same power consumption.

TABLE 5 Delay normalized to 1,320 ns.

		ADDERS	
		RCA	CLA
Multiplier	QBS	$P_A=P_B= 28.3 \text{ mW}$	$P_A=P_B= 28.2 \text{ mW}$
	BOOTH	$P_A=P_B= 21.1 \text{ mW}$	$P_A=P_B= 20.9 \text{ mW}$
	ARRAY 32	$P_A=P_B= 45.5 \text{ mW}$	$P_A=P_B= 45.4 \text{ mW}$

The combination that minimizes power can be read directly from the table and uses a CLA adder and a Booth multiplier.

The energy values shown in table 4 correspond to those needed to compute two additions and two multiplications with the different types of adders and multipliers. Thus, they are independent of the schedule as they are based on "energy per operation" values. The combination that minimizes energy consumption can be read directly from the table (CLA adder and Booth multiplier). This type of information is very useful if the objective is to maximize battery life, but only without regard for timing constraints (for example, if the maximum delay allowed were less than 180 ns, then the above solution would be infeasible). This type of objective

does not involve any architectural-choice optimization, as the architectures that consume the least energy should always be chosen (precisely Booth multipliers and CLA adders).

The concern of this thesis is with applications that involve timing constraints, either in the form of maximum delay or minimum throughput. These need both architectural choice and scheduling optimizations for power consumption minimization.

The example used in this section is very small with only four operations and considers only two schedules. The solution space of this type of problem grows exponentially with the number of operations of the data flow graph. With the limited set of adders and multipliers used in this example, the number of combinations per schedule is 2×2 (two additions, two types of adders) plus 4×4 (two multiplications, each being implemented by four types of multipliers) = 20. A typical practical application may contain hundreds of operations and a larger library from where to select the modules. With 50 additions and 50 multiplications and our library the number of choices is approximately $2^{50} \times 4^{50} = \sim 10^{45}$. This is the complexity of the assignment problem alone. The scheduling problem will be more apparent when we consider peak power and silicon area.

3.5 Scheduling, Assignment and Peak Power

The previous section dealt with average power, without considering the power profiles of each operator and the resulting peak power profile of the entire circuit. For the reasons explained earlier (high currents, power supply droop, electromigration and reliability, EMI), peak power must also be considered in many applications.

Figures 21 and 22 show the peak power profiles of a data flow graph with three additions implemented with ripple-carry adders (schedules C, D, and E), carry-lookahead adders (schedules G and H) and a mix of both types of adders (schedule F).

Peak power and maximum delays are shown in table 6. Schedules D and E are faster than schedule C, but their peak power is higher as well. Schedule F is faster than C, yet has very similar peak power due to a different assignment (uses a CLA for one of the operations).

Inputs: A, B, C ;
 Outputs: Y, Z ;
 $X = A + B$;
 $Y = B + C$;
 $Z = C + D$;

FIGURE 21

Effects of schedules C, D, and E on peak power.

FIGURE 22

Effects of schedules F, G, and H on peak power.

Schedule H is the fastest, has the highest peak power, but consumes the least average power when used at a delay of 60 ns (i.e., a normalized delay).

TABLE 6 Delays and peak power of several schedules.

		Delay (ns)	Peak Power (mW)	Average Power at min. delay (mW)	Average power at 60 ns (mW)
Schedule	C	60	32.7	22.7	22.7
	D	20	98.1	68.1	22.7
	E	40	65.4	34.1	22.7
	F	50	37.3	25.6	21.3
	G	30	37.3	37.3	18.7
	H	10	111.9	112.0	18.7

The table above shows that peak power and average power optimization are different problems. While the optimum configuration (for a particular goal) may seem obvious for this small example, it is quite a different matter with much larger real applications. Typically, the total number of combinations, including assignment and scheduling, can reach over 10^{500} , depending on the size of the data flow graph and on the timing constraint used. In addition to the scheduling of operations to precise time slots, all precedence relations between operations must be obeyed (these can easily reach thousands).

3.6 Should Silicon Area be Considered?

The two previous sections presented how assignment and scheduling affect average and peak power. We must now consider a third important issue: silicon area.

While increasing system parallelism speeds up a circuit and may compensate for slower operators (particularly when using 3.3V), it carries an area cost which cannot be overlooked. Table 7 shows the number of transistors used by the different types of adders and multipliers used in this thesis. Alternatively, the values of this table can be substituted by the actual silicon areas (in μm^2 , for instance) of the library used.

TABLE 7 Transistor counts of adders and multipliers.

RCA	CLA	QSS	BOOTH	ARR 16	ARR 32
1,330	2,326	1,944	3,808	11,386	32,896

Scheduling plays a very important role in the silicon area needed. Consider the schedules of figures 21 and 22. Schedule C assigns the additions to nonoverlapping time slots. Hence, the implementation can use a single adder for all three operations (sharing of resources). Similarly, schedule D needs three adders to perform all operations in parallel. Therefore, the improved time performance comes at the cost of three times as many transistors. Schedule H is the fastest but also requires the largest number of transistors for its three CLA adders. Figure 23 shows relative performance of delay, power and transistor count for each schedule compared to schedule C.

This graph also shows that each assignment/scheduling combination has unique characteristics. The best one depends on what type of function is being optimized: area, average or peak power, delay.

Since operators using a supply voltage of 3.3V are slower, it is possible that for tight timing constraints the number of transistors needed will be much higher than with supply voltage of 5V. This might be due to increased parallelism (thus increasing the number of operators of each type), or to the use of faster (and larger) architectures (i.e., array multiplier instead of Booth multipliers).

Consider schedule A of figure 24 operating at 5V and using RCA adders and Booth multipliers. The delay is 360 ns and the number of transistors is 5,138. The same configuration and the same schedule with a supply voltage of 3.3V has a delay of 700 ns (schedule A1).

If a delay of 360 ns is desired, the options are to use a schedule such as A, but using array multipliers (schedule A2 of figure 25), or a schedule similar to B (on the same figure) and an CLA adder. The total transistor count for both options is 35,222 and 9,942 respectively. Therefore, in this example a 3.3V implementation needs considerably more transistors than a 5V implementation. In some cases, the large area may render the 3.3V solution impractical.

FIGURE 23

Performance of schedules C, D, E, F, G, and H. Values are relative to those of schedule C.

In other cases, it may not even be possible for a 3.3V solution to meet very fast timing constraints.

3.7 Defeating the Power Profiles

From the Spice simulations it is clear that some characteristics are always present in the power profiles. This is the case, for example, of a large amount of energy being spent as soon as the inputs are clocked. In this section we examine situations in which the profiles may not be accurate. To do this we try to find particular cases which should “defeat” the profiles. One such case should involve the ripple-carry adders. Two very different conditions may occur:

- the inputs are clocked and absolutely no carry is generated in any bit position

FIGURE 24

The effects of supply voltages on total delay.
Transistor counts are identical with similar schedules.

FIGURE 25

Reducing delay at 3.3V with schedules A2 and B. Schedule A2 achieves a smaller delay by using faster hardware while schedule B by increasing parallelism.

FIGURE 26

Power profiles of ripple/no ripple conditions.

- the inputs are clocked and a carry ripples through all the way from the least significant bit to the most significant bit.

These very different situations suggest that their energy distributions should be completely different. However, this is not the case, as shown in figure 26. In both cases the same number of inputs is changed (33 inputs), but for the ripple case the inputs are carefully chosen to fully propagate the carry bit.

Both curves are actually extremely close (notice the difference in the circles pointed by arrows). If a clock period of 10 ns is used, then the RCA requires two clock cycles to compute a 32-bit addition at 5V. The average power consumption in both cycles is (45.3 mW, 3.7 mW) for the ripple condition and (44.2 mW, 1.5 mW) for the no-ripple condition. The ripple effect is also clearly seen for the ripple case between 3 ns and 18 ns approximately. Most of

the peak power occurs actually at the beginning of the cycle when the inputs are clocked. This occurs because the registers are fairly big compared with the adder itself. Obviously this does not necessarily mean that similar results will be obtained for special conditions on larger or more complex operators (i.e., array multipliers).

The above test on an RCA indicates that the power distributions are relatively similar for very different conditions. However, in the above simulation all inputs were changed and clocked at the same time. It may also be expected that if the inputs remain stable, a considerably smaller amount of power would be consumed. This is confirmed by figure 27 (a), which shows the power consumed when only two, five, ten and fifteen inputs are randomly changed.

Figure 27 (b) shows a comparison between the ripple case (in which 33 inputs change) and the case when only fifteen inputs change. The difference here is much smaller.

Table 8 shows the average power of these examples during both cycles of the RCA adder.

TABLE 8 Power comparison among several cases (RCA).

	Power (mW)	
	0-10 ns	10-20 ns
ripple	45.3	3.7
no ripple	44.2	1.5
0 inputs	0.7	1.6
2 inputs	5.0	3.0
5 inputs	11.1	1.8
10 inputs	22.9	0.8
15 inputs	30.3	0.9
64 inputs	68.9	37.1

These values confirm that power varies significantly depending on how many inputs change from cycle to cycle. If no inputs change, the average power drops to very low levels. The average power rises gradually as more inputs change. When 15 inputs change the average power is quite close to the average power of table 3.

(a) Different number of inputs changing

(b) The ripple case and 15 inputs changing

FIGURE 27

Peak power with different number of inputs changing.

3.8 Concluding Remarks

The results shown in the previous section indicate that, by carefully selecting the inputs, it should be possible to obtain quite different power profiles compared with the “average power profiles” (which are obtained with a large number of input samples). On the other hand, when these values are lower than the average, the error would be on the safe side. A few options can be concluded from these observations:

- The largest possible values (all inputs changing) can be used to be on the safe side (i.e., a worst case scenario).
- A more complex statistical model can be used. For example, one in which it would be guaranteed (or at least expected) that two thirds of the cases will fall within a certain margin of error of the power profile. This is similar to a standard deviation measure. This approach is similar to the current model but with more careful consideration on the number of samples taken. For instance, with a Montecarlo-type approach the precision obtained can be precisely defined by the number of samples taken [48].
- The operator could be simply shut down if the inputs do not change. Then it would not consume power at all but its outputs from the previous computation would have to be available. The resulting effect is that the operator will behave as an idle component. This can be particularly useful for the larger and more time consuming operations (i.e., multiplication).

In any case, it will be seen in later chapters that optimization for peak power should always be performed, since it costs very little in terms of increased average power. The benefits, on the other hand, can be quite substantial, as electromigration effects and EMI are reduced and voltage and current levels are more stable.

It is also clear that more work is needed in the area of power profile characterization of each operator. The cost of an operator shut-down has also not been considered. While it is true that power drops to insignificant levels if no switching activity occurs (at least with technologies over 0.1 μm and without significant leakage currents), there is an area and power cost associated with the shut-down circuitry. This should also be studied further.

IV Hardware

This chapter discusses digit-serial arithmetic implementations and briefly presents the multiplier architectures used. The behavioral synthesis tools presented in this thesis choose the best combination of these architectures to optimize area, power and/or delay.

4.1 Digit-Serial Operations

This section explains the benefits of digit-serial operations using an addition example. It shows how bit and digit-serial arithmetic can be used to reduce silicon area while maintaining fast operating speeds.

Figure 28 shows the block diagram (a), an implementation with registers (b), and a simple timing diagram of a one-bit adder (c).

FIGURE 28

1-bit adder block diagrams.

(a) without registers; (b) with registers; (c) timing diagram.

The delay of the adder is the longest time elapsed between the instant the clock latches the inputs into the registers and the results are ready. The delay of the adder in figure 28 (b) can be calculated as:

$$delay_1 = T_r + T_a$$

where T_r is the time needed by the registers to store and latch the inputs (including setup times and clock to output times) and T_a is the delay through the full adder. Under normal conditions, the adder would operate continuously, with the clock occurring as fast as possible to maximize the rate of additions per unit of time (the *throughput*).

The area used by the entire configuration of this one-bit adder is

$$area_1 = A_a + 5 \times A_r$$

where A_a is the area of the full adder and A_r is the area of each register

Consider now a two-bit addition. A two-bit adder is shown in figure 29 (a). This is a ripple configuration in which the carry-out of the least significant bits (a_0 and b_0) is fed into the carry-in of the most significant bits (a_1 and b_1). To generalize the applications of this adder, a register stores the carry-in. Figure 29 (b) shows the timing diagram for this adder. In this case, the delay is:

$$delay_2 = T_r + T_a + T_a = T_r + 2 \times T_a$$

The delay equals the time for the register to store the input values plus the delay through the least significant adder plus the delay through the most significant adder.

The area used by this adder is:

$$area_2 = 2 \times A_a + 8 \times A_r$$

FIGURE 29

A two-bit adder.

(a) two-bit adder (b) timing diagram.

FIGURE 30

Two-bit addition using a one-bit serial adder.

The two-bit addition can be implemented in an alternate manner by using the one-bit adder twice (serial addition). The first time it adds the least significant bits a_0 and b_0 with a carry-in of 0 and produces the outputs z_0 and c_{out} . The second time it adds the most significant bits a_1 and b_1 with the carry-out generated by the first addition. This adder is shown in figure 30 (a). The carry-out is connected to the carry-in of the one-bit adder. The carry-in register shows a reset line necessary so that a 0 is used by the first addition.

Figure 30 (b) shows the timing diagram for this bit-serial addition. At t_0 , a_0 and b_0 are loaded into the registers and the addition starts, producing result z_0 and the carry-out. At t_1 , a_1 , b_1 , and the carry-out of the first addition are loaded into the registers and the second addition starts. At t_2 , z_1 is ready. Therefore, the total delay for the two-bit addition is equal to two clock cycles, and each clock cycle must be greater than the following delay (where $_{1s}$ means 1-bit in serial configuration):¹

$$delay_{1s} = T_r + T_a$$

Therefore, the total delay for the addition of two bits with this 1-bit adder is (where $_{2,1s}$ means a 2-bit addition implemented with a 1-bit serial adder):

$$delay_{2,1s} = 2 \times T_r + 2 \times T_a$$

The area used by this adder is:

$$area_{2,1s} = A_a + 4 \times A_r + A_{rr}$$

A_{rr} is the area of the carry-in register, which must include the reset capability (which may consist of a simple AND gate). Note that an extra register has been included in the formula to store the result z_0 as well (it must be either stored or used immediately by the following stage). Table 9 summarizes the results for a two-bit addition using a two-bit adder and using a one-bit serial adder.

TABLE 9 Areas and delays for a two-bit addition.

two-bit adder		one-bit serial adder	
total delay	area	total delay	area
$T_r + 2 \times T_a$	$2 \times A_a + 8 \times A_r$	$2 \times T_r + 2 \times T_a$	$A_a + 4 \times A_r + A_{rr}$

1. The delay of the reset register is considered to be the same as the delay of a normal register.

If T_a is much larger than T_r , then the total delay of both approaches is very similar. The difference is only T_r , which is the time needed to store the carry-out from the first addition. However, the areas of both approaches is quite different. A_a is much larger than A_r and than A_{rr} . While the two-bit adder uses $2 \times A_a$, the bit-serial approach only uses $1 \times A_a$. In practical terms, the area of the two-bit adder is slightly less than twice the area of the one-bit serial adder. In other words, the one-bit serial adder is slightly larger than half of a two-bit adder. This is the fundamental concept of using serial instead of parallel operators: significantly less area with slightly slower speed.

The same ideas can be extended to a four-bit addition. It can be implemented in a number of ways: using a four-bit adder, using a two-bit adder twice in series, or using a one-bit adder four times in series. Figure 31 shows the four-bit adder, the two-bit sequential adder, the one-bit sequential adder and the corresponding timing diagrams for a four-bit addition.

Table 10 shows the delay and area values for a four-bit addition implemented with these adders. In terms of area, the four-bit implementation is roughly four times larger than the one-bit serial implementation and about two times larger than the two-bit serial circuit ($4 \times A_a$, $2 \times A_a$, and $1 \times A_a$, plus several A_r , respectively). In terms of delay, the bit-serial adder is slightly slower than the two-bit serial adder, which, in turn, is slightly slower than the four bit adder (delays of $4 \times T_a$ plus several T_r).

TABLE 10 Delay and area of the four-bit addition implementations.

four-bit adder		two-bit serial adder		one-bit serial adder	
total delay	area	total delay	area	total delay	area
$4 \times T_a + T_r$	$4 \times A_a + 14 \times A_r$	$2 \times (2 \times T_a + T_r)$	$2 \times A_a + 8 \times A_r + A_{rr}$	$4 \times (T_a + T_r)$	$A_a + 6 \times A_r + A_{rr}$

From the above results, the values for areas and delays of additions of any regular size word length (1, 2, 4, 8, 16, and 32 bits) can be deduced. These are different for full-word and digit-serial implementations. Considering that:

$$n = \text{word length (in bits)}$$

FIGURE 31

Implementations of a four-bit addition.

and

k = digit length (in bits)

The formulae are:

$$\text{delay} = n \times T_a + T_r \quad (\text{full-word}) (n=1 \dots 32)$$

$$delay = \frac{n}{k} \times (T_r + k \times T_d) \quad (\text{digit-serial}) (n=2\dots 32; k=1\dots n/2)$$

$$area = (3n + 2) \times A_r + n \times A_d \quad (\text{full-word}) (n=1\dots 32)$$

$$area = k \times A_d + (n + 2k) \times A_r + A_{rr} \quad (\text{digit-serial}) (n=2\dots 32; k=1\dots n/2)$$

Table 11 shows the area and delay values for 32-bit additions.

TABLE 11 Delay and area for implementations of 32-bit addition.

	Digit Size (bits)					
	1	2	4	8	16	32
area	$A_d + 34A_r + A_{rr}$	$2A_d + 36A_r + A_{rr}$	$4A_d + 40A_r + A_{rr}$	$8A_d + 48A_r + A_{rr}$	$16A_d + 64A_r + A_{rr}$	$32A_d + 98A_r$
delay	$32T_r + 32T_d$	$16T_r + 32T_d$	$8T_r + 32T_d$	$4T_r + 32T_d$	$2T_r + 32T_d$	$T_r + 32T_d$

Figure 32 shows a qualitative graphical representation of this timing/area relationship.

4.1.1 Transistor Counts, Time Delays and Power Consumption

It can be seen from the previous section that significant area savings can be achieved by using digit-serial adders instead of full-word adders, provided that A_d is significantly larger than A_r and that T_d is also significantly larger than T_r . To obtain more realistic values, all the adder configurations from 1 to 32 bits were designed and tested with HSpice. The transistor models used were the “typical” models of Northern Telecom. The channel length used was 0.8 μm . Table 12 shows the delays and transistor counts of each configuration. Delays are the minimum possible delay for a full 32-bit operation by assuming a 100% duty cycle (i.e. operations start as soon as the previous ones finish). These designs include muxes and shift-registers but do not include extra control circuits that may be necessary to operate each particular adder.

FIGURE 32

Qualitative representation of the timing/area relationship. As the digit size decreases the area is reduced at a higher rate than the delay increase.

TABLE 12

Addition: Transistor counts, delays and power consumption.

	Digit size (in bits)					
	1	2	4	8	16	32
transistor count	296	328	392	520	776	1280
total delay (ns)	52.5	36.9	29.3	24.5	22.3	19.7
power (mW)	2.91	3.58	4.74	6.45	8.96	12.72

The normal approach to perform a 32-bit addition is to use the 32-bit adder. By using a 16-bit serial adder instead, there are savings of 39% in the number of transistors used (776 against 1280), while the delay is increased by 13% (22.3 ns against 19.7 ns). If a 1-bit serial adder is used, there are savings of 76% in the number of transistors used (296 against 1280), that is, the number of transistors used is less than $\frac{1}{4}$ of the number used in the 32-bit configuration. The delay is increased by 263% (52.5ns against 19.9ns), that is, the delay is 2.7 times larger.

Therefore, by using a digit-serial approach the area savings are considerably larger than the delay penalties. The area coefficient α , delay coefficient β , and measure of efficiency η are defined below and plotted in figure 33. These correspond to a 32-bit addition implemented with digit-serial adders of size k bits.

$$\alpha = \frac{\text{area}(k)}{\text{area}(32)}$$

$$\beta = \frac{\text{delay}(k)}{\text{delay}(32)}$$

$$\eta = \frac{1}{\alpha \times \beta}$$

FIGURE 33

Plot of α , β , and η for a 32-bit addition.
 α is $\text{area}(k)/\text{area}(32)$; β is $\text{delay}(k)/\text{delay}(32)$;
 η is $1/(\alpha\beta)$;

η is a reasonable measure because a reduction of 50% in area yields an improvement of 100%, while an increase of 100% in the delay yields a reduction of 50%. However, while the plot of figure 33 might indicate that the best efficiency is obtained with a digit size of 4 bits, η is by no means an absolute and uncontested efficiency measure. It assumes that area and delay are equally important. This may not be the case in some applications. Furthermore, during the synthesis process it is usually necessary to include timing constraints, which may render this measure impractical or useless. With timing constraints, it will be necessary to choose digit sizes so that operators are fast enough to satisfy them, regardless of the "best" area/speed relationship. This will be more apparent later on the applications chapters, in which timing constraints are used.

In addition to area and delays, table 12 also shows values for power consumption for each digit size configuration. These values were obtained directly from the power values provided by HSpice, on simulations using the same 0.8 μm technology described earlier. The inputs to each adder were cycled through multiple combinations and the simulations were run for long periods of time to obtain accurate values. The frequencies for each set of inputs were changed appropriately for each digit size. Smaller sizes use higher frequencies, while larger sizes use lower frequencies (adders are run at maximum speed).

Although power results may appear to correlate well with the area values, they are independent of each other. Furthermore, the power values are obtained with 100% duty cycle and they have to be adjusted for idle periods. Therefore, area-minimization synthesis and power-minimization synthesis are totally different problems. However, it is possible to combine both parameters as long as a correctly weighted formula is used (or more advanced techniques, such as a niched GA Pareto approach). This formula may be different for each particular case, depending on which parameter is more important. Perhaps the ideal cost function should translate both silicon area and power consumption into dollar amounts.

The example shown in this section used a ripple-carry adder, mainly because of its simplicity and for being so easy to understand in digit-serial implementations. However, the concepts presented are valid for the other common types of adders, such as carry-lookahead, carry select, carry-skip, etc., with minor variations. For example, a carry-lookahead technique can be applied to 4-bit, 8-bit, 16-bit, and 32-bit adders, but is not too practical for a 2-bit adder. A

32-bit addition can be performed with a 32-bit CLA adder or with a 16-bit CLA adder used twice, or an 8-bit CLA adder used four times, etc. The timing/area relationship is merely different.

4.2 Multiplication and Multipliers

Binary fixed-point multiplication can be accomplished with sequential or combinatorial techniques. The sequential techniques discussed here are Booth's algorithm, the bit-pair recoding method and a quasi-serial multiplier. The combinatorial techniques discussed include array multipliers and modular array multiplication.

The sequential techniques offer the advantage of relatively small hardware consisting of nothing more than a few registers, muxes and an n -bit adder. The combinatorial techniques use far more hardware but operate considerably faster as well. This is a typical trade-off between area and speed.

4.2.1 Booth Algorithm

The most basic multiplication algorithm is the paper-and-pencil method, which simply shifts and adds the multiplicand n times according to the bit values of the n -bit multiplier. The hardware needed is very small and simple consisting of (shift) registers and a full-word adder. The Booth algorithm and the Bit-Pair Recoding scheme perform the same multiplication approximately twice as fast with approximately the same area [45, 50].

The number of additions during a multiplication can be reduced by scanning the multiplier bits one by one. This is based on the fact that p consecutive 1s on a binary string are equal to the subtraction of the values of only two positions:

$$\begin{array}{cccccccc}
 \dots & j+p & j+p-1 & \dots & \dots & j & j-1 & \dots \\
 \dots & 0 & 1 & \dots & \dots & 1 & 0 & \dots
 \end{array}$$

$\underbrace{\hspace{10em}}_{p \text{ consecutive 1s}}$

But

$$2^{j+p} - 2^j = 2^{j+p-1} + 2^{j+p-2} + \dots + 2^{j+1} + 2^j$$

Then the string is equivalent to:

$$\begin{array}{ccccccccccc}
 \dots, & j+p+1, & j+p, & j+p-1, & \dots, & j+1, & j, & j-1, & \dots \\
 \dots, & 0, & 1, & \underbrace{0, \dots, 0}_{p-1 \text{ consecutive 0s}}, & -1, & 0, & \dots
 \end{array}$$

↑
Addition
↑
Subtraction

As an example, the value of 30 may be represented as the binary string

$$0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 = 16 + 8 + 4 + 2 = 30$$

This can be rewritten as

$$0 \ +10 \ 0 \ 0 \ -1 \ 0 = 32 - 2 = 30$$

This is valid for any number of consecutive 1s (including a single one). The Booth recoded multiplier algorithm works equally well for both positive and negative numbers, as shown in the example below.

0 1 0 1 1 (+11)	0 1 0 1 1
1 1 0 1 0 (-6)	0 -1 +1 -1 0
<div style="display: flex; justify-content: space-around;"> 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 1 0 0 0 0 1 0 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0 </div>	
<div style="display: flex; justify-content: space-around;"> 1 1 1 1 0 1 1 1 1 0 (-66) </div>	

The transformation of the string with consecutive 1s to the one with an addition and a subtraction is called “skipping over 1s”. It can be seen that this technique can speed up the multiplication process by reducing the number of additions required. However, as seen in the above example, both the original string and the recoded string have a total of three 1s, which implies that both will need three additions to perform the multiplication. In fact, it is possible that the recoded string will cause even more additions than the original string. Therefore, the

speed-up factor is not constant and depends on each particular multiplier. In the worst-case situation the multiplier will have sequences of 1s followed by 0s ($\dots 1010101\dots 01\dots$), which will cause almost twice as many additions than originally. Table 13 shows recoded strings for worst-case multiplier, a normal multiplier, and a good multiplier.

TABLE 13

Good and bad cases of Booth's technique.

Worst case (recoded)	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1
Normal (recoded)	1	1	1	0	0	1	1	0	1	1	0	0	1	1	1	0
	0	0	-1	0	+1	0	-1	+1	0	-1	0	+1	0	0	-1	0
Good case (recoded)	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1
	0	+1	0	0	0	0	0	0	-1	0	0	+1	0	0	0	-1

This data-dependency of the multiplication's execution time may be less important if there exists some knowledge of the values of the multipliers. For example, if the values are small (either positive or negative) the high order bits are expected to be all 0s or 1s and the efficiency of the algorithm will be quite high. However, the next section will present a method that is twice as fast as the standard add-shift technique and that has a constant execution time.

4.2.2 Bit-Pair Recoding

With the Bit-Pair Recoding technique a maximum of $n/2$ additions will be generated for an n -bit multiplier [43]. This is done by looking at two bits at a time and adding or subtracting a particular value of the multiplicand each time (therefore $n/2$ operations). This is shown on table 14.

The procedure can be verified on the example shown in figure 34. From this example, multiplying a number by 54 is equivalent to adding the multiplicand (-2) times, (+8) times, (-16) times and (+64) times, that is, adding the multiplicand a total of $-2 + 8 - 16 + 64 = 54$ times.

While the standard add-shift method would have required 8 additions, the bit-pair scheme requires only 4. The original Booth algorithm also requires 4 additions in this case. However, bit-pair requires 4 additions for any 8-bit multiplier, unlike Booth's technique.

FIGURE 34

Example of bit-pair recoding.

TABLE 14

Bit-pair recoding table.

Bit Pair		Right side	Operation on multiplicand
$j+1$	j	$j-1$	
0	0	0	-
0	0	1	+1 x
0	1	0	+1 x
0	1	1	+2 x
1	0	0	-2 x
1	0	1	-1 x
1	1	0	-1 x
1	1	1	-

Hence, by consistently looking at pairs of bits, the total number of additions to be performed is always $n/2$, where n is the word length of the multiplier.

FIGURE 35

Basic hardware for bit-pair recoding multiplication.

A possible hardware design to implement the bit-pair recoding technique is shown in figure 35 (from [45]). This is very similar to the hardware required for the standard add-shift method. The biggest difference is that three bits are now decoded from the multiplier to choose which version of the multiplicand to add. A 5-to-1 multiplexer is also used for this selection (multiplexes among $+1x$, $-1x$, $+2x$, $-2x$, and $0x$ the multiplicand).

4.2.3 The Quasi-Serial Multiplier

The Quasi-Serial Multiplier (QBS) was presented by Earl Swartzlander in [44]. It uses a faster scheme than basic add and shift multiplication and requires comparable hardware. It is based on the following expression of a multiplication:

$$|A \times B| = \sum_{p=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k \times 2^k \times b_p \right) \times 2^p = \sum_{l=0}^{2n-2} \left(\sum_{m=0}^l a_m \times b_{l-m} \right) \times 2^l$$

This expression can be implemented directly by the simple circuit reproduced on figure 36. During the first cycle, the least significant bit of the multiplier is added with the least significant bit of the multiplicand and this generates immediately the least significant bit of the result (which will be output in the next cycle). Then the multiplicand is shifted by one bit and its two least significant bits are now added with the two least significant bits of the multiplier, the number of ones is counted and added into the result register. This process is repeated a total of $2n$ times.

4.2.4 Modular Array Multiplication

Combinatorial planar array multipliers are much faster than those using sequential techniques mainly because no synchronization and no controlling software are required. By software, we are referring to the relatively complicated programming required in computer programs to implement multiplications or to the microprogramming of synchronous sequential designs. Planar arrays, however, require (almost) only logic gates that operate as combinatorial circuits.

It is possible to implement binary multiplication of positive values in a combinatorial logic array. Furthermore, the multiplication can be modularized because entire sections can be performed in parallel.

FIGURE 36

The Quasi-Serial Multiplier block diagram.

Consider the example of figure 37, which shows the usual paper-and-pencil algorithms for eight-bit operands. Four distinct four-bit sections can be clearly seen. This operation can then be modularized in sections of four-bit operands using identical modules. The multiplication of two eight-bit operands can be performed using the four modules shown in figure 37. Each module performs local multiplication independently of the results of other modules.

Each module in figure 37 can perform the complete multiplication of 2 four-bit operands. The multiplier may be defined as the concatenation of two four-bit numbers A_H and A_L and the multiplicand may be defined as the concatenation of two four-bit numbers B_H and B_L according to table 15.

In figure 37, module 1 performs the multiplication of A_L and B_L , module 2 performs the multiplication of A_H and B_L , module 3 performs the multiplication of A_H and B_H , and finally,

FIGURE 37

Modules for eight-bit operands multiplication.

TABLE 15

Splitting multiplier and multiplicand into 2 numbers each.

PART	BITS			
A_L	a_3	a_2	a_1	a_0
A_H	a_7	a_6	a_5	a_4
B_L	b_3	b_2	b_1	b_0
B_H	b_7	b_6	b_5	b_4

module 4 performs the multiplication of modules A_L and B_H . Each of these is a complete and normal four-bit by four-bit multiplication. For example, the two numbers A_H and B_H can be multiplied by the paper-and-pencil method as in figure 38.

This is exactly the same multiplication that module 3 performs. To generate the final product, these subproducts are then added together.

$$\begin{array}{r}
 A // \quad a_7 \quad a_6 \quad a_5 \quad a_4 \\
 B // \quad b_7 \quad b_6 \quad b_5 \quad b_4 \\
 \hline
 \quad \quad \quad a_7b_4 \quad a_6b_4 \quad a_5b_4 \quad a_4b_4 \\
 \quad \quad a_7b_5 \quad a_6b_5 \quad a_5b_5 \quad a_4b_5 \\
 \quad a_7b_6 \quad a_6b_6 \quad a_5b_6 \quad a_4b_6 \\
 a_7b_7 \quad a_6b_7 \quad a_5b_7 \quad a_4b_7 \\
 \hline
 2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0
 \end{array}$$

FIGURE 38

Paper-and-pencil multiplication of A_H by B_H .

4.2.4.1 A Multiply Module

Figure 39 shows the design of a 4 x 4 array multiplier using a module (a *Non-Additive Multiply Module*, as called by Cavanagh [45]). Note that the operations performed by this module are an exact match of those performed by the paper-and-pencil method.

4.2.4.2 Wallace trees

The four multiplier modules generate several products. This means that to generate the final result, several inputs with the same weight must be added (for instance, modules 1, 2, and 4 generate a bit with weight 2^4). These bits are added by carry-save adders organized as Wallace trees. An n -input Wallace tree is a bit-slice summing circuit, which produces the sum of n bit-slice inputs [51]. Figure 40 (a) shows the representation of a three-bit-slice Wallace tree (W_3). Note that there are 3 input bits and 2 output bits. Figure 40 (b) shows a five-bit-slice Wallace tree (W_5), with 5 inputs and 3 outputs. Note that this tree is made of three W_3 trees. Figure 40 (c) shows a seven-bit-slice Wallace tree (W_7), with seven inputs and 3 outputs. Note also that this tree is made of four W_3 trees.

The alignment of the four 4x4 modules is shown on figure 41 for an 8x8 multiplication [45]. The figure clearly shows the bits where Wallace trees are needed (2^4 through 2^{11}).

FIGURE 39

A 4 x 4 NMN (Non-additive multiplier module).

A more general module arrangement is shown in figure 42 for array multiplication networks ranging in size from 4 x 4 to 32 x 32 bits [43]. The figure shows how 4x4, 8x8, and 16x16 modules can be stacked up to form the higher-order products. It also shows the size of the Wallace trees required for each bit weight. For example, for a 16 x 16 array, Wallace trees of sizes 3, 5, 7, 7, 5 and 3 are required (the number of blocks on the vertical direction); for a 32 x 32 array, trees up to 15 inputs are needed.

This figures shows nicely how smaller modules can be reused to compute different sections of the multiplication. As we saw with the 8x8 multiplication, the four 4x4 modules are totally independent of each other. Therefore they can be used in parallel or in series, i.e., shared among the multiplications $A_H \times B_H$, $A_H \times B_L$, $A_L \times B_H$, and $A_L \times B_L$.

FIGURE 40

Wallace trees: (a) W3; (b) W5; (c) W7.

If the modules are used in series, the area needed is one quarter of the area of the 8x8 multiplier plus the overhead for the multiplexers and additional control circuitry.

The ARR32 multiplier used in this thesis is the 32-bit implementation represented by figure 42. The ARR16 multiplier is the 16x16 block shown in the same figure. The same module is used four times to compute a complete 32-bit multiplication.

FIGURE 41

Arrangement of 4x4 modules for an eight-bit multiplication.

4.3 Delays, Transistor Counts and Average Power Consumption

Table 16 shows the delays, transistor counts and average power consumption values used for the digit-serial adders and multipliers. Adders are of the ripple-carry type and multipliers are of the array type (1 and 2-bit multipliers are extrapolations regarding total delay and area). The supply voltage is $5V^2$.

The delays are adjusted for a 100 MHz clock frequency. The detrimental effects of clock frequency are clear on the smaller adders as their delay is perfectly linear with the digit size (i.e. the bit-serial adder is twice as slow as the 2-bit adder, which in turn is twice as slow as the 4-bit adder and so on). This is because the smaller adders are not used as fast as they could. For

2. Note that these implementations are slightly different from the ones discussed in the previous chapter. The input registers and the gate-level designs are different.

FIGURE 42

Arrangements for 4×4 , 8×8 , 16×16 , and 32×32 multipliers.

TABLE 16

Characteristics of digit-serial operators (5V).

			Delay (ns)	Power (mW)	Transistors
ADDERS	Digit Size (bits)	1	320	2.9	296
		2	160	3.3	328
		4	80	3.9	392
		8	40	5.2	520
		16	30	7.9	776
		32	20	13.8	1280
MULTIPLIERS	Digit Size (bits)	1	1510	39.8	4434
		2	1140	42.8	4668
		4	650	51.1	4914
		8	330	75.1	5628
		16	170	143.0	11002
		32	100	323.5	32896

example, the bit-serial adder has an intrinsic delay of 1.64 ns, however the minimum clock cycle is 10 ns. A much higher clock frequency will take advantage of this very fast adder. The best efficiency would be obtained with a frequency of 610 MHz, which is too high for current CMOS technology. Nevertheless, the smaller adders are still useful as they use fewer transistors and can be used when the timing constraint allows.

The delays, transistor counts and average power consumption values of the adders and multipliers used in architectural-choice problems are shown in tables 3 and 7 in chapter III. Details of the simulations are also discussed in chapter III.

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

V Genetic Algorithms

This chapter presents background on genetic algorithms, their history, relationship to biology, and fundamental theory. It discusses several issues, such as their programmability and parallelization, and also briefly introduces genetic programming.

Many combinatorial optimization methods have been developed. In general, each has its own particular application domain and limitations. For instance, smaller problems can be solved by very simple techniques such as random search and enumeration. Random search can also be applied to very large and complex problems in which the optimization method cannot be easily determined.

For medium size problems, dynamic programming, and branch and bound type solutions are commonly used. Dynamic programming is a very well-known technique that can be applied to problems with a sequence of interrelated decisions. The problem is usually divided into stages and a decision is required in every stage. Each of these stages has several states. The technique is used when the fitness function is the sum of the fitness functions of each stage of the problem. In addition, given a state, the next decision cannot depend on those stages already solved. Furthermore, a recursive relationship must exist for the cost or fitness between stages [52]. Branch and bound is a technique that attempts to reduce the amount of combinatorial explosion of the search space that can be applied to problems with an underlying tree structure [53].

For larger problems, beam search, random search, genetic algorithms, and simulated annealing are some of the more common methods. Beam search retains only the best nodes of a branch and bound search keeping memory within manageable limits. Simulated annealing is a modified hill-climbing technique with inspiration from metallurgical processes. It attempts to climb a local hill, usually accepting the new point if it is a better one and sometimes even

if it is a worse one (but in this case with a decreasing probability). In other words, it attempts to avoid being trapped in a local hill by occasionally accepting a bad move. Similarly to hill-climbing, and contrary to genetic algorithms, it moves from a single point to another and does not see an overall picture of the solution space [54].

Genetic algorithms, on the other hand, are an optimization technique proven to provide robust, efficient, and effective search in complex search spaces. The technique has widespread application in many areas, including business, sciences and engineering. Genetic algorithms are successful because [55-56]:

- They are based on very simple, but very powerful operations.
- They are not restricted by assumptions about the search space (continuity, existence of derivatives, unimodality, etc.).
- They exhibit an inherent parallelism, exploiting the search space with multiple points.
- They are a robust technique, balancing efficiency (the time to reach a solution) and efficacy (the quality of the reached solution) in many different environments.
- They are very flexible and easy to program, allowing fast experimentation.

5.1 Biology and History of Genetic Algorithms

Genetic algorithms were developed to mirror the laws of natural evolution and genetics. Evolution occurs on chromosomes, organic structures that encode the characteristics of living beings. In nature, biological beings compete for resources and mating. Only the strongest (or sometimes the smartest) manage to survive and procreate, causing their chromosomes to be copied into the genetic material of their offspring. This is known as the process of *natural selection* or “survival of the fittest.”

Evolution occurs with reproduction. Children are created combining genetic material from the parents' chromosomes (*crossover*) while sometimes genes are altered, introducing completely new characteristics in the population (*mutation*). As such, evolution operates on chromosomes and not directly on the biological organisms they represent. Furthermore, evolution

does not remember how well each characteristic performed in the past (i.e., it has no memory of previous performance).

The birth of Genetic Algorithms occurred in the 1970's, when John Holland incorporated these biological concepts into computer programs [55]. Since then, GAs have evolved enormously to become an important topic of computer science. They have also sparked off the development of Genetic Programming (which uses the same ideas to evolve computer programs).

5.2 Basics

GAs are based on a structured, but randomized, information exchange. They use probabilistic rules to generate and evolve populations. Similarly to nature, hundreds or possibly thousands of generations occur. In each generation a new set of artificial creatures (strings) is created combining parts and pieces of the previous one and introducing an occasional new part for variation.

Genetic algorithms require the structure of the problem to be coded as a finite-length string with a finite alphabet. As an example, consider the function

$$f(x) = 1.7x^2 - \frac{x^3}{20} \quad \text{if } (x \leq 21) \text{ or } (x \geq 25)$$

subject to the constraint

$$f(x) = 0 \quad \text{if } 21 < x < 25$$

The function is plotted for the integer range 0-31 in figure 43. First, the parameter x is coded as a finite-length string. This can be done simply using a binary string $a_1 a_2 a_3 a_4 a_5$, where a_i can be 0 or 1. The value of the string corresponds to the binary value of x , that is, the string value ranges from 0 0 0 0 0 (decimal 0) to 1 1 1 1 1 (decimal 31).

FIGURE 43

Plot of function $f(x)$.

Function has a discontinuity between $x=21$ and $x=25$. Maximum is 286.6 at $x=21$.

The objective here is to try to find the best x that maximizes the function $f(x)$. Most other optimization approaches might start from an initial point and then apply some transition rules (usually deterministic) to obtain the next point. This method may be risky as it usually locates false peaks in problems with multiple local optima (multimodal search spaces), and it may get stuck on them. GAs, on the other hand, start from a population of several initial values and evolve from this to a new and improved population. Therefore, while traditional techniques move cautiously from one point to the next, GAs move in parallel from a number of points to a number of better points.

The initial population is usually chosen randomly. Table 17 shows an example of an initial population, showing each string's fitness value. This fitness value here is merely the value of $f(x)$ applied to the value represented by the binary string.

TABLE 17 Initial population and fitness values.

String #	Binary String	Decimal Value	Fitness	Percent of Total
1	1 0 1 0 1	21	0.0	0.0
2	0 0 0 1 1	3	14.0	1.8
3	0 1 0 1 0	10	120.0	15.0
4	1 1 0 1 1	27	255.2	31.9
5	1 1 1 1 0	30	180.0	22.5
6	1 0 0 0 0	16	230.4	28.8

After this start, successive populations are generated using surprisingly simple operators which manipulate the chromosomes strings, copy them and exchange their pieces. The three major operations are:

- Reproduction
- Crossover
- Mutation

5.2.1 Reproduction

Reproduction is a process which makes copies of members of the population. These are selected according to a probability proportional to their fitness. This means that better members have a higher chance of contributing one or more offspring in the next generation. It also means that the new generation should have a higher average fitness value.

An easy way to implement the reproduction operator is to create a roulette wheel where each current string has a slot whose size is proportional to the fitness. In the previous example, the sum of all the fitnesses of the six individuals is 799.6. String 1 receives a null slot since its fitness is 0. String 2 receives a slot corresponding to 1.8% of the total. String 3 receives a slot corresponding to 15.0% of the total and so on, as shown in table 17. The roulette wheel for this initial generation is shown in figure 44. To reproduce, the wheel is simply spun six times,

(a) Single arm wheel

(b) Baker's multiarm wheel

FIGURE 44

Roulette wheels for initial population.

Baker's wheel prevents highly fit members from being missed and/or from dominating the population.

to create a new generation of six individuals. Since the chance of a particular slot being selected depends on the fitness of the individual, more highly fit strings will have a higher number of offspring in the following generation. The roulette wheel is shown on figure 44 (a).

A much improved technique for selection is Baker's multiarm spinner method [57], in which the spinner has as many equidistant arms as there are members in the population. Each member gets a piece of the roulette pie according to its fitness value. The spinner is spun, and each arm then points to a member that will be reproduced in the next generation. In the previous example, since there are six members, the arms are separated by 60 degree angles as shown on figure 44 (b).

Baker's method greatly improves selection, eliminating the possibility of very bad selections in which bad members are still chosen in spite of their low probabilities (this is always a possibility when using pseudo-random numbers). It also prevents highly fit members from being selected over and over and eventually dominating the population. In fact, Baker's method works so well that a single run of a GA is similar to the average of many runs using one-armed wheels.

Once a member has been selected for reproduction, a copy of its string is entered in a *mating pool*, where crossover and mutation will take place.

As an example, let us assume that after spinning the wheel the results are those shown in table 18. These strings (4, 6, 4, 5, 6, and 3) correspond to the mating pool from which generation 1 will be obtained.

TABLE 18 The mating pool.

Spin #	String Selected
1	4
2	6
3	4
4	5
5	6
6	3

5.2.2 Crossover

During the crossover operation members of the mating pool are selected at random. Then an integer position i along the strings is chosen at random. Two new strings are created by

exchanging the two substrings of each chromosome (divided at position i). For instance, given two strings A_1 and A_2 ,

$$\begin{array}{lcl} A_1 & = & 0 \ 1 \ 1 \ 0 \ 1 \\ A_2 & = & 1 \ 1 \ 0 \ 1 \ 0 \end{array}$$

let us assume that the crossover point selected is 3. The strings are then split at the crossing point:

$$\begin{array}{lcl} A_1 & = & 0 \ 1 \ 1 \ - \ 0 \ 1 \\ A_2 & = & 1 \ 1 \ 0 \ - \ 1 \ 0 \end{array}$$

The separated parts of the strings are then swapped between the strings, generating two new strings A'_1 and A'_2 :

$$\begin{array}{lcl} A'_1 & = & 0 \ 1 \ 1 \ 1 \ 0 \\ A'_2 & = & 1 \ 1 \ 0 \ 0 \ 1 \end{array}$$

Table 19 shows a possible set of randomly chosen mates from the mating pool of our example. The table also shows the randomly chosen crossover site and the new population (generation 1) obtained. The average fitness value of the new population is now 219.8, compared with 133.3 of generation 0 (initial generation). These results are fairly typical as the average fitness value increases with each new generation. Typically, there are big increases with the initial generations and the increases get smaller as the number of generations grows.

TABLE 19

Mates, crossover sites and generation 1.

String #	Generation 0: Binary string	Mate	Crossover site	Generation 1: Binary string	Decimal value	Fitness value
1	1 1 0 1 1	4	2	1 1 1 1 0	30	180.0
2	1 0 0 0 0	6	3	1 0 0 1 0	18	259.2
3	1 1 0 1 1	5	4	1 1 0 1 0	26	270.4
4	1 1 1 1 0	1	1	1 1 0 1 1	27	255.2
5	1 0 0 0 0	3	2	1 0 0 1 1	19	270.7
6	0 1 0 1 0	2	3	0 1 0 0 0	8	83.2

Each member of the population represents a complete idea of how to perform the given task. Each of the strings contains many substrings which are referred to by Goldberg as "notions". Genetic algorithms exploit the huge number of good notions that exist in the population by combining them through crossover. Eventually, this process results in better members.

5.2.3 Mutation

The least important genetic operator is mutation. Although reproduction and crossover search and combine chromosomes, they may sometimes lose genetic material or may fail to find new useful material. Mutation helps protect against this by introducing totally new material. In practical terms, mutation is the random alteration of the value of a string position (with a very small probability). In most cases, this means changing a bit from 1 to 0 or vice-versa. Typical mutation rates are in the 0.1% range, that is, one bit every thousand or so. The relative lack of importance of mutation will be clear in the chapter VI, which discusses implementation issues and compares several mutation rates.

5.3 Reproduction: Steady State or Generational GA?

There are two basic reproductive techniques: *generational reproduction*, and *steady-state reproduction* [58, 59].

The first technique is the one referred to previously in this chapter in which an entire new population is created at each generation.

The second one replaces only one or a few members of the population at a time. A few new members are selected according to their fitness and are immediately introduced in the new generation. In order to keep the population size constant, an equal number of members has to be removed.

Deletion of members can be performed according to many methods:

- **Random deletion:** Members are chosen randomly and are then deleted from the population.

- **Least fit:** The least fit members of the population are deleted (i.e., the bad members).
- **Exponential ranking:** The worst member has a certain probability of being deleted, for example 1/5. If it is not deleted, then the next to worst member has a 1/5 chance of being deleted (giving it a total chance of 1/25), and so on.
- **Reverse fitness:** This is the opposite of the normal selection. The probability of deletion is proportional to how bad its fitness is.

From the above techniques, random deletion is definitely the least effective in most cases [58].

The efficiency of the steady-state deletion techniques depends on the particular problem. The two reproduction methods, generational and steady-state are quite different and the choice of which to use can dramatically affect the performance of the GA. Studies have shown that both approaches are roughly equivalent with respect to the Schema theorem (see section “The Schema Theorem: GA Theory” on page 94). However, it can be seen that steady-state GAs can run potentially much faster than their generational cousins. This is not only because of the faster reproduction operation (CPU wise, this task might be insignificant compared with the other computational tasks involved - specially in the very complex problems we are interested in), but also because of faster convergence.¹

Figure 45 shows an approximation of the results of an experiment involving a population of size 100 initially containing 10 members with fitness 10, 10 members with fitness 9, 10 members with fitness 8 and so on². The steady-state approach uses least-fit deletion and only replaces one member at each generation. It can be seen that the proportion of “best” members rises to about 100% of the total after about only 200 “generations” (these are not really generations since they create only a few new members), while in the generational approach, the proportion of “best” members rises to about 90% of the total population after 20 generations. The authors claim that, because of the number of evaluations involved, 20 generations of a generational GA are equivalent to 2000 “generations” of a steady-state approach. Thus, the

1. Premature convergence may become a problem though.

2. This is extracted from [58].

FIGURE 45

Generational vs. Steady-State GAs.

steady-state approach has very good potential. These results were only partially validated by my own experiments (please refer to sections 6.5 and 6.6).

While generational GAs must provide a guarantee that the best members will be present in the next generations, there is no such a need with steady-state approaches, since the best members are always included in the next generation. Furthermore, steady-state allows a performance improvement by using a technique that Davis calls *steady-state without duplicates* (SSWD, for short), in which children that are identical to their parents are discarded [60]. As a result, no two members of the population are identical. This may speed up the convergence to the right solution while at the same time preventing the population from fully or prematurely converging.

SSWD introduces the overhead of computing a very large number of tests to check whether a child exists in the current population. However, as mentioned earlier, in real-world problems most of the computational time is spent in the evaluation of individuals.

FIGURE 46

One-point crossover.

A single point is randomly chosen to "cut" the chromosome.

5.4 Types of Crossover

Crossover may be implemented in several ways. The most common approaches are:

- **One-point:** This is the type used in the previous example. It consists of swapping contiguous pieces of 1:0 strings, as shown on figure 46.
- **Two-point:** This crossover also swaps substrings but the substring may start from anywhere in the original strings, creating, in fact, two substring exchanges, as shown on figure 47.
- **Uniform (also known as multi-point crossover):** This type uses a mask of bits, the same length as the chromosome length to exchange bits from one string to the other, as shown on figure 48.

FIGURE 47

Two-point crossover.

The chromosome is "cut" by two random points.

All three methods can be seen as mask-based. It is just the pattern of the mask that changes.

The efficacy of each crossover method depends on the size of the "notions" embedded in each string. For instance, problems in which large substrings that must stay together to preserve a highly fit solution may not perform well with the very fragmented mask of uniform crossover (since this would destroy the notions). However, uniform crossover has been shown to perform better at combining "schemata" (the building blocks that exist in a chromosome - see section 5.6 on page 94) than one or two-point crossovers [61]. It is also generally accepted that two-point crossover performs better than one-point crossover [60].

FIGURE 48

Uniform crossover.
Multiple random points "cut" the chromosome.

5.5 Fitness Techniques

Fitnesses may need to be normalized to avoid performance degradation on a GA. There are several fitness normalization techniques. Let us first consider the reasons for performance degradation.

Consider an example in which the objective function values range from 0 to 2.0 and another function which is the same function except that values are added by a constant 200. The latter function will have values ranging from 200.0 to 202.0. If a good individual has a fitness value of 1.7 in the first case, it will have a value of 201.7 in the second case. In the first case, the

good individual will be reproduced at a much higher rate since the selection procedure compares an individual's fitness to the average fitness of the population. If the average is 1.0 and the population is of size 10, in the first case the individuals' share of the selection pie is 17.00%. In the second case, this corresponds to an individual with fitness 201.7 and an average of 201. This individual's share of the selection pie is only 10.03%. Since GAs are supposed to work as in the first case, some type of normalization is needed.

Table 20 shows several typical normalization techniques used with GAs. The first line corresponds to the original fitnesses of 6 individuals of a population (not the one used previously). The second and third lines show the normalized fitnesses of a *windowing* technique in which each chromosome's fitness is equal to the amount by which its fitness exceeds the minimum fitness of the entire population. A minimum amount larger than the smallest fitness in the population might be used to protect against the lowest chromosomes having no chance of reproduction at all (an interesting feature to preserve genetic diversity). The fourth and fifth lines of table 20 show linear fitness normalizations in which the chromosomes are ordered by decreasing evaluations. The rate of decrement is a parameter in this type of normalization.

TABLE 20 Fitness techniques.

	Ind. # 1	Ind. # 2	Ind. # 3	Ind. # 4	Ind. # 5	Ind. # 6
Original Evaluation	200	9	8	8	4	1
Windowing with Minimum = 1	199	8	7	7	3	1
Windowing with Minimum = 10	199	10	10	10	10	10
Linear Fitness with Decrement = 1	100	99	98	97	96	95
Linear Fitness with Decrement = 20	100	80	60	40	20	1

Note that these examples include an individual with a much better fitness than most of the population, a *super-individual*. This individual would probably eliminate its competitors in a few generations if the fitness is not normalized and the original evaluation is used. This is not a good strategy because the individual will have very little chances of exchanging genetic material with other members of the populations and, as a result, the population will probably converge rapidly.

Chromosome: 0 1 0 1 0 1 1	Number of schemata: $2^7 = 128$	
Examples of schemata:	1 1 * 1 1 * 1	* * * 1 1 * 1
	* 0 * 1 * *	1 * 0 * 1 * *
	1 * 0 * 1 * 0	0 * * * 0 1 1

FIGURE 49

Examples of schemata (*building blocks*) in a chromosome.

The other issue is the presence of a *close race*, in which several individuals have very similar fitnesses. GAs often find sets of very close competitors near the end of the runs. In this case it may be advisable to increase the amount of selection pressure so that the best competitors will reproduce disproportionately.

It has been found that linear normalization works well for both difficult phenomena, the presence of a super-individual and of close races [57, 60]. However, it might be possible to obtain better results by shifting dynamically from one normalization technique to another as the GA evolves. I have not seen this discussed in the literature.

5.6 The Schema Theorem: GA Theory

Crossover combines *building blocks* of highly fit members. Figure 49 shows a typical chromosome and some of its *building blocks*. Each of these is represented by a string made up of three characters: 0, 1 or *. A 0 or 1 at specific positions means that the chromosome must have identical values at those positions for the chromosome to contain the block. A "*" indicates a "don't care".

Each building block is called a schema. The success of genetic algorithms is based on their manipulation of schemata, as shown by Holland's Schema Theorem:

If f is the average fitness of all chromosomes containing a schema b , n is the number of chromosomes in the population that contain b , a is the average fitness of all chromosomes in the population, then the number of instances of b in the next population is $n \times \frac{f}{a}$ (not counting the effects of crossover and mutation).

In other words, this theorem says that a schema that occurs in a chromosome with above-average fitness will occur more frequently in the subsequent generation and that a schema that occurs in below-average chromosomes will appear less frequently.

Crossover leaves a schema intact if the crossover point is outside the schema (i.e., the schema is not cut), but it will probably modify the schema when the crossover point falls inside the schema. For example, consider the schemata 0 * * * * 1 and * * 0 0 1 * *. It is clear that the longer schema will hardly survive a crossover operation, while the shorter one is less likely to be destroyed. As a result, schemata of short lengths are usually maintained by the crossover operator. Mutation, at the normal low rates, does not affect short schemata. Therefore, short and highly fit schemata are propagated by giving exponentially increasing samples to the better members' chromosomes. For similar reasons, schema preservation is quite sensitive to the coding alphabet used and to the representation of the problem.

Goldberg shows a proof that n^3 schemata are processed for each generation's n evaluations³. This means that about n^3 ideas or notions are actually being tried at each generation. This is called *implicit parallelism* [56].

5.7 Parallelization of GAs

Having a large number of points that are evaluated at each generation, GAs can take advantage of parallel machines or distributed architectures to greatly speed up performance. In fact near linear speedup can be achieved [62].

For example, a simple (and simplistic) approach is to have each member evaluated in a different machine or on a different node of a network. What is done most of the time is to divide

3. Note: This proof has been contested on the Internet (comp.ai.genetic).

the population into groups and to place each subpopulation in a processor or node. Each processor may run the GA on its population and then send the best individuals to its neighboring processors. This may achieve a significant speed-up but has a risk of isolating subpopulations.

Parallel implementations may also be recommended for some complex applications, such as the simulation of natural phenomena (weather forecast). These applications use a huge number of variables and therefore require excessively large populations to be feasible with sequential implementations.

5.8 Hybrid Genetic Algorithms

While genetic algorithms are robust and perform efficiently with a variety of problem types, they might not be the *best* algorithm to solve a particular application. GAs work well by using a (usually) binary representation which can encode nearly everything. However, specific knowledge about the application is not present in the representation (except for the fitness function).

The real-world problems that must be optimized might be very hard and complex, and resources for the optimization might be limited. It may be necessary to develop a special type of GA that uses the available knowledge of the application, if such exists. This may be particularly useful when there are already existing methods to optimize the problem. In [60], Davis suggests the following procedure for hybridizing a GA:

- Use the current encoding. This guarantees that the domain knowledge embodied in the encoding used by the current algorithm will be preserved. It will also make the hybrid algorithm feel natural to the user.
- Hybridize where possible, by incorporating the positive features of the current algorithm. For example, if the current algorithm is fast, it can be used to generate or to add individuals to the initial population. It might also be advantageous to incorporate the algorithm's encoding *and* decoding strategies.

- Adapt the genetic operators by creating crossover and mutation operators for the new type of encoding by analogy with a bit string crossover and mutation operator. Incorporate domain-based heuristics as operators as well.

The use of these three techniques may allow the creation of a very efficient specialized algorithm. By applying these three principles, the resulting hybrid GA will likely do better than a robust GA applied to the specific problem. It will almost certainly do better than the current specialized algorithm since its good features will be combined with the powerful genetic operations. However, the specialized heuristics incorporated must be very carefully studied so that the GA does not become “yet another heuristic” and loses one of its main advantages, robustness.

5.9 Programmability

Yet another advantage of genetic algorithms is their easy programmability. A simple bit string encoding of the problem and the objective function is all that the GA needs to start running. The algorithm does not care about the details of the application, it just solves it.

The details of a particular application may be extremely complex. For instance, in this thesis a GA is used on a problem that combines combinatorial choice with scheduling. This is a very large and complex application, for which no efficient heuristics methods exist. The problem is so complicated that the programming of a heuristic, if it existed, on a computer would be too difficult. With GAs there is no such problem.

5.10 Another Example

This section presents another example of GAs in action.

The problem is to optimize the function (referred to as *double sinc*)

$$f(x, y) = \left(0.3 + \frac{\sin \frac{x - 226.05}{34}}{\frac{x - 226.05}{34}} \right) \times \left(0.3 + \frac{\sin \frac{y - 328.05}{35}}{\frac{y - 328.05}{35}} \right)$$

FIGURE 50

Plot of a double sinc.
Function has a discontinuity around $x=540$ and $y=336$.

subject to

$$f(x, y) = 0.001 \quad \text{when } 432 \leq x \leq 648 \text{ and } 264 \leq y \leq 408$$

Figure 50 shows a plot of this function for positive values of x and y up to 1024. Note that this function is not continuous. Therefore, derivatives are not defined. This is absolutely not a problem for GAs since they do not actually know *what* they are optimizing, they simply optimize the function without knowledge of its behavior.

Each solution of this problem will be represented by a 20-bit chromosome made of two substrings. Each substring has a length of 10 bits and represents the decimal values of either x or

y. Therefore, using this representation, there are a total of 1,048,576 possible solutions (2^{20}). The population size is 10 and the number of generations allowed is also 10.

Figure 51 shows the initial randomly generated population and a typical 10th generation. This was obtained using a generational GA, with Baker's arm for selection, two-point crossover (with a probability of crossover of 100%) and no mutation at all. The best solution obtained is within 10% of the true optimum⁴.

Note the diversity of the initial population. This is expected since this generation was obtained randomly. At generation 10, it can be seen that many individuals are quite similar and that the same schemata is present in several chromosomes. These results are fairly typical of genetic algorithm runs. In the initial phases the population is heterogenous in that it consists of randomly generated chromosomes that are quite dissimilar. At this point the performance of the GA is similar to that of an algorithm that generates random chromosomes and then evaluates them. Beginning with the next generation (generation 1 in our example), crossover starts playing a stronger role in recombining features of the best chromosomes. This recombination phase lasts for several generations when the population converges somewhat on a single solution. From this point on, the algorithms searches around the region of that solution with mutation being the principal introducer of diversity in the population.

Figure 52 shows the evolution of the average fitness and of the best fitness of the population through the 10 generations. This is a fairly typical fitness curve. The average fitness usually increases from generation to generation due to the Schema theorem. The incumbent optimal solution might actually be found before the last generation, as in the case in figure 52. Note that the best solution with a fitness of 1.47 was found at generation 8. The algorithm can easily keep track of the incumbent optimum and in which generation it was obtained.

4. These solutions were obtained with a GA developed in a spreadsheet program (Quattro Pro).

GENERATION 0:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	1	1	0
2	1	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	1	1	1
3	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	0	1	1	0	0
4	1	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	1	0
5	1	1	1	1	1	1	0	1	1	0	0	1	0	1	0	0	1	1	1	0
6	1	0	1	1	1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	0
7	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
8	0	1	0	1	0	0	1	1	0	0	0	1	1	1	0	1	0	1	0	1
9	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	1
10	0	0	1	0	0	0	0	0	0	1	1	1	0	0	1	1	1	0	1	0
X										Y										

X	Y	FIT
561	798	0.09
638	431	0.09
400	172	0.00
584	838	0.07
1014	334	0.33
739	504	0.03
307	204	0.11
332	469	0.03
999	633	0.10
129	826	0.14

TOTAL:	1.04
AVERAGE:	0.10
MAXIMUM:	0.33

GENERATION 10:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	1	0	0
2	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	1	0	0
3	0	1	0	0	0	0	1	1	1	0	0	1	0	1	0	0	1	1	0	0
4	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	1	0	0
5	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	1	1	0	0
6	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	0
7	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	1	0	0
8	0	1	0	0	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0
9	0	1	0	0	0	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0
10	0	1	0	0	0	0	1	1	0	0	0	1	0	1	0	1	1	1	0	0

X	Y	FIT
264	332	1.42
264	332	1.42
270	332	1.34
264	332	1.42
276	348	1.21
260	332	1.47
264	332	1.42
268	332	1.37
266	332	1.40
268	348	1.32

TOTAL:	13.85
AVERAGE:	1.25
MAXIMUM:	1.47

FIGURE 51

Initial and last populations of double sinc example.
Average and maximum fitness of last generation are significantly higher.

FIGURE 52

Evolution of average and maximum fitness values.

5.11 About Genetic Programming

A discussion of genetic algorithms would not be complete without mentioning their younger “cousin”, genetic programming (GP). Using GP, entire computer programs can be genetically “bred” to solve problems in many different areas of artificial intelligence, machine learning, and symbolic processing.

GP evolved to try to answer the question: “how can computers learn to solve problems without being explicitly programmed?”. In other words, how to make the structure, size and shape of the program be part of the solution, rather than part of the question [63].

GP is usually met with skepticism because the vast majority of successful scientific research is based on approaches that are “correct, consistent, justifiable, deterministic, orderly, parsi-

monious, and decisive". GP contradicts all seven principles. Yet, it has been successfully applied to many areas [64]:

- optimal control (centering a cart, balancing a broom on a moving cart),
- pattern recognition (translation-invariant one-dimensional shape in a linear retina),
- discovering inverse kinematic equations (moving a robot arm to designated target points),
- sequence induction (recursive procedures for generating Fibonacci and Hofstadter sequences),
- planning (navigating along a trail and developing a robotic plan for stacking blocks in a desired order),
- emerging behavior (discovering computer programs for locating food, carrying food to a nest),
- empirical discovery (rediscovering planetary laws, finding non-linear econometric equations, etc.),
- machine learning of functions,
- automatic programming (solving pairs of linear equations, quadratic equations with complex roots, finding trigonometric identities),
- simultaneous architectural design and training of neural networks,
- missile target recognition from noisy images.

In genetic programming the individuals in the population are compositions of functions and terminals appropriate to the particular problem domain. The set of functions used typically include mathematical functions, conditional logical operations, arithmetic operations, and domain-specific functions. The set of terminals typically includes inputs (sensors) appropriate to the problem domain and constants.

The search space includes all possible compositions of functions that can be recursively composed of the available terminals and functions.

The basic genetic operators used are reproduction and crossover. Crossover operates on two expressions from the parents and produces two offspring expressions using parts from each parent. Because entire sub-trees are exchanged, this crossover always produces correct expressions. Figure 53 shows two parental and two offspring expressions depicted as trees with ordered branches. The crossover point is indicated.

One of the major issues of GP is the fitness function. Each offspring must be given a fitness value corresponding to how well it performs the desired function. For this, a number of tests must usually be generated, that is, each child program must be run through the tests.

Genetic programming is a promising field still in its early stages. Based on the number of applications in which programs have been genetically bred it appears that the technique is suitable to a wide range of applications. These include real world problems such as finding missile targets [65]. GP is an area with many difficult problems to solve. One of the problems faced by researchers is that, in general, GP bred programs are fairly unreadable, and therefore, unmaintainable (but perhaps this won't be needed in the future).

FIGURE 53

Example of crossover in genetic programming.

VI Implementation

This chapter presents details of the algorithms and the software implementation of the method discussed in this thesis. The original implementation of the genetic algorithm was based on the C version of the “Simple Genetic Algorithm” [66], which, in turn, was based on the original algorithm presented in David Goldberg’s book. The implementation has been extensively modified to include features such as Baker’s arm selection, two-point crossover, steady-state behavior, pruning, extended initialization, etc.

Many graphs comparing the efficiency of several “genetic” parameters are included in this chapter. These are presented as average solution values (the transistor counts on an area minimization problem) and as the CPU times needed to reach those solutions. All the values are averages of 20 to 60 runs on a Sun Sparcstation 5¹. Results are then analyzed on section 6.6 on page 124.

Digit-choice problems refers to those problems in which the optimizer must choose the digit size of each operator. *Architectural-choice* problems refers to those problems in which the *type* of architecture for each operator must be chosen (carry lookahead adder or ripple carry adder, QBS or BOOTH multipliers, etc.).

6.1 Encoding

Finding an efficient problem encoding is crucial to the performance of a GA. The encoding used is different for *shared* and *non-shared* problems. Consider the data-flow graph shown in

1. CPU times are based on non-optimized compilations. Optimized compilations will run approximately 2.5 to 3 times faster.

FIGURE 54

A data flow graph and a typical chromosome.

figure 54 (a). A typical chromosome is shown in (b). This chromosome is a simple string of binary values (the dark and white circles) made of several smaller strings. Each of these sub-strings encodes some characteristics of each operation of the data flow graph, which the genetic algorithm will try to optimize. These characteristics are:

- the type of operator
- the scheduling information (only for shared problems)

Note that there are two types of optimization problems: digit-serial, in which the width of each operator must be chosen, and architectural-choice, in which the architecture of each 32-bit adder or multiplier must be chosen. Each of these problems can actually be implemented in a shared or non-shared fashion. In the shared mode the operators can be reused among operations, saving considerable area but requiring the scheduling of operations.

6.1.1 Non-Shared Problems

In non-shared problems the only choice is the type of operator that will implement the operation of the data flow graph. If the problem involves digit-serial choice, then the type options are the digit sizes (from 1 to 32 bits), as shown in table 21.

TABLE 21 Operator types for digit-serial choice.

	TYPE					
	0	1	2	3	4	5
Digit size	1 bit	2 bits	4 bits	8 bits	16 bits	32 bits

In architectural-choice problems, the choices are which architecture to use for each addition (CLA, RCA) or multiplication (QBS, BOOTH, ARR16, ARR32). In mixed-voltage problems, the two supply voltages (3.3 V and 5 V) effectively double the number of possible options as each operator can use either voltage. These options are shown in table 22.

TABLE 22 Architectural choices (mixed-voltage).

Operation	TYPE	Choice
Addition	0	RCA 3.3V
	1	RCA 5V
	2	CLA 3.3V
	3	CLA 5V
Multiplication	0	QBS 3.3V
	1	QBS 5V
	2	BOOTH 3.3V
	3	BOOTH 5V
	4	ARR16 3.3V
	5	ARR16 5V
	6	ARR32 3.3V
	7	ARR32 5V

The encoding can then be very simply represented by a chromosome such as the one shown in figure 55. The computer representation of the chromosome uses 1, 2, or 3 bits for each operation. In digit-serial problems the default size is 3 bits. Since this allows a representation

“type” field

FIGURE 55

Non-shared problem chromosome and representation.

of eight different types and there are only six, two of them must be remapped to existing ones (type 6 is remapped into type 0 and type 7 into type 1). This creates some redundancy in the encoding whose practical effect is that two of the six choices of each substring are represented twice. This is not a problem since members of the population are still selected on the basis of fitness.

FIGURE 56

Table descriptors with an example.

Operation 21 is identified as a multiplication by the *ONDT*. The *HDT* says that 2 bits are used for choice starting at position 2.

In architectural-choice problems, the default sizes are 2 bits for additions and 3 bits for multiplication (four possible adders and eight possible multipliers).

The number of bits per operation is not fixed. Each operation can have a user-defined number of bits. For example, if the user wants an operation to be implemented specifically by a 32-bit or a 16-bit operator then a one-bit field must be specified for that operation. An example is shown in figure 56, which also shows some of the data structures. These are the *Operation Descriptor Table (ONDT)* and the *Hardware Descriptor Table (HDT)*. The first table describes the type of operation (addition or multiplication) while the second table describes the BPO, FIRST, and BFL fields for each operation:

- **BPO (Bits per operation):** The number of bits to use for this operation. Range: from 0 to 3.
- **FIRST:** The first type to use (default is type 0). This indicates in which position of table 22 the search for each operator begins.
- **BFL: Bits for location:** number of bits to use for selecting a time slot (only used in shared problems)

In the example of figure 56, operation 21 is identified as a multiplication by the *ONDT*. Then *HDT* says that the first choice is a Booth 3.3V multiplier while the last choice is a ARR16-5V (field BPO is 2 and field FIRST is 2).

It is possible to specify these parameters for all operations at once. For instance, with a simple line in the input file the user can specify that all additions must be performed by CLAs and that all multiplications must be performed by either Booth or Array multipliers.

Note that the set operators is ordered and stored internally according to increasing delays. This allows easy generation of a set of initial time-feasible solutions (please refer to section 6.3.2 on page 122) as the algorithm knows easily which are the fastest operators.

6.1.2 Shared Problems

In shared problems the operators are reused and, hence, the operations must be scheduled. The encoding must be expanded to include the scheduling information, as shown in figure 57.

The extra field in the chromosome represents the slot in which the operation is scheduled. As with the fields that describe the choices of operator type, these slot fields have variable sizes. The sizes can even be 0 bits, when the operation will always be scheduled to slot “0”. This may be the case, for instance, of the first operation of a data flow graph, which must always start at slot 0. Smaller sizes reduce the memory requirements and speed up the computation.

The slots are not absolute. They are the slot in which each operation must commence relative to the latest execution slot of all the operations’ predecessors in the data flow graph:

$$\text{Starting Slot} = \lceil \text{All predecessors' last slot} \rceil + 1$$

FIGURE 57 Encoding in shared problems. For each operation a second substring identifies the relative slot in which the operation is scheduled.

This will become clearer with the example on figure 58.

Let us assume that the chromosome contains the scheduling information of table 23. Let us also assume that all adders have a delay of 30 ns and that the clock frequency is 100 MHz, which results in clock steps of 10 ns. From the data flow graph we know that the following precedence relationships exist:

- Operation 0 before operations 1, 2 and 3
- Operations 1, 2, and 3 before operation 4

TABLE 23 Scheduling information of an example chromosome.

	OPERATION				
	0	1	2	3	4
SLOT	0	4	0	2	1

FIGURE 58

Scheduling: Example of relative slots.

The information of table 23 shows that operation 0 is scheduled to slot 0. Since this operation has no predecessors, it is scheduled to the absolute slot 0 (at time = 0 ns) and uses slots 0, 1, and 2. Table 23 shows that operation 1 should be scheduled to relative slot 4. This means that it will be scheduled four slots after operation 0's completion. Therefore, operation 1 is scheduled to absolute slot 7 (time = 70 ns). Similarly, operation 2 is scheduled to absolute slot 3 (time = 30 ns) and operation 3 is scheduled to absolute slot 5 (time = 50 ns).

Next, the encoding shows that operation 4 should be scheduled to relative slot 1. Since this operation has three predecessors in the data flow graph (operations 1, 2, and 3), it will be scheduled to one slot after the latest of them. In this case, the latest operation is operation 1, which uses slots 7, 8, and 9. Therefore, operation 4 is scheduled to absolute slot 11.

Relative slots are used since absolute slots would create too many infeasible solutions, as precedence relationships are violated. For example, with absolute slots the encoding shown in table 23 would indicate that operations 2 and 3 would start before the end of operation 0 and that operation 4 would start before the end of all its predecessors. This is typical of a scheduling problem using absolute slots. In fact, the vast majority of schedules would be infeasible (as confirmed by early implementations).

The data structure for storing the scheduling information is shown on figure 56. The last field (*BFL, Bits For Location*) indicates the number of bits to use for the relative slot of each operation. Typically, all operations use the same number of bits. The first operations of the data flow graph (those without predecessors) are clear exceptions as they should be scheduled to slot 0.

The number of bits for slots affects the efficiency of the software. An exceedingly small number may cause an unnecessary use of resources as operations would tend to overlap too much, thus increasing the hardware required. An exceedingly large number would tend to produce many time-infeasible solutions as slots values tend to become greater and would also slow down the processing. These are eliminated as the algorithm progresses and the generations evolve (however, they may cause premature convergence due to “niching” [56]). In addition, a large number of bits increases the memory requirements.

Currently, the number of bits for location is determined based on the minimum and maximum possible slots of each operation. These minima and maxima are calculated at the beginning of execution and use the information regarding the slowest and fastest operators that may implement each operation. For example, let us assume that there is a choice between two types of adders to implement the operations of the data flow graph of figure 58. The delay of the adders is 10 ns and 30 ns. Then we know that operations 1, 2, and 3 cannot possibly start before 10 ns and that operation 4 cannot start before 20 ns. Let us assume that there is a timing constraint of 100 ns. Then we know that operation 4 cannot possibly start after 90 ns (anything later would render the solution infeasible). Then, operations 1, 2, and 3 cannot possibly start after 80 ns. Then operation 0 could not possibly start after 70 ns (however, this operation being the first of the data flow graph is scheduled to slot 0 beforehand). This information is shown in table 24.

TABLE 24 Minima, maxima, number of slots and bits needed.

Operation	Minimum	Maximum	Maximum Number of Slots	Bits needed
0	0	0	1	0
1	10	80	8	3
2	10	80	8	3
3	10	80	8	3
4	20	90	8	3

The table also shows the maximum number of slots needed (*mobility*):

$$\frac{\text{Maximum} - \text{Minimum}}{\text{Clock Cycle}} + 1$$

This indicates the absolute maximum number of slots in which each operation can be scheduled. The number of bits to use for location is simply the best approximation to cover all possible choices. In this example, 3 bits should be used for operations 1, 2, 3, and 4.

Note that this guarantees the existence of a feasible solution in terms of satisfying precedence relations, but it does not guarantee feasibility regarding timing constraints. However, the calculation of slots' minima and maxima allows a rapid determination of whether the time constraint is feasible or not. Obviously this does not consider feasibility regarding the other constraints (area, power).

6.2 Parameter Setting

The "genetic" parameters that the user can define are:

- PC: Probability of Crossover
- PM: Probability of Mutation
- GEN: Number of Generations
- POP: Population Size

The user can also define the number of trials for the extended initialization procedure, as explained in section 6.3.1.

6.2.1 PC: Probability of Crossover

This parameter defines the probability with which a member of the population will be crossed with another member after reproduction. Genetic algorithms are very robust regarding this parameter and, in general, any “reasonable” values will work satisfactorily. The question is how to define what is reasonable. Many GA researchers state that good values are from 10% to 30%, others say from 20% to 60%. We have found that most values in this range work well. Figure 59 shows a test in which area is optimized on the fifth-order elliptic wave filter (EWF) example with different PCs. Results are plotted for two probabilities of mutation (0.001 and 0.01) and a delay of 1,500 ns. Population size is 1,000 and the optimizations run for a fixed number of generations (30).

It is clear from the graph that the performance with very low PC rates is worse than with higher rates. The figure also clearly shows how crossover affects CPU time. This increase in CPU time with higher crossover values is not due to the crossover operation itself but to the larger number of new and different offspring introduced. These newer members have to be evaluated and this is where most of the CPU time is spent (in the objective function).

6.2.2 Probability of Mutation

This parameter defines the probability with which each bit of a chromosome string will be flipped from 0 to 1 or vice-versa. The probability is usually kept very low, at about 1 for every 1,000 bits. Figure 60 shows the effects of mutation on area minimization of the EWF example for a delay of 1,500 ns and two probabilities of crossover. Population size is 1,000 and the number of generations is 30.

The best results are obtained with probabilities between 0.01 and 0.001. Note also that, the higher the probability, the longer the optimization takes in terms of CPU seconds. This is not due to the mutation operation itself, which is extremely simple and fast, but to the increased number of new and different strings that are created. These have to be evaluated and this is where most of the CPU time is spent. With this explanation, it may be expected that the CPU times may “saturate” as at one point no more different strings can be created (i.e., saturation occurs when all strings have been altered). At this point the CPU time cannot increase. However, figure 60 shows that the CPU time decreases with very high mutation probabilities. The

(a) Transistors

(b) CPU Time

FIGURE 59

Effects of probability of crossover.

Population size is 1,000; number of generations is 30.

FIGURE 60

Effects of probability of mutation.

Saturation occurs when no more new strings are created.

FIGURE 61

Effects of number of generations.

CPU times increase linearly but quality of solutions does not.

drop is due to excessive mutation producing too many infeasible solutions (these are processed very fast with a pruning mechanism - see section 6.4 on page 124).

6.2.3 Number of Generations

CPU time increases linearly as the number of generations is increased. On the other hand, the population can evolve but only up to a certain point or a certain number of generations. Improvements on average solutions tend to “saturate” as it gets very hard or impossible to find better solutions. Effects of number of generations on average solutions and CPU times are shown on figure 61.

This is the EWF example with a delay of 1,500 ns, population size of 1,000, probability of crossover of 0.3, and probability of mutation of 0.001.

6.2.4 Population Size

CPU time is linearly proportional to population sizes, i.e., with exactly the same conditions (probabilities of crossover and mutation, number of generations, etc.). CPU time will double if the population size is doubled.

The average solution also improves with bigger populations, but the improvements become smaller. This occurs because populations tend to “saturate” for the same reasons given in the previous section. Effects of population size are shown on figure 62 for the EWF example with a delay of 1,500 ns, crossover rate of 0.3, mutation rate of 0.001 and considering three numbers of generations (20, 40, and 80).

6.3 Initial Populations

Initial populations are supported by two means: extended initial populations and population seeding.

6.3.1 Extended Initial Populations

Extended initial populations (EIP) are simply a random search performed before the first generation. Each member of the population, i.e. each solution, is randomly generated a certain number of times and the best of these is then selected to become a member of generation 0. The number of times each member is generated is the extended initialization *buffer size* and is user-defined. This means that if the population size is 1,000 and the extended buffer size is 10, a total of 10,000 members are tried before the genetic algorithm starts its work. The algorithm is shown in figure 63.

This extended initial population should produce a better population base and should improve the quality of the solutions subsequently generated. Figure 64 shows an area optimization of the EWF example for a probability of crossover of 0.3, probability of mutation of 0.001, 30 generations and a delay of 1,500 ns.

(a) Transistors

(b) CPU Time

FIGURE 62

Effects of population size.

CPU times increase linearly with the population size but the quality of the solutions does not.

```

for i=0 to POPULATION_SIZE
  Population_Member[i] = BAD_MEMBER
  for j=0 to EXTENDED_BUFFER_SIZE
    generate (Trial[j])
    if Trial[j] better than Population_Member[i]
      Population_Member[i] = Trial[j]

```

FIGURE 63

Algorithm for extended initial population.

FIGURE 64

Effects of extended initial populations. Solutions improve with a larger extended buffer size (i.e., the number of initial solutions generated).

The greater the number of initial members tried, the better the solutions will be. However, the longer the computation will be as well. It can be seen from the figure that at some point the improvement on the solutions is marginal or even nil, while the CPU times increase linearly.²

Consider extended buffer sizes of 1, 5, and 10. The data is shown on table 25.

TABLE 25 Effects of extended initialization buffer sizes.

	Extended initialization buffer size		
	1	5	10
Transistors	10074	9636	9348
CPU time (sec.)	36.6	40.6	46.8

Transistor counts are improved from 10074 (buffer size 1), to 9636 (buffer size 5), to 9348 (buffer size 10), while CPU times increase from 36.6 sec. to 40.6 sec. and 46.8 sec. respectively. In other words, increasing the extended initialization buffer size from 1 to 5 yields a 4.3% improvements in the solution but increases CPU time by 10.9%. Increasing the buffer size from 1 to 10 yields an improvement of 7.2% and a CPU time increase of 27.8%.

6.3.2 Population Seeding

In some problems with tight timing constraints it may be necessary to seed the initial population with feasible members. An easy method of generating feasible solutions consists of setting all operators to be the fastest ones available and setting all scheduling slots to 0. For example, a feasible solution should be obtained by using only carry lookahead adders and array multipliers and setting all the relative slots to 0. If the solution is not feasible, then none exists³.

This single feasible solution usually generates enough momentum to start the genetic evolution process. From this solution, better solutions are derived at each generation. Instead of

2. This extended initialization method can be regarded as purely random search. In fact, with this capability the software can be used as a random search method just by setting the number of generations to 0.

3. Regarding time-feasibility only.

FIGURE 65

Single and multiple seeds bidimensional analogy.

using a single solution, better results can be obtained by generating a number of initial feasible solutions. These are simply variations using the fastest operators with more varied scheduling slots or using the next-to-fastest operators with a combination of scheduling slots. These seeds are generated automatically by the software but they are different for every problem as they are based on whichever the fastest operators are (i.e., whether using Booth and array multipliers, or QBS and Booth, etc.).

An analogy is shown on figure 65 for a bidimensional search space. With multiple seeds the solution space is better covered from the start. Obviously, with a multidimensional problem and a very large solution space (i.e. $\sim 10^{100}$), it is impossible to cover all “corners” of the

space. However, the more varied genetic material produces a much larger number of different solutions.

6.4 Pruning

Pruning consists of discarding bad or infeasible solutions before they are fully evaluated. For example, given a time constraint, the actual delay of each solution can be easily obtained before the actual area and power values are calculated. Obtaining the delay of a solution is much faster than calculating how many operators are needed at every clock cycle or what the peak power consumption is at every cycle. These two activities involve filling and comparing data from numerous tables whereas delays are computed by following the precedence relations of the data flow graph and checking the latest (last) operation (s).

6.5 Steady-State Reproduction

Figure 66 shows the results of the optimization using the steady-state reproduction technique instead of the generational technique. After 20,000 “generations” (iterations) no improvement occurs.

6.6 Analysis

The previous graphs show how the changing of a parameter affects the efficiency of the implementation, both in terms of the quality of the solutions and the CPU time required. They do not, however, show how directly the different alternatives performs with the same CPU time. For example, figure 59 shows that using a probability of crossover of 0.7 the optimizer achieves better solutions than with a probability of 0.2 and shows that a probability of mutation of 0.01 gives better solutions than a probability of 0.0001, and so on. The CPU time graphs show that the better solutions are achieved with longer runs. This makes perfect sense, but how can we say that one parameter is better than the other? A fair way is to run all optimizations for the same amount of CPU time and then compare the results.

FIGURE 66

Results using steady-state reproduction.
Population converges after 20,000 "generations".

Figure 62, for instance, shows that a population of 4,000 achieves better solutions than a population of 1,000. However, if both populations are run for 30 seconds, the smaller population fares significantly better. Average transistor count is 10,521 for the population of 1,000 and 11,770 for the population of 4,000. Results are similar for a run time of 90 seconds: 9,433 and 10,420, respectively.

Table 26 shows the results of all the previous tests for CPU times of 30 seconds and 90 seconds.

Analyzing the table we may conclude that the probability of crossover has very little influence on the algorithm's efficiency for values over 0.2.

The algorithm is also very robust regarding the probability of mutation for any values lower than 0.01. This includes no mutation at all (probability 0.0).

TABLE 26

Results with fixed CPU times (30 seconds and 90 seconds).

Reproduction Method	Extended Buffer Size	Population Size	PM	PC	Area (transistors)	
					CPU Time (seconds)	
					30	90
GENERATIONAL	1	1,000	0.001	0.0	16,448	13,394
				0.1	11,528	9,866
				0.2	10,778	9,669
				0.3	10,566	9,326
				0.4	10,412	9,349
				0.5	10,337	9,637
				0.6	10,545	9,487
				0.7	10,443	9,487
				0.8	10,500	9,411
				0.9	10,473	9,641
				1.0	10,511	9,681
GENERATIONAL	1	1,000	0.01	0.0	13,363	10,160
				0.1	11,883	9,760
				0.2	11,404	9,553
				0.3	11,021	9,570
				0.4	11,096	9,279
				0.5	10,932	9,494
				0.6	10,819	9,348
				0.7	10,841	9,354
				0.8	10,761	9,430
				0.9	10,612	9,345
				1.0	10,524	9,341
GENERATIONAL	1	1,000	0.00000	0.3	10,487	9,691
			0.00001		10,394	9,733
			0.0001		10,597	9,811
			0.001		10,497	9,654
			0.01		10,893	9,470
			0.1		12,372	10,996

TABLE 26 Results with fixed CPU times (30 seconds and 90 seconds).

Reproduction Method	Extended Buffer Size	Population Size	PM	PC	Area (transistors)	
					CPU Time (seconds)	
					30	90
GENERATIONAL	1	1,000	0.0000	0.6	10,429	9,378
			0.00001		10,324	9,684
			0.0001		10,297	9,472
			0.001		10,440	9,424
			0.01		10,947	9,396
			0.1		12,563	11,485
GENERATIONAL	1	250	0.001	0.3	11,788	10,424
		500			10,482	9,871
		1,000			10,733	9,428
		2,000			11,324	9,930
		4,000			11,770	10,420
		8,000			13,779	10,713
GENERATIONAL	5	1,000	0.001	0.3	10,455	9,450
	10				10,663	9,273
	15				10,942	9,188
	20				11,788	9,455
STEADY-STATE	1	1,000	0.001	1.0	10,584	9,803
(RANDOM)					15,454	13,567

Population size does have an effect as the best results are obtained with populations between 1,000 and 2,000 individuals.

Extended initialization also proves useful, specially for larger CPU times. This may occur because a very low number of generations can be computed with a low CPU time (since time is spent generating the initial solutions). In particular, buffer sizes between 10 and 15 achieved slightly better results.

Steady-state reproduction is somewhat disappointing with larger CPU times (as also happened in figure 66). For a CPU time of 30 seconds the results are very close to those obtained with a generational technique. However, with a run time of 90 seconds, the steady-state solutions have approximately 5% larger areas compared with the best of the generational results.

This appears to indicate that the steady-state approach converges fast to a good solution but then it cannot get out of this solution even if much longer CPU times are allowed.

The last line of table 26 shows the results obtained by pure random search. These were obtained by performing extended initialization for the full CPU time. It is clear that the results are much worse (areas approximately 47% larger with a CPU time of 20 seconds and approximately 44% with a CPU time of 90 seconds).

6.7 Algorithm

Initially, a recursive function finds all the paths in the given data flow graph. At the same time, a precedence table is built and a list of “followers” for each operation is obtained. This is done only once at the beginning of execution. At each generation and while each solution is being decoded, the algorithm uses this list to keep track of the latest slot used by that solution. This allows the algorithm to avoid checking each path for timing constraint violations (the number of paths can be extremely large). An example of a list of followers and a list of paths is shown in figure 67.

The algorithm has been implemented with many modern features such as:

- Steady-State and Generational reproduction
- Two-point crossover
- Baker’s selection mechanism
- Population seeding
- Extended initial populations
- Pruning mechanism
- Non-duplicate objective function evaluation

These features have been described previously, except for the last item, which refers to the condition when an offspring is identical to a parent and there is no need to waste valuable CPU time evaluating the solution.

The basic algorithms are shown on figure 68. Reproduction is applied in a straightforward manner, by choosing individuals with probabilities proportional to their fitness. Two-point

List of Followers

Operation	Followers		
0	1	2	3
1	4		
2	4		
3	4		
4			

List of Paths

Path	Operations		
0	0	1	4
1	0	2	4
2	0	3	4

FIGURE 67

List of followers and list of paths.

crossover is then used to combine the “genetic” material of two randomly chosen individuals (from the highly fit reproduced pool).

6.7.1 Where Does Crossover Occur?

The crossover points are chosen to be at boundaries between operations, so that the genetic material representing each operation does not get disrupted. Mutation is then applied by switching a bit randomly.

6.8 Objective Function

The objective function is very complex due to the large number of variables to consider and to the multiple constraints. Basically, the objective is to give better fitness values to individu-

FIGURE 68

Algorithms for generational and steady-state GAs.
Evaluate_Solution() is the objective function.

als that use less area, or consume less power, or have smaller delays while still satisfying the timing/power/area requirements.

6.8.1 Calculating Transistor Counts

For non-shared problems, the circuit's area is calculated simply by adding the areas of each operator. For shared problems, the circuit's area is calculated by checking the maximum

FIGURE 69

Scheduling and peak power profile of an example.
This data flow graph has two additions and two multiplications.

number of operators of each type at every time slot. The area is then the largest total area found.

Consider the example of figure 69. It consists of two additions and two multiplications. The operations are scheduled over 14 time slots. Scanning all slots reveals that only one adder is needed since the additions do not overlap. However, from slot 4 to 9, two multiplications overlap, therefore two multipliers are needed. In this small example, both additions are implemented by adders of the same type and both multiplications are implemented by two multipliers of the same type. If different adders and multipliers are used the procedure is analogous.

6.8.2 Calculating Average Power

Average power is calculated by adding the average power values of each operation multiplied by each operation's delay and then dividing the total by the overall delay.

If P_{ADD} and P_{MUL} are the average power consumptions for the adders and multipliers and t_{ADD} and t_{MUL} the corresponding delays, then the average power consumption is calculated by:

$$P = \frac{\sum_{adders} N_{ADDi} \times P_{ADDi} \times t_{ADDi} + \sum_{multipliers} N_{MULj} \times P_{MULj} \times t_{MULj}}{delay}$$

Where N_{ADDi} and N_{MULj} represent the number of adders and multipliers of each type (i types of adders and j types of multipliers).

6.8.3 Calculating Peak Power

Peak power is calculated similarly by adding the power value from each operator's power profile at every clock cycle. In the case of figure 69 the total peak power is the same as the individual peak power of the adders and multipliers, except during slots 4 to 9. During these slots the power profiles of the two multipliers, which are shifted by one cycle, are added together.

6.8.4 Calculating Delay

Delay minimization is similar to the other types of optimizations. The total delay of each solution is calculated by checking the latest execution slot of the operations, as explained in section 6.7.

6.8.5 Multiobjective Optimization

The software allows the optimization of a weighted formula of any of the above goals. These parameters are not currently being normalized but the user can define any arbitrary weight for

each parameter. For example, (ba=0.4 bp=0.4 bk=0.2) on the command line indicates that 40% of the area value, 40% of the average power value, and 20% of the peak power will be used.

This feature is very useful as it allows optimization of one parameter considering just a touch of another. For instance, area efficient solutions can be found that also do not consume excessive peak power values. This feature also allows the use of Pareto fronts (please refer to chapter VIII).

6.8.6 Penalty Functions

The objective function uses a system of penalties for those solutions that have violated the given constraint (time, area or power), as shown in figure 70. This system of penalties allows solutions that are infeasible but close to optimal to be kept alive for subsequent generations, so that the good genetic material can still be used. Consider a solution that is optimal in area yet violates the timing constraint by 10 ns. This solution should not be discarded as it is possible that a very minor modification of the scheduling will bring it back to the feasible region. Such a solution contains very good “genetic” material that must be preserved for subsequent generations to evolve.

A typical formula for area objective functions is shown in figure 71. Consider an area minimization example in which the timing constraint of 1,500 ns was violated by 150 ns. The penalty is 40% of the area (30% fixed penalty plus 10% of variable penalty corresponding to the percentage of time violation). This is equivalent to increasing the area of the solution by 40% (reducing its fitness by 40%).

Since the software can actually consider multiple constraints (i.e., minimize area with area and average power constraints), the penalties generated by the penalty function of each constraint are added together.

Other types of penalty functions were tried, notably quadratic functions (in which the penalty is proportional to the square of the violation) and penalties based on the number of timing constraint violations (the number of paths that exceed the maximum allowed delay). These penalty functions did not behave significantly different from the linear function used.

```

Time_penalty = Time - Time_Constraint
Power_Penalty = Power - Power_Constraint
Peak_Power_Penalty = Peak_Power - Peak_Power_Constraint
If (Time_Penalty > 0) penalty += (0.3 + Time_Penalty)*Shared_Area
If (Avg_Power_Penalty > 0) penalty += (0.3 + Avg_Power_Penalty)*Shared_Area
If (Peak_Power_Penalty > 0) penalty += (0.3 + Peak_Power_Penalty)*Shared_Area
If (penalty < 0) penalty = 0
fitness = Max_Circuit_Area - Shared_Area - penalty
If (fitness < 0)    fitness = BAD_FITNESS

```

FIGURE 71

Area objective function with penalty functions.

Because the objective of genetic algorithms is to maximize, the fitness functions for area, average power, peak power, and delay are modified to:

$$Fitness = Large_Constant - area - penalty$$

$$Fitness = Large_Constant - average\ power - penalty$$

$$Fitness = Large_Constant - peak\ power - penalty$$

$$Fitness = Large_Constant - delay - penalty$$

The large constant is any large number that will be greater than any of the parameters to optimize (area, power, delay). For example, in area minimization problems, the maximum circuit's area can be used as a constant. It can be easily calculated by assuming that each operation is implemented by the largest possible operator. Since this maximum area will generally be much larger than the solutions's actual area, this has the added benefit of low selec-

FIGURE 70

Some typical penalty functions.

tion pressure. This reduces the differences in the good fitnesses in the mating pool avoiding very good solutions to take over the population (as too many instances could be introduced in subsequent generations). Table 27 shows an example with eight solutions.

This spread reduction also improves the chances of very bad solutions being reproduced, as is the case of solutions 5, 6, and 7 in the table. Although it represents an area over 20 times larger than the best solution, the differences in fitnesses are much smaller (8.8% and 14.4%). To avoid these very bad solutions from been reproduced, the fitnesses are scaled in relation to the average population fitness by using a pressure factor. Table 27 also shows the scaled fitnesses and resulting proportions. The two worst members are eliminated from the population. Note that in a real problem there will be hundreds or thousands of solutions so that the original fitness proportions (percentages) will be much smaller.

TABLE 27 Effects of large constants on fitness proportions.

#	Actual Area	Fitness under $C=80,000$	(%)	Scaled fitness with pressure factor = 1.3	Scaled%
0	1,000	49,000	14.4	6,625	26.7
1	2,000	48,000	14.1	5,625	22.7
2	3,000	47,000	13.8	4,625	18.7
3	4,000	46,000	13.6	3,625	14.6
4	5,000	45,000	13.3	2,625	10.6
5	6,000	44,000	13.0	1,625	6.6
6	20,000	30,000	8.8	0	0.0
7	20,000	30,000	8.8	0	0.0

6.9 Tools: PASSOS and Power-Profiler

Two tools were developed based on the ideas presented in this chapter: Power-Profiler and its predecessor PASSOS.

PASSOS stands for Power-Area-Speed Silicon Optimizer System and is the tool that optimizes area, average power, and delay in digit-serial applications. Power-Profiler optimizes area, average power, peak power, and delay in architectural-choice applications (those using different types of hardware for each type of operation).

6.10 Software Structure

Figure 72 shows the general flow of data in the software (for both PASSOS and Power-Profiler). Circles represent processes while rectangles represent data structures.

The creator creates and initializes the population according to the defined population size and type of initialization (extended, initial feasible, or purely random). The objective function decodes an individual's string according to the defined problem structure and computes its fitness according to the defined objectives and constraints. The aging process copies the new population into the old population simply by copying all the individuals' strings. It also checks that the desired CPU time has not elapsed. When it has elapsed it also reports the best solution found. *xover* performs crossover on two individuals according to the desired proba-

(a) Top-level view

(a) Data flow

FIGURE 72

Data flow in PASSOS and Power-Profiler.
Circles represent processes while rectangles represent data structures.

bility. **mutation** performs mutation on the bits of an individual according to the desired mutation probability.

6.11 Input Files

Input files consist of very simple commands specifying the data flow graph structure (operation types and precedence arcs), optimization goals and constraints, genetic algorithm parameters. A typical input file is shown on figure 73.

6.12 Batch Commands

The tools can also be run in batch mode which allows parameters to be defined on the command line. This is essential for the experimentation that was performed in this thesis in which a problem has to be run many times to obtain the average of the runs or to obtain the Pareto fronts (the set of noninferior solutions, please see section 8.8 on page 171 for an example). The simplest way to run Power-Profiler is as follows:

- **PP input_file output_file [optional parameters]**

The following parameters can be specified in the command line:

- **[ba=area weight]**
- **[bp=power weight]**
- **[bk=peak weight]**
- **[c=CPU seconds]**
- **[g=number of generations]**
- **[m=mutation rate]**
- **[o=optimize: area | average power | peak power | delay]**
- **[p=population size]**
- **[r=random seed]**
- **[u=pressure amount]**
- **[x=crossover rate]**
- **[web].**

```

* Application with two additions and two multiplications
.
OP0: ADD
OP1: MUL
OP2: MUL
OP3: ADD
* Precedence relations:
.
0 1
0 2
1 3
2 3
.
Default bits for adders: 1 0 4
Default bits for multipliers: 2 0 5
.
Optimize: Area
Average power limit: 1,000 mW
Peak power limit: 1,500 mW
Delay: 200 ns
Supply voltage: Mixed
.
Clock cycle: 10 ns
.
Generations: 50
Population: 1,000
Crossover: 0.6
Mutation: 0.001
.
* Optional parameters may follow

```

FIGURE 73

Sample input file.

6.13 The World-Wide-Web Interface

Power-profiler runs on the WWW (World Wide Web), a hypertext system that allows users throughout the world to access and retrieve multimedia information. Through the web, using browsers such as Netscape or Mosaic, users can run a limited version of Power-Profiler on several of the applications used in this thesis or they may even define their own data flow graph. The URL is:

<http://www.doe.carleton.ca/~rsm/Power/power.html>

Readers are encouraged to try the Web interface. A behavioral synthesis power optimization tutorial is also available at the URL:

<http://www.doe.carleton.ca/~rsm/Power/tutorial.html>

This tutorial is very much a summary of the ideas presented here with some of the figures in color.

An html page with some of the results (in color) of this thesis is available at:

<http://www.doe.carleton.ca/~rsm/Power/results.html>

The WWW GUI is shown in figure 74. Appendix A describes how to define a custom data flow graph.

FIGURE 74

The World-Wide-Web GUI.

VII Data Flow Graphs and Digit-Serial Applications

This chapter presents results of the digit-serial choice optimization methodology applied to several well-known problems [67-70]. These range in size from 23 to 48 operations. The examples used are presented in table 28. FIR is a 16-point FIR filter, EWF is the fifth-order elliptic wave filter, DCT is the discrete cosine transform.

TABLE 28 Characteristics of typical applications.

	FIR	EWF	DCT
Operations	23	34	48
Precedence relations	22	47	65
Paths	8	57	52
Typical solution space	10^{45} - 10^{60}	10^{70} - 10^{90}	10^{100} - 10^{150}

The results presented in this chapter were obtained with PASSOS, the tool that deals with digit-serial optimization. Only results for shared problems are presented (i.e., those in which operators are reused among operations). Results for non-shared applications can be found in [37].

7.1 Data Flow Graphs

The data flow graph of the fifth-order elliptic wave filter is shown on figure 75. It consists of 26 additions and 8 multiplications. This problem has been extensively studied in the literature and is probably the most used example in high-level synthesis research.

FIGURE 75

The EWF data flow graph.

FIGURE 76

The 16-point FIR filter data flow graph.

The data flow graph of the 16-point FIR filter is shown on figure 76. It consists of 15 additions and 8 multiplications.

The data flow graph of the Discrete Cosine Transform is shown on figure 77. It consists of 32 additions/subtractions and 16 multiplications.¹

1. Subtractions are considered as additions for all purposes.

FIGURE 77

The DCT data flow graph.

Applications can actually include these filters or any other circuit that can be described as a data flow graph. These applications and their word-parallel implementations are discussed in many references [67-70].

7.2 Assumptions

In the literature the delay for multipliers and adders is usually assumed to be two "clock-steps" and one "clock-step" respectively. Since the 32-bit multiplier used here has a delay of 100 ns, comparisons are made by assuming 1 clock-step = 50 ns. PASSOS' digit-serial results are plotted and then compared with the best schedules found in the literature and also with the best 32-bit solution found by PASSOS itself. As mentioned earlier, PASSOS works with real time in nanoseconds, so the real delay can be obtained by using the above relationship (e.g., a delay of 17 clock-steps corresponds to a real delay of 850 ns).

The 32-bit adder used in this thesis has a real delay smaller than that assumed in the literature. This is possible because with a clock frequency of 100 MHz the multiplier's delay is equivalent to ten clock cycles while the adder's delay can be less than five clock cycles. This obviously benefits PASSOS in terms of the quality of the solutions. However, the solutions found in the literature can be used as an upper bound for the digit-serial solutions. Furthermore, comparisons are also made between digit-serial results and full parallel results using the real delays in both cases.

It should be noted that PASSOS can be easily be used to perform the same type of scheduling that is found in the literature. This is achieved by defining the delay of a multiplier as 20 ns, the delay of an adder as 10 ns, and the clock cycle as 10 ns (or any other equally proportional relationships between the delays and the clock cycle). The areas can be defined as 1,000 transistors for each operator or whatever other value. The actual value is irrelevant if the objective is to try to find the solutions that use the fewest number of adders and multipliers (as

opposed to the number of transistors). Table 29 shows the best results found in the literature for the non-pipelined fifth-order elliptic wave filter (EWF).

TABLE 29 Best solutions in the literature.

		NO. of Adders	NO. of Multipliers
Delay	17 c-steps	3	3
	18 c-steps	2	2
	21 c-steps	2	1
	28 c-steps	1	1

Using the 2-to-1 delay relationship for adders and multipliers PASSOS was able to find all of the above solutions. On the average of 10 runs, the 17 c-step solution is found in less than 1 second of CPU time while the 18 c-step solution is found in approximately 45 seconds. The 21 c-step and 28 c-step solutions are found inconsistently but the next closest to optimal solutions (2 adders and 2 multipliers for 21 c-steps and 2 adders and 1 multiplier for 28 c-steps) are found in approximately 14 seconds and 16 seconds respectively.

The above results were obtained without tuning PASSOS to run on such a different problem. PASSOS was designed to solve problems that require much more complicated data manipulations which are still needlessly being performed on the above simplified problem. It is in the more complex problems involving simultaneous scheduling and assignment and using much finer clock resolution that PASSOS excels while the other methods would be unable to perform.

7.3 Results

This section shows results obtained by PASSOS using digit-serial operators. Whenever possible these results are compared with the best results found in the literature for traditional word-parallel implementations.

Figure 78 shows some results for the EWF. As mentioned earlier, at 17 clock-steps, the best previously reported result uses 3 multipliers and 3 adders (102,528 transistors - 312 mW)

while PASSOS' solution uses 45,954 transistors and 255 mW (a 16-bit adder, a 32-bit adder, a 32-bit multiplier, and a 32-bit multiplier), saving 55% in area and 18% in power.

At 28 c-steps, the best parallel solution uses one multiplier and one adder (34,176 transistors - 189 mW), while PASSOS reports a solution with 23,556 transistors and 153 mW (two 16-bit adders and two 16-bit multipliers), saving 31% in area and 19% in power.

With delays larger than 28 c-steps, the savings in area are even greater as it is not possible for the parallel solution to use fewer transistors (they cannot use fewer than one adder and one multiplier), while PASSOS can use operators of smaller size and still meet the timing constraints. For example, at 100 c-steps (5,000 ns), PASSOS savings in area increase to 82%! Furthermore, PASSOS reports a solution with a delay equivalent to 16 c-steps, while the fastest solution in the literature has a delay of 17 c-steps. This happens because of the unrealistic assumption (in the literature) that an adder takes one c-step². For this case, the figure also shows the PASSOS 32-bit solution, which uses real delays. The digit-serial approach saves 32% in area and 16% in power.

The dashed line labelled as PASSOS-P on figure 78 (a) corresponds to the transistor count obtained when the objective is to minimize power. It is much higher than when the objective is to minimize area (transistor count) itself. Similarly, PASSOS-A in figure 78 (b) corresponds to the power obtained when the objective is to minimize area. This shows that optimizing for one or the other gives significantly different results. For example, at 17 c-steps, optimization for power results in 255 mW while optimization for area results in 312 mW. Intermediate results can be obtained by optimizing for both area and power with weighted coefficients. For example, optimizing the formula $0.5 \text{ Power} + 0.5 \text{ Area}$ at 17 c-step results in a solution with 60,292 transistors and 282 mW.

Figure 79 shows similar results for the Discrete Cosine Transform (DCT).

Figure 80 shows results of the INU5 example (please see chapter IX). This is a significantly larger and more complex problem.

2. By using a much finer clock resolution, PASSOS can find even faster solutions with delays as low as 12-csteps.

(a) Area

(b) Power

FIGURE 78

Area and power results of the EWF.

Parallel-real time are the solutions using real 32-bit operators.

Parallel literature are the solutions from the literature that define 2 and 1 c-steps delays for multipliers and adders. PASSOS-P

corresponds to the solutions that optimize power while PAS-

SOS-A corresponds to the solutions that optimize area.

FIGURE 79

Results of the DCT example.

FIGURE 80

Results of the INU5 example.

PASSOS also allows the user to specify a single digit size (or set of digits) to use. Figure 81 shows the results when only one digit-size is allowed in the EWF (that is, all widths are the same for all units). For the same number of c-steps, it is clear that fewer transistors are needed in the digit-serial cases than in the 32-bit parallel cases. It can also be seen that for a given timing constraint, the best solution is not always found with the smallest operators.

Figure 82 shows an example of delay minimization when power constraints or area constraints are given in the finite impulse response example (FIR). Compared with the parallel implementations, smaller delays are always achieved by PASSOS. Several solutions with low power and area are also found for which there no feasible parallel (32-bit) solutions. These solutions are found by using the smaller and less power-hungry digit-serial operators.

7.4 Analysis

In conclusion, by mixing smaller and less power-hungry digit-serial operators, PASSOS is able to find a wide variety of solutions with significantly lower transistor counts and/or power consumption, or, alternatively, faster circuits with the same transistor counts and/or power of the parallel approach.

PASSOS is very easy to use. To change applications the user merely invokes the program with a different input file which describes the data flow graph, the optimization type (area, power, delay, area-power), and the constraints. No tuning of parameters is required. The user simply defines the population size and the number of generations.

Figure 83 shows the typical area and power savings (in percentage) for the EWF, DCT and INU5 applications (please refer to page 179 for a description of this example). Average area savings are in the range 30-90% while average power savings are in the range 15-30%.

FIGURE 81

Area and power with single digit operators (EWF).
All operations are performed with adders or multipliers of exactly the same digit size.

(a) Area constraint

(a) Power constraint

FIGURE 82

Delay minimization with area and power constraints (FIR). These figures show the best delay achieved with each area or power constraint.

FIGURE 83

Area and power savings of EWF, DCT, and INUS5.

VIII Architectural-Choice Applications

The previous chapter presented the results of the methodology applied to problems in which the width of each operator is selected so that silicon area or average power are optimized. In this chapter, the methodology is applied in a similar manner, but this time selecting the type (architecture) of each adder (e.g., RCA or CLA), and the type of each multiplier (e.g., BOOTH, ARR32). Peak power consumption is also considered since these modules have also been characterized for their typical peak power distributions (power profiles).

The results presented in this chapter were obtained with Power-Profiler.

8.1 Applications

Power-Profiler attempts to find the best combination of operators for a given delay and for a given optimization goal. Table 30 presents typical assignments obtained as the result of combined area/average power optimization. The numbers shown correspond to the quantity of each type of operators needed for each case.

TABLE 30 Typical results: assignments of FIR, EWF, DCT.

		RCA	CLA	QSS	BOOTH	ARR16	ARR32	Transistors	Average Power (mW)	Peak Power (mW)
FIR (1,000ns)	3.3V	1	1	4	2	0	0	19,048	47.2	120.0
	5V	0	1	0	3	0	0	13,750	113.1	175.5
EWF (1,000ns)	3.3V	1	1	0	1	0	2	73,256	150.1	229.0
	5V	0	1	0	2	0	0	9,942	117.3	301.5
DCT (1,000ns)	3.3V	2	1	0	8	0	0	35,450	70.3	139.3
	5V	1	1	0	4	0	0	18,888	227.7	578.0

8.2 Single Module Behavioral Synthesis

Conventional single module behavioral synthesis assign the operations of a data flow graph to specific modules. If only Booth multipliers are used in the EWF example, the smallest possible delay at 5V is 590 ns. There are no feasible solutions with this type of multiplier for delays under 590 ns. Array multipliers must be used if the desired delay falls within this range. Figure 84 (a) shows the average power consumption of the conventional approach obtained with a limited library (CLA for all additions with either ARR32 for all multiplications or BOOTH for all multiplications). The resulting power curves are compared with the solution that selects and mixes components from this set. All these results are obtained with Power-Profiler. Figure 84 (b) shows the corresponding transistor counts for these solutions (please note that the optimization objective is average power and not area).

The exclusive use of Booth multipliers in the delay range of 590-700 ns saves approximately 60% in power on average and about 85% in transistor counts. However, under 590 ns no feasible solutions are found.

Power-Profiler is able to bridge the curves of the Booth and array multipliers, offering the best combination of power and transistor count. It achieves this by using combinations of adders and multipliers. For example, at a delay of 570 ns, the best solution uses two Booth multipliers and one array multiplier, saving 39% in average power over the array multiplier solution.

8.3 Different Supply Voltages

The use of either 3.3V or 5V supply voltages (but not mixed) is considered in figure 85. It shows results of power optimization on the FIR and EWF examples. As expected, average power is significantly lower at 3.3V for the larger delays (i.e., slower circuits). However, with the smaller delays the 3.3V power is actually higher than at 5V. This occurs because more of the faster operators must be used when timing constraints are tighter. Array multipliers, for example, are very power-hungry even at 3.3V. Furthermore, with tight constraints peak power will also be higher at 3.3V due to the greatly increased parallelism.

(a) Average power consumption

(b) Transistor counts

FIGURE 84

Comparison with single module approach.

(a) EWF

(a) FIR

FIGURE 85

Average power of EWF and FIR at 3.3V and 5V.
Power at 3.3V can actually be higher for fast circuits.

8.4 Different Power Optimization Goals

Different optimization goals give very different results even if these goals are average power and peak power. This is shown in figure 86 (a), which shows the 3.3V EWF peak power when optimizing for average power and for peak power itself. The differences in peak power at the lower end of delays (e.g., 800 ns) is quite significant (220%). Similarly, figure 86 (b) shows the average power for both optimization goals. The largest difference in this case is approximately 110% at a delay of 1,100 ns.

8.5 Overall Peak Power Profiles

Power distribution graphs, or power profiles, along the time axis are shown in figure 87 (a) and (b) for both average and peak optimization of the DCT example with a delay of 1,000 ns at 5V. The peak power of each 10 ns clock cycle is shown. In this case optimization for average achieves a solution that consumes 227.0 mW (using 55,510 transistors) while with peak optimization this value rises to 233.1 mW (with 21,640 transistors). Although with a slightly larger average power (2.7%), it is clear that with peak optimization the power is much better distributed throughout the entire delay.

A better way to achieve results that are good in all three aspects (area, average power and peak power) is to use the triple optimization feature of Power-Profiler. For instance, if the tool is called as:

```
Power_Profiler dct out ba=0.33 bp=0.33 bk=0.33
```

then it optimizes all three parameters with equal weight. The resulting power profile is shown in figure 88. This solution uses 16,916 transistors and consumes 232.9 mW.

In these examples the small penalty in average power is well worth a much superior power distribution. The advantage of minimizing the power peaks is fairly obvious with a reduction of approximately 7% on peak power.

(a) Peak power when optimizing peak and average power

(b) Average power when optimizing peak and average power

FIGURE 86

Effects of different optimization goals.
Optimization for peak or average power give very different results.

(a) Average power optimization

(b) Peak power optimization

FIGURE 87

Peak power profiles for a delay of 1,000 ns.
Very significant reductions in the number of transistors
and peak power at a very small cost in average power.

FIGURE 88

Results of triple optimization (area/average power/peak power). Transistor count is significantly reduced at a small cost on average and peak power.

8.6 Mixing Supply Voltages

Power-Profiler is also capable of finding the best mixed-voltage solutions. Consider again the FIR and EWF examples optimized for average power, but this time comparing the results with a mixed-voltage approach. These results are shown on figure 89. At 3.3V, the minimum feasible delay for the EWF is 700 ns. Faster circuits require the use of 5V supply voltages. Solutions with much lower power consumption can be found by mixing operators of different supply voltages and combining the best of both worlds. For instance, it is clear from the figure that the mixed-voltage approach achieves considerably lower power for delays lower than 1,200 ns.

(a) EWF

(b) FIR

FIGURE 89

Mixing supply voltages.

The mixed-voltage solution always has the lowest power.

The same is true for the FIR example for delays smaller than 470 ns. Table 30 shows the assignment results of the FIR for a delay of 400 ns, using supply voltages of 3.3V, 5V, and mixed.

TABLE 31 Assignments for the FIR example with a delay of 400 ns.

		RCA	CLA	QBS	BOOTH	ARR16	ARR32	Transistors	Average Power (mW)	Peak Power (mW)
FIR (1,000ns)	3.3 V	5	1	4	3	0	5	182,016	323.2	855.0
	5 V	0	7	0	8	0	0	44,086	282.9	865.1
	Mixed	3.3 V	2	3	0	8	0	41,858	92.2	147.8
		5 V	0	2	0	0	0			

Note the sharp contrast between these results, which assume a tight timing constraint (400 ns), and the results shown in table 30, which assume a more relaxed timing constraint (1,000 ns). The average power at 3.3V is higher than at 5V with a tighter constraint, while the opposite occurs with the relaxed constraint. These results are all expected, but the impressively better results obtained with a mixed supply voltage require some explanation.

The schedules of the 3.3V and 5V cases of table 30 are shown in figure 90, while the schedule of the mixed supply voltage approach is shown in figure 91. Note that these are all optimizations for average power without directly considering area or peak power.

The mixed-voltage solution performs all multiplications at 3.3V, while doing some of the additions at 5V and others at 3.3V. The few additions at 5V provide just enough speedup to meet the 400 ns timing constraint, while a large reduction in power is achieved by lowering the supply voltage of all the multiplications.

8.7 Considering Silicon Area and Supply Voltages

The transistor counts corresponding to the solutions of figure 89 are plotted in figure 92. While the power consumption at 3.3V may be lower, its transistor count is usually higher than with a supply voltage of 5V. It is clear that the increased parallelism needed with the lower voltages causes an increase in silicon area. There are two important issues to note:

FIGURE 90

Schedule of the FIR example with a supply voltage of 3.3V.

FIR - Mixed Voltage
 Optimization objective: average power
 Average Power: 92.2 mW
 Peak Power: 147.8 mW
 Transistors: 41,858

□ CLA 5 V

▨ RCA 3.3V

□ CLA 3.3V

FIGURE 91

Schedule of the FIR example with a mixed supply voltage.

- It is possible that 3.3V operators may be too slow for the fastest timing constraints. This is the case of the EWF at a delay of 500 ns and 600 ns. With these delays there are no feasible 3.3V implementation, even using array multipliers.
- The increase in area may be too high (i.e., infeasible) to be accommodated for a particular application.

The above transistor counts are the result of optimizations for average power without considering area. It is possible to lower the transistor counts by optimizing a combination of area and average power instead. Figure 93 shows the transistor counts and average power of the FIR with a mixed supply voltage and considering the following weighted optimizations:

- 0.5 Power + 0.0 Area (i.e., the same as in figure 89)
- 0.5 Power + 0.1 Area
- 0.5 Power + 0.5 Area

(a) EWF

(b) FIR

FIGURE 92

Considering transistor counts and supply voltages. These graphs correspond to the examples of figure 89.

(a) Average power

(b) Transistor counts

FIGURE 93

Power optimizations considering area.

As the area weight is increased the number of transistors decreases but the power increases.

FIGURE 94

Schedule of the combined area-power optimization. Transistor count is much lower while average power is slightly higher.

The results show that, in general, increasing the weight for area lowers transistor counts but it increases power consumption. Figure 94 shows the schedule of the solution that considers a weight of 90% for power and 10% for area for the FIR example at 5V. The area reduction is significant (59%) while the average power increase is marginal (0.2%). It is clear that at least a small area weight should always be included in the power optimizations.

Although the methodology presented here has the capability of optimizing a weighted combination of any of the optimization goals, it is unlikely that the designer will know beforehand what are the correct weights that should be given to each component (e.g., whether it should be 40% for area and 60% for power or 60% for area and 40% for power, etc.). To consider the area-power trade-offs, a better alternative is the use of Pareto fronts.

8.8 Pareto Fronts

A Pareto front is a group of *undominated* (or *noninferior*) points in multiobjective optimization. This concept can be applied in behavioral synthesis (and in general IC design) when both silicon area and average power have to be optimized. The easiest way for this is to use a weight to each parameter and then perform the joint optimization. Pareto fronts, however, provide a full list of all possible noninferior points to the designer so that then he or she is able to choose the best options.

Pareto fronts can be easily obtained by combining the batch and the multiobjective optimization capabilities of Power-Profiler by following the approach described in [72]. The method consists of performing several optimizations, each with its own different set of weights for the parameters.

Figure 95 (a) shows the area/average power Pareto front of the EWF with a delay of 1,500 ns (5V) while (b) shows it for the DCT with delays of 500 ns and 600 ns (a mixed supply voltage is used in both examples). The set of noninferior solutions means that given a certain area there is no point with better power and given a certain power there is no point with better area than the corresponding point in the curve.

The batch file used looks as follows:

```
Power_Profiler dct out0 ba=0.1 bp=0.5
Power_Profiler dct out1 ba=0.2 bp=0.5
Power_Profiler dct out2 ba=0.3 bp=0.5
Power_Profiler dct out3 ba=0.4 bp=0.5
Power_Profiler dct out4 ba=0.5 bp=0.5
Power_Profiler dct out5 ba=0.6 bp=0.5
Power_Profiler dct out6 ba=0.7 bp=0.5
Power_Profiler dct out7 ba=0.8 bp=0.5
Power_Profiler dct out8 ba=0.9 bp=0.5
```

Pareto fronts are a great tool that offer the designer a snapshot of the universe of all noninferior solutions, so that he or she can make a trade-off between area and power.

(a) EWF with mixed supply voltage

(b) DCT with mixed supply voltage

FIGURE 95

The area-average Power Pareto fronts.
These offer the complete area-power picture to the designer.

FIGURE 96

Delay optimization with power constraints.

8.9 Optimizing Delay With a Power Constraint

Figure 96 shows an example of the delay optimization feature of Power-Profiler with a power constraint. This is the FIR example operating at 3.3V and at 5V. The 3.3V solution achieves a better delay for a low value of power (200 mW): 450 ns against 550 ns of the 5V solution.

For larger power values the situation is reversed. For example, for a power constraint of 300 mW the 3.3V solution has a delay of 420 ns, while the 5V solution has a delay of 370 ns. On the other hand, a 3.3V implementation cannot offer the smaller delay range that the 5V achieves easily (for that matter, the 3.3V solutions cannot even consume more than 600 mW).

8.10 CPU Times

Table 32 shows the cpu times needed for the EWF, FIR, and DCT examples under four timing constraints. These are averages of 10 runs and they represent the times to reach the best solutions, and solutions which are within 1% of the best.

TABLE 32 CPU times (seconds) on a SparcStation 5.

Delay	FIR		DCT		EWF	
	best	within 1%	best	within 1%	best	within 1%
800 ns	20.8	9.2	342.6	259.6	< 1.0	< 1.0
1000 ns	16.4	7.7	333.0	13.0	196.7	25.8
1600 ns	26.8	17.8	162.0	24.2	205.9	16.4
2000 ns	42.5	11.0	419.0	45.2	360.0	44.8

The average cpu times in minutes for all the timing constraints are shown in table 33.

TABLE 33 Average cpu times for all timing constraints (minutes).

	FIR	DCT	EWF
Best	0.19	5.23	1.18
Within 1%	0.44	1.42	0.37

These CPU times are very reasonable for the size and complexity of the problems. Note that the times increase as the delay is increased for each example. This happens mainly for two reasons:

- With the longer delays the tables that the software must fill and compare become larger, demanding more from the CPU.
- The total number of possible solutions and the number of feasible solutions increase with more relaxed timing constraints.

8.11 Analysis

The results presented in this chapter confirm that 3.3V solutions can consume significantly lower power than 5V solutions, at least for the more relaxed timing constraints. They also show that a large increase in silicon area may result with tighter constraints. In fact, even power consumption can be much higher at 3.3V (if, for example, power-hungry array multipliers have to be used to speedup the circuit).

It can also be concluded that mixed-voltage results are very promising as they achieve consistently and significantly better than both 3.3V and 5V over most of delay range. In the worst case they perform equal to one of the two voltages, as the mixed-voltage solutions revert to 5V solutions with fast constraints, and to 3.3V solutions with slow constraints.

This significant reduction in power consumption happens because of two major reasons:

- The 3.3V implementations must use several of its fast $\times$ operators to achieve a relatively small delay. If the data flow graph contains many multiplications this translates into several array multipliers used in parallel, which means large power consumption and silicon area. A mixed-voltage solution may use a combination of Booth multipliers operating at both 3.3V and 5V to meet the timing constraint, and at the same time lower power consumption and transistor counts.
- Mixed-voltage solutions are able to combine 5V adders with 3.3V multipliers. This saves a considerable amount of power, since the multipliers are the most power-hungry operators, and the 5V adders may provide just enough speedup to meet the timing constraints.

The second point implies that mixed-voltage solutions will behave equally well even if only the most power-efficient adders and multipliers are used. This is confirmed by figure 97, which is the same as figure 89 (b), but allowing only CLA adders for additions and only Booth multipliers for multiplications.

Even at a delay of 320 ns the power consumption of the mixed-voltage solution (347.9 mW) is lower than at 5V (353.6 mW). This is achieved by implementing a single addition at 3.3V.

A mix of 3.3V and 5V adders and multipliers is used for the delays of 350 ns and 370 ns.

3

of/de

3

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

FIGURE 97

The FIR example using only Booth multipliers and CLA adders. Curve is very similar to figure 89; the mixed-voltage approach still performs better than 3.3V and 5V.

Solutions with delays between 400 ns and 470 ns use only 3.3V multipliers, but use a mix of 3.3V and 5V adders.

The examples in this chapter also showed that optimization for average and peak power achieve very different results. Since the spectacular reductions in peak power and area observed in figures 87 and 88 came at a very small increase in average power, perhaps the most important conclusion is that the optimizations for one parameter should always consider a small amount of the other parameters.

The use of Pareto fronts has been demonstrated for those cases in which both area and power are important. These are easily obtained with Power-Profiler and give the designer important flexibility if he or she is to make a compromise between area and power. Other Pareto fronts can be obtained for other combinations:

- average power and peak power
- peak power and area
- delay and average or peak power
- delay and area
- etc.

Three-dimensional Pareto surfaces are also possible by using a similar technique in which the weights of three parameters are varied from run to run.

As a final note, figure 98 shows the schedules of the FIR example when optimized for area (top) and for peak power (bottom). The delay is 400 ns and a mixed supply voltage is used.

Optimization for area achieves a considerably smaller transistor count (59%) than when optimizing average power (figure 91) but consumes significantly more power (206%). Note also that this schedule compares favorably with the schedule of figure 94, both in terms of average power and area. This is only a reflection that the optimizations are not perfect.

Optimization for peak power also yields considerably lower peak power than when optimizing average power (25%) but uses only slightly more average power (0.7%). The schedule is very tight as there is a clear bottleneck with operations 0, 1, 8, 9, and 16 (please refer to the data flow graph in figure 76).

FIGURE 98

Schedules of the FIR example for area and peak power optimization.

IX Larger Applications

This chapter describes applications of the methodology to a couple of much larger and more complex examples. These examples are based on two unrolled versions of the fifth-order elliptic wave filter (INU3 and INU5). The performance deterioration of the tools with the larger examples is examined, both in terms of quality of the solutions found and the increased CPU times.

9.1 Unrolling Data Flow Graphs

Data flow graphs can be unrolled by duplicating the entire structure and connecting the outputs of one stage with the inputs of another. By doing so, the complexity of the problem is greatly increased as the number of operations and the number of precedence relations increase linearly, while the number of paths increases geometrically. Figure 99 shows the EWF example unrolled once, i.e., creating a two-stage unrolled filter.

Two additional examples are similarly obtained and are then used in this chapter to consider the effects of larger problems: a three-stage unrolled filter (INU3) and a five-stage (INU5) unrolled filter, represented in figure 100.

The characteristics of these examples are shown in table 34.

FIGURE 99

INU2: The Elliptic Wave Filter unrolled once.

FIGURE 100

Three-stage (INU3) and five-stage (INU5) EWF.

TABLE 34 Characteristics of unrolled filters.

	EWF	INU3	INU5
Operations	34	102	170
Precedence relations	47	157	269
Paths	57	185,193	601,692,057
Typical solution space	$10^{40}-10^{80}$	$10^{150}-10^{250}$	$10^{400}-10^{500}$

9.2 Quality and Efficiency of Solutions

The INU3 and INU5 examples were optimized for average power consumption and the results compared with the original problem under equivalent timing constraints. The results are shown on tables 35 and 36.

TABLE 35 Results for relaxed timing constraint (5V).

	EWF	INU3	INU5
Time constraint	1500 ns	4500 ns	7500 ns
Optimum (best)	78.16 mW	78.23 mW	78.76 mW
Within 5%	5.7 sec	0.6 min	2.8 min
Within 2%	6.0 sec	2.9 min	5.6 min
Within 1%	16.3 sec	7.1 min	16.8 min
Best	205.4 sec	59.0 min	49.0 min

TABLE 36 Results for tight timing constraint (5V).

	EWF	INU3	INU5
Timing constraint	550 ns	1650 ns	2750 ns
Optimum (best)	271.81 mW	252.27 mW	248.00 mW
Within 5%	22.1 sec	10.3 min	25.3 min
Within 2%	22.1 sec	15.6 min	35.4 min
Within 1%	22.1 sec	15.6 min	44.3 min
Best	45.3 sec	16.3 min	57.6 min

These tables show the results obtained for each of the three applications under a relaxed timing constraint and under a tight timing constraint. The timing constraints for the EWF are

multiplied by three for the INU3 case, and multiplied by five for the INU5 case. This assures that exactly the same solutions can potentially be obtained for all cases. The CPU times correspond to averages of 10 runs on a SparcStation 5.

For the relaxed constraint, the best solution obtained for the INU3 example is within 0.09% of the solution of the EWF case, while the best solution for the INU5 case is within 0.7%. Because the timing constraint is relaxed, it is likely that the solution obtained for the single stage EWF is the best that could have been reached for the larger problems as well. Therefore, the small differences above represent the deterioration of the solution when solving the larger problems.

The tables show the average CPU times to reach the best solution, and to reach solutions with power consumptions 1%, 2%, and 5% larger than the best (i.e., solutions that are 1%, 2%, and 5% worse than the best). CPU times increase considerably to reach the best solutions but they may be perfectly acceptable depending on the production volume. If smaller times are desired there is always the option of using a parallel GA.

There is no quality deterioration with the tight timing constraint. On the contrary, there is a significant improvement in the solutions when compared with the EWF solution: 7.2% less power for the INU3 and 8.6% less power for the INU5. These improvements are obtained because of the greater freedom to schedule operations in the case of the unrolled filters.

9.3 Some Examples

Figure 101 shows the overall power profile the INU3 example for (a) average power and (b) peak power optimization. With peak power optimization the largest power peak is reduced by 50%.

Figure 102 shows the overall power profiles of the INU5 example under (a) average power and (b) peak power optimizations. With peak power optimization the largest peak is reduced by 46%.

Finally, figure 103 (a) shows the average power consumption of the INU3 example while (b) shows the average power of the INU5 example as a function of delay. Once again, the power

(a) Optimization for average power.

(b) Optimization for peak power.

FIGURE 101

INU3 power profiles.

With peak power optimization the peaks are significantly lower.

(a) Optimization for average power.

(b) Optimization for peak power.

FIGURE 102

INU5 power profiles.

The five stages of the filter are more clearly seen when optimization is for average power.

consumption of the 3.3V is actually higher than at 5V for small delays. The advantages of the mixed-voltage solutions are also prominent.

9.4 Analysis

This chapter showed applications of Power-Profiler to the INU3 and INU5 examples. These are very large data flow graphs with 102 and 170 operations respectively.

The performance was observed under tight and relaxed timing constraints. Under tight constraints there was a quality improvement (7.2% and 8.6% less power) of the solutions when compared with equivalent constraints of the original EWF example. These improvements occur because operations on the unrolled problems have more scheduling freedom, although this also makes the optimization problem harder. Under relaxed constraints there was a small deterioration (0.7% and 0.09% more power). In this case the solutions of the EWF constitute an upper bound to the solutions of the unrolled versions, and no improvement is possible. CPU times for both approaches are quite reasonable.

Both examples were also optimized for average power under 3.3V, 5V, and mixed supply voltages. The mixed-voltage solutions always provide the lowest power, as was also shown by the results of the previous chapter.

These examples seem to indicate that Power-Profiler can efficiently handle very large and complex problems. The main current limitation appears to be random access memory (RAM). This limitation is mainly due to the recursive function that finds all the paths of the data flow graph, as the function calls itself thousands of times. This limitation can be easily eliminated simply by designing the function in a non-recursive manner.

(a) Average power on the INU3 example.

(b) Average power on the INU5 example.

FIGURE 103

Average power as a function of delay.
Again the mixed-voltage solutions have the lowest power.

X Conclusion

This chapter presents a summary of this thesis' contributions and conclusions, discusses its limitations, and describes possible future work.

10.1 Contributions

This thesis contains several contributions. It introduces a method that combines the selection of hardware components with scheduling, optimizing both processes globally and simultaneously. The methodology chooses the best unit for each operation and schedules the operations to optimize the use of resources. By using this approach, solutions with significantly smaller silicon areas and power consumption can be found. Genetic algorithms were successfully used for this complex joint assignment/scheduling optimization in a behavioral synthesis context.

The research presented in this thesis is also among the first, if not the first, that optimizes average power consumption in behavioral synthesis by considering:

- shutting down idle components
- lowering and mixing supply voltages
- using slower but less power-hungry operators

Furthermore, this thesis also introduces the minimization of peak power at the behavioral level through the use of *power profiles*. Some very interesting results have been observed from this approach. For example, it is clear that parallel operations should not start on the same clock cycle, as it is at the start where most of the peak power is generated by most multicycle operations.

Digit-serial arithmetic has been used for behavioral synthesis. It was found that significantly lower transistor counts and power consumption can be achieved by using smaller and less power-hungry operators in those paths in which the timing constraints allow. The method can also mix bit-serial, digit-serial and parallel operators.

The method also introduces supply voltage choice in behavioral synthesis. The method chooses the best supply voltage (5V or 3.3V) to satisfy system specifications while minimizing the resources needed. It is also capable of combining different supply voltages (mixed-voltage designs).

The method and tools presented can optimize a large set of desirable and realistic constraints. The following independent parameters can be considered:

- silicon area (in terms of transistor counts)
- average power consumption (in mW)
- peak power consumption (in mW)
- delay (in ns)

The same parameters can also be used as constraints (e.g., minimize area with a maximum delay and maximum peak power). Furthermore, the following three parameters can be jointly optimized (i.e., simultaneously minimized) by means of a user-defined weighted formula:

- silicon area
- average power consumption
- peak power consumption

The method and tools developed use real values for area, power, and delay, thereby eliminating restrictions imposed by the more traditional heuristic methods. By doing this, the complexity of the problem is greatly increased but more realistic assumptions about the design space are used.

In addition to using real values for the circuit parameters, the methodology and tools presented here have been applied to very large problems represented by data flow graphs with hundreds of operations. These are significantly larger than any other previous application

reported by behavioral synthesis researchers. The results are quite good both in terms of the quality of the solutions obtained and the CPU times required.

10.2 Limitations

This section discusses some of the limitations of this research.

10.2.1 Limited Modules and Architectures

A few adder and multiplier architectures were designed and then simulated in Spice. Although only two types of adders were designed, they constitute a very reasonable sample of the design space. The ripple-carry adder is among the smallest and slowest while the carry-lookahead adder is among the fastest and largest. Digit-serial versions of the RCA were also tested. Many other types exist (carry-select, carry-skip, etc.) but their use would not make a significant difference for the chosen clock frequency. The CLA needs a single clock cycle and the RCA needs two clock cycles at 5V, while they need two cycles and three cycles at 3.3V respectively. Other adder architectures would not greatly increase the choice for timing but they may offer more choice regarding silicon area and power. From the data tables used in this thesis, it is clear that multipliers are much larger and slower than adders. Therefore, multipliers are far more important than adders in most circuits, as they are the determining factor regarding speed and area.

Three types of multipliers were used, including array multipliers (and their digit-serial versions), Booth multipliers and Quasi-Serial multipliers. These offer a reasonable choice variety in terms of speed and area. An important architecture that was not represented is the family of pipelined multipliers. These can be incorporated without much effort.

Many more widely used architectures have not been tested: combined arithmetic units, ALUs, dividers, comparators, subtractors, etc. It would be interesting to see the effects of using these modules. However, the important idea of this thesis is that any hardware unit that has been characterized for delay, area and power parameters can be used in a tool such as PASSOS or Power-Profiler.

For research purposes I was very tempted to use an imaginary larger data set with imaginary properties. Such a set could define, for instance, eight types of multipliers with eight different delays, eight different areas, and eight different sets of power values. It would have made the choices much more interesting, but it would not have been realistic.

10.2.2 Effects of Clock Frequency

The effect of different clock frequencies has been mentioned earlier. A slightly different frequency can affect the efficiency of each architecture used. For example, a slightly faster clock would make the 16-bit array multiplier operate much faster than the Booth multiplier. Such a clock would also increase power consumption on the clock buffers.

10.2.3 Other Circuit Components

The modelling data used does not include important components such as interconnect area or control circuitry. Interconnect can use, in fact, the majority of a chip's area, although with today's four metal layer technologies this is not as significant. However, the ideas presented here are still valid as the tools find the best combination of adders and multipliers to meet given specifications.

Area, delay, and power estimates of these components will almost certainly have to be included in future tools.

10.2.4 Other Behavioral Functions

Some other typical behavioral synthesis functions have not been considered directly. This is the case of allocation, or the art of minimizing the number of registers, muxes and busses of a chip¹. This problem was mentioned in the latest high-level synthesis symposium, and the conclusion of some of the discussion groups (focus groups) was that whether a design uses one fewer register or mux than another is not too significant. We believe that research should be devoted instead to realistic circuitry that considers real areas, delays and power consump-

1. It should be noted that all input registers are included in the operators used in this thesis.

tion values. This thesis has shown that the technology is available today to tackle these much larger and complex problems.

10.2.5 Digit-Serial Implementations

The digit-serial implementations used are far from being efficient. Adders are of the ripple-carry type. This would make the ones using narrower words highly attractive as the same addition could be performed with much smaller area yet at almost the same speed. This advantage is offset by the realistic clock frequency used, which makes the narrower word adders highly inefficient (!) as only a small portion of the clock cycle is actually used in a useful manner.

As mentioned earlier, adders are not really that important when compared with multipliers. Furthermore, the smaller digit-serial multipliers are highly inefficient as they compare unfavorably with the Quasi-Serial multiplier.

The additional control circuitry required by those solutions that mix different word widths has not been considered. This could be significant.

10.2.6 Lower Supply Voltages

The parameterization of components with a supply voltage of 3.3 volts was obtained simply by lowering the VCC voltage in Spice. This causes the well-known linear increase in delay and quadratic reduction in power consumption. However, with more efficient processing technologies, delay increases are expected to be much smaller. Power consumption will also be affected with these technologies. Furthermore, the effects of deep submicron technologies on delay and power are only beginning to be understood.

10.2.7 Power Consumption

Power consumption estimation is a very complex and difficult task. Different input data may cause vastly different currents and, therefore, very varied power consumption values. The approach used in this thesis uses only average values obtained with a limited randomized set of input vectors. The accuracy of these averages can certainly be improved by increasing the

number of samples. A statistical model may offer even more complete results, but it would still offer no guarantee as to the real values. Furthermore, the Spice simulations are themselves a limitation for the larger operators (i.e. multipliers), as they consume a very large amount of CPU time. This limitation can be overcome by using parallel techniques (as in [72], for example).

The power profiles used in this thesis are also obtained with the above procedure. Although it was shown that the profiles may be deceived (or be deceiving) under certain circumstances, and for certain architectures, it was shown that the highest peak power values are invariably obtained during the first cycle of operation, when inputs are clocked. This crucial piece of information can certainly be used by the scheduler to avoid starting operations at exactly the same clock cycle.

10.3 Summary of Important Results and Conclusions

This thesis shows that behavioral tools can be successfully used for simultaneous scheduling and assignment in the synthesis of ICs to minimize silicon area, average power consumption, peak power consumption and delay.

As a tool to perform the optimizations, genetic algorithms are very robust and extremely versatile. The research conducted for this thesis showed that the technique allows fast and easy major modifications in the optimization parameters:

- objective function
- constraints
- problem structure

The use of genetic algorithms has been extremely successful. Although the optimization problem is highly complex, it was shown that very large applications with huge search spaces can be solved with relative ease.

Low Power

The importance of low power systems becomes obvious when we look at today's sales figures of tens of millions of dollars for portable computing and communication devices. However, the benefits of low power consumption apply not only to portable devices but also to ASICs in general. As integrated circuit feature sizes are becoming smaller and the clock frequencies are increasing, power consumption is becoming even more important. There is no doubt that high-level low-power design systems are needed now.

One of the current trends seems to be towards lower supply voltages as power drops quadratically. This idea is validated by the results presented here. However, lower voltages without highly optimizing the physical process also come at a cost. This is usually in the form of larger area particularly for high-performance circuits. The results presented here also indicate that mixed-voltage systems are very attractive, as they can combine the low power of 3.3V systems with the low transistor counts of 5V systems.

The advantages of attacking the power problem from the behavioral level are many. Lower-level tools are time consuming and very restrictive. The results presented here show that behavioral tools such as Power Profiler allow the ASIC designer:

- to find the best combination of operators from a library to meet a desired performance, while satisfying constraints (area, delay, power)
- to experiment with alternative solutions before any final implementation or even lengthy simulation (Spice simulations of array multipliers!). This allows superior exploration of the design space, since different designs can be evaluated quickly and without restrictive assumptions.
- to avoid costly redesigns by obtaining estimates and optimizing power consumption during the early stages of the design, and not after the prototypes have already been fabricated.

Power-Profiler tackles both energy consumption (through average power) and peak instantaneous power. While we used a limited set of hardware components, other operators can be easily incorporated. The tool is very flexible, fast, and easy to use.

Results from several applications showed that significant reductions in average power can be achieved by correctly choosing the operators from the designer's library. It was also shown that by considering the typical power distributions of operators, power peaks can be reduced by as much as 70%, and much better distributions along the time axis can be obtained. The results also showed that it is very cheap to reduce peak power, as a few milliwatts of extra average power are enough to significantly cut the power peaks. As mentioned earlier, reducing peak power stabilizes voltage levels, reduces electromigration and electromagnetic interference.

Although the use of power profiles is still an early science, many results presented here demonstrated that minimizing peak power as a joint optimization goal (i.e., with average power) is always beneficial and also carries a negligible cost in average power.

The use of Pareto fronts has been demonstrated for those cases in which both area and power are important. These curves show the universe of all noninferior solutions in the area-power plane. It was shown that these are easily obtained with Power-Profiler, and they give the designer important flexibility when a compromise is needed between area and power.

While we still cannot say that behavioral CAD tools always outperform human quality in the entire design cycle and for every design, they certainly are a big help to the designer in reducing not only area and power, but also design errors and ASICs' time-to-market. For these reasons, these tools are playing bigger roles in the design of ASICs.

10.4 Future Work

The main emphasis of future work should be allocated to the power consumption issues at the behavioral level. It is an extremely interesting and fascinating area that is still vastly undiscovered. The validation of the use of power profiles should be studied further. Although the concepts are valid, such as a method for minimizing the worse peak power scenario or for avoiding operations starting at the same clock cycle, better and more complete models can be developed. Furthermore, these can be incorporated with other power minimization techniques at lower levels of design (or even at higher levels by using low-power algorithms).

The work presented in this thesis can be extended to other types of operators (multipliers, adders, ALUs, etc). Ideally, an existent library of components should be used where each module would have better defined parameters (such as area in μm^2). Interconnect, storage, and control circuitry could also be considered.

The effects of clock frequency should be studied further. Higher frequencies increase the power consumed in the clock buffers, but improves the efficiency of the multicycle operations. A tool that combines the ideas presented here with one that chooses the optimum clock frequency (or even the use of multiple clocks) would also be very useful.

Bibliography

- [1] D. Gajski et al., *High-Level Synthesis, Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, 1992.
- [2] R. San Martin and J. P. Knight, "Operations Research in the High-Level Synthesis of Integrated Circuits", *Computers and Operations Research*, pp. 845-856, October 1993.
- [3] M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, pp. 301-318, Feb. 1990.
- [4] R. A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, Norwell, MA, 1991.
- [5] A.C. Parker, J. Pizarro, and M. Mlinar, "MAHA: A Program for Data Path Synthesis", *Proceedings of the 23rd Design Automation Conference*, pp. 461-466, las vegas, 1986.
- [6] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn, *Algorithmic and Register-Transfer-Level Synthesis: The System's Architect Workbench*, Kluwer Academic Publishers, Norwell, MA, 1990.
- [7] R. Jain, A. Mujumdar, A. Sharma, and H. Wang, "Empirical Evaluation of Some High-Level Synthesis Scheduling Techniques", *Proceedings of the 28th Design Automation Conference*, pp. 210-215, 1991.
- [8] C. Hwang and Y.C. Hsu, "Zone Scheduling", *IEEE Tran. on CAD of Integrated Circuits and Systems*, pp. 926-934, July 1993.
- [9] J. Lee, Y. Hsu, and Y. Lin, "A New Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis," *Proceedings of the International Conference on Computer-Aided Design*, pp. 20-23, 1989.

- [10] C.H Gebotys, "Optimal Scheduling and Allocation of Embedded VLSI Chips," *Proc. of the 29th Design Automation Conference*, pp. 116-119, 1992.
- [11] T.A. Ly and J.T. Mowchenko, "Applying Simulated Evolution to Scheduling in High-Level Synthesis," *Proc. of the 33rd Midwest Conference on Circuits and Systems*, August 1990.
- [12] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICS," *IEEE Transactions on CAD of Integrated Circuits and Systems*, pp. 671-679, June 1989.
- [13] A. Safir and B. Zavidotique, "Towards a Global Solution to High-Level Synthesis Problems," *Proc of the 1990 EURODAC*, pp. 283-288, 1990.
- [14] N. Wehn, M. Held, and M. Glesner, "A Novel Scheduling/Allocation Approach for Datapath Synthesis Based on Genetic Paradigms," *Proceedings of the IFIP TC10/ WG10.5 Workshop on Logic and Architecture Synthesis*, pp. 47-56, Paris, France, May 1990.
- [15] S. Ali, S. M. Sait, and M. S. T. Benten, "GSA: Scheduling and Allocation using Genetic Algorithm," *Proceedings of the 1994 EURODAC*, pp. 84-89, 1994.
- [16] R. Camposano and R. K. Brayton, "Partitioning Before Logic Synthesis," *Proceedings of the ICCAD'87*, pp. 237-246, Santa Clara, CA, November 1987.
- [17] S. Devadas and A.R. Newton, "Algorithms for Allocation in Data Path Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 768-781, July 1989.
- [18] R. Camposano, "Path-Based Scheduling for Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 85-93. January 1991.
- [19] Focus groups, *7th International Symposium of High-Level Synthesis*, Niagara-on-the-Lake, Ontario, Canada, May 1994.

- [20] E. F. Girzyc and J. P. Knight, "An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling", *Proceedings of the ICCD'84*, October 1984.
- [21] T. J. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Boston, MA, 1985.
- [22] J. Midwinter, *Improving Interconnect for the Behavioral Synthesis of ASICs*, M.Sc. Thesis, Carleton University, February 1988.
- [23] F.J. Kurdahi and A.C Parker, "REAL: A Program for Register Allocation," *Proceedings of the 24th Design Automation Conference*, pp. 210-215, 1987.
- [24] C.J. Tseng, and D.P. Siewiorek, "Automated Synthesis of Data Paths on Digital Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379-395, July 1986.
- [25] C. H. Gebotys and M. I. Elmasry, "A Global Optimization Approach for Architectural Synthesis," *Proceedings of ICCAD'90*, pp. 258-261, Santa Clara, CA, November 1990.
- [26] R.C Jaeger, *Introduction to Microelectronics Fabrication*, Addison-Wesley, Reading Massachusetts, pp. 138, 1988.
- [27] H. Kriplani, F. Najm, and I. Haji, "Maximum Current Estimation in CMOS Circuits," *Proc. of the 29th ACM/IEEE Design Automation Conference*, pp. 2-7, June 1992.
- [28] H. B. Huntington and A. R. Grone, *J. Phys. Chem Solids*, NO. 20, pp. 76, 1961.
- [29] D. P. Foty and E.D Nowak, "MOSFET Technology for Low-Voltage/Low-Power Applications," *IEEE Micro*, vol. 14, NO 3, pp. 68-77, June 1994.
- [30] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low Power CMOS Digital Design", *IEEE J. Solid-State Circuits*, Vol. 27, No. 4, 1992, pp. 473-483.

- [31] W. Lee, U. Ko, and P.T. Balsara, "A Comparative Study of CMOS Digital Circuit Families for Low-Power Applications," *Proc. Inter. Workshop on Low-Power Devices*, Napa Calif., pp.129-138, April, 1994.
- [32] C. Nagendra, U.K.Mehta, R.M.Owens and M.J.Irwin, "A Comparison of the Power-Delay Characteristics of CMOS Adders," *Proc. Inter. Workshop on Low-Power Devices*, Napa, CA., pp.231-236, April, 1994.
- [33] L. Benini, M. Favalli, and B. Ricco, "Analysis of Hazard Contribution to Power Dissipation in CMOS ICs," *Proc. Inter. Workshop on Low-Power Devices*, Napa, CA., pp. 27-38, April, 1994.
- [34] W. Jone and C.Fang, "Timing Optimization by Gate Resizing and Critical Path Identification," *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 135-139, June, 1993.
- [35] K. Van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schaliij, and A. Peeters, "Asynchronous Circuits for Low Power: A DCC Error Corrector", *IEEE Design & Test of Computers*, pp. 22-32, Summer 1994.
- [36] E. Olson and S.M. Kang, "Low-Power State Assignment for Finite State machines.", *Proc. Inter. Workshop on Low-Power Devices*, Napa, CA, pp. 63-68, April, 1994.
- [37] R. San Martin and J. P. Knight, "Genetic Algorithms for Optimization of Integrated Circuit Synthesis", in *Proceedings of the Fifth International Conference on Genetic Algorithms and their Applications*, 432-438, Morgan Kaufmann, San Mateo, Ca, 1993.
- [38] R. Mehra and J. Rabaey, "Behavioral Level Power Estimation and Exploration," *Proc. Inter. Workshop on Low-Power Devices*, Napa, CA., pp. 197-202, April, 1994.
- [39] R. San Martin and J. P. Knight, "PASSOS: A Different Approach for Assignment and Scheduling in High-Level Synthesis", in *Proceedings of the 37th Midwest Symposium of Circuits and Systems*, August 1994.

- [40] R. San Martin and J. P. Knight, "Power-Profiler: Optimizing ASICs Power Consumption at the Behavioral Level," *Proc. 32nd ACM/IEEE Design Automation Conference*, accepted for publication, San Francisco, June, 1995.
- [41] A.P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R.W. Brodersen, "Optimizing Power Using Transformations", *IEEE Tran. on CAD of Integrated Circuits and Systems*, vol. 14, NO. 1, pp. 12-31, January 1995.
- [42] A. P. Chandrakasan, A. Burstein, and R. W. Brodersen, "A Low Power Chipset for Portable Multimedia Applications", *Proceedings of the IEEE International Solid-State Circuits Conference*, Feb. 1994, pp. 82-83.
- [43] B. Arnold, "The Long Road to Low-Voltage systems," *ASIC and EDA: Technologies for System Design*, Verecom Corp. pp.9-18, Oct., 1993.
- [44] C. Nakata and J. Brock, *H4C Series Design Reference Guide*, Rev. 1, Motorola Inc., 1993.
- [45] J. J. F. Cavanagh, *Digital Computer Arithmetic*, McGraw-Hill Book Company, New York, NY, 1984.
- [46] E. E. Swartzlander Jr., *Computer Arithmetic*, IEEE CS Press, Los Alamitos, CA, 1990.
- [47] F.N. Najm, "A Survey of Power Estimation Techniques", *IEEE Transactions on VLSI*, pp. 446-455, Vol. 2, NO 4, December 1994.
- [48] R. Burch, F.N. Najm, P. Yang, and T. N. Trick, "A Monte Carlo Approach for Power Estimation Techniques", *IEEE Transactions on VLSI*, pp. 63-71, Vol. 1, NO 1, March 1993.
- [49] P. E. Landman and J. M. Rabaey, "Black-Box Capacitance Models for Architectural Power Analysis", *Proc. Inter. Workshop on Low-Power Devices*, Napa, CA., pp. 165-170, April, 1994.

- [50] A. D. Booth, "A Signed Binary Multiplication Technique", *Quarterly Journal of Mechanics and Applied Mathematics.*, vol. 4, part 2. pp.236-240, 1951.
- [51] S. C. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 14-17, Feb. 1964.
- [52] R. Bellman, *Dynamic Programming*, Princeton University Press, 1957.
- [53] W.L. Winston, *Operations Research, Applications and Algorithms*, PWS-Kent, 1991.
- [54] R.A. Rutenbar, "Simulated Annealing Algorithms: An Overview," *IEEE Circuits and Devices Magazine*, pp. 19-26, January 1989.
- [55] J. H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, Mi, 1975 (also MIT Press, Cambridge, MA, 1992).
- [56] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Ma, 1989.
- [57] E. J. Baker, "Reducing Bias and Inefficiency in the Selection Algorithm", in *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*, 14-21, Morgan Kaufmann, San Mateo, Ca, June 1987.
- [58] G. Syswerda, "A Study of Reproduction in Generational and Steady-State Genetic Algorithms", 94-101, in *Foundations of Genetic Algorithms*, Morgan Kaufmann, San Mateo, Ca, 1991.
- [59] D. Whitley, "GENITOR: a Different Genetic Algorithm". In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, Denver, Co, 1988.
- [60] L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reynhold, New York, NY, 1991.
- [61] G. Syswerda, "Uniform Crossover in Genetic Algorithms". In *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, 2-9, Morgan Kaufmann, San Mateo, Ca, June 1989.

- [62] R. Tanese, "Parallel Genetic Algorithm for a Hypercube", in *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*, 177-183, Morgan Kaufman, San Mateo, Ca, 1987.
- [63] J. R. Koza, *Genetic Programming*, The MIT Press, Cambridge, Ma, 1992.
- [64] J.R. Koza, "Evolving a Computer Program to Generate Random Numbers Using the Genetic Programming Paradigm", in *Proceedings of the Fourth International Conference on Genetic Algorithms and their Applications*, 37-44, Morgan Kaufmann, San Mateo, Ca, 1991.
- [65] W. A. Tackett, "Genetic Programming for Feature Discovery and Image Discrimination", in *Proceedings of the Fifth International Conference on Genetic Algorithms and their Applications*, 303-309, Morgan Kaufmann, San Mateo, Ca, 1993.
- [66] R. E. Smith, D. E. Goldberg, and J. A. Earickson, "SGA-C: A C Language Implementation of a Simple Genetic Algorithm", *TCGA Report No. 91002*, The Clearinghouse for Genetic Algorithms, University of Alabama, Tuscaloosa, AL, 1991.
- [67] S.Y. Kung, H.J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, pp. 258-264, 1985.
- [68] I.C. Park and C.M. Kyung, "FAMOS: An Efficient Scheduling Algorithm for High-Level Synthesis", *IEEE Tran. on CAD of Integrated Circuits and Systems*, pp. 1437-1448, October 1993.
- [69] C.T. Hwang, Y.C. Hsu, and Y.L. Lin, "PLS: A Scheduler for Pipeline Synthesis", *IEEE Tran. on CAD of Integrated Circuits and Systems*, pp. 1279-1286, September 1993.
- [70] J.A. Nestor and G. Krishnamoorthy, "SALSA: A New Approach to Scheduling with Timing Constraints", *IEEE Tran. on CAD of Integrated Circuits and Systems*, pp. 1107-1122, August 1993.

- [71] R. de Neufville, *Applied System Analysis*, McGraw Hill Publishing Company, New York, chapter 8, 1990.
- [72] C.R. Pon, R. Saleh, and T. Kwasniewski, "Distributed Circuit Simulation Using Waveform Relaxation in a Slotted-Ring Architecture," *Proceedings of the 1994 Canadian Conference on Electrical and Computer Engineering*, pp. 545-548, September 1994.

Appendix A The World Wide Web Interface

This appendix describes how to define a data flow graph on the Power-Profiler World Wide-Web (WWW) interface.

A.1 GUI

The graphical user interface for problems that do not require a custom data flow graph definition was shown on figure 74. The interface is very simple and only requires the user to click-and-select the parameters he or she desires. The selectable parameters are:

- probability of crossover**
- probability of mutation**
- population size**
- number of generations**
- supply voltage**
- CPU time**
- example to use**

After defining the parameters, which are all optional, the user simply clicks on the GO button. After the specified time the web server will return the results indicating the choice of adders and multipliers, the transistor count, average and peak power consumption of the best solution found.

Currently this WWW interface is a limited version of what Power-Profiler can do. Some limitations have been imposed to prevent excessive CPU utilization. Error checking is also very limited.

File Options About Help

Document Title:

Document URL:

To define a data flow graph, please specify whether the operations are additions or multiplications:

0	1	2	3	4	5	6	7	8	9
☐ +	☐ +	☐ +	☐ +	☐ +	☐ +	☐ +	☐ +	☐ +	☐ +
☐ *	☐ *	☐ *	☐ *	☐ *	☐ *	☐ *	☐ *	☐ *	☐ *

Next, please specify the connections between operations (i.e., the edges of the data flow graph):

☐ 0-1 ☐ 0-2 ☐ 0-3 ☐ 0-4 ☐ 0-5 ☐ 0-6 ☐ 0-7 ☐ 0-8 ☐ 0-9

☐ 1-2 ☐ 1-3 ☐ 1-4 ☐ 1-5 ☐ 1-6 ☐ 1-7 ☐ 1-8 ☐ 1-9

☐ 2-3 ☐ 2-4 ☐ 2-5 ☐ 2-6 ☐ 2-7 ☐ 2-8 ☐ 2-9

☐ 3-4 ☐ 3-5 ☐ 3-6 ☐ 3-7 ☐ 3-8 ☐ 3-9

☐ 4-5 ☐ 4-6 ☐ 4-7 ☐ 4-8 ☐ 4-9

☐ 5-6 ☐ 5-7 ☐ 5-8 ☐ 5-9

☐ 6-7 ☐ 6-8 ☐ 6-9

☐ 7-8 ☐ 7-9

☐ 8-9

When you have finished setting the parameters, press here to GO.

[Back](#) [Forward](#) [Home](#) [Reload](#) [Open...](#) [Save As...](#) [Clone](#) [New Window](#)

FIGURE 104

GUI for custom data flow graph definition.

A.2 Defining a Data Flow Graph

The GUI for data flow graph definition is shown in figure 104.

The process of defining a data flow graph is very simple, consisting of two definitions:

- the operation type (addition or multiplication) of each operation of the data flow graph
- the connections, or edges, in the data flow graph

To define the type of operation the user merely clicks on the appropriate box for each operation. The only restriction is that if a data flow graph has n operations, they must be defined sequentially from $OP0$ to OPN (i.e., without skipping a definition).

To define the edges, the user clicks on the boxes corresponding to each edge or connection in the DFG. For example, if operation 1 is preceded by operation 0 then the box labelled 0-1 should be selected. Figure shows an example which corresponds to the data flow graph of figure 2.

0	1	2	3	4	5	6	7	8	9
◆ +	◆ +	◆ +	◆ +	◆ +	◆ +	◆ +	◆ +	◆ +	◆ +
◆ *	◆ *	◆ *	◆ *	◆ *	◆ *	◆ *	◆ *	◆ *	◆ *

■ 0-1	■ 0-2	□ 0-3	□ 0-4	□ 0-5	□ 0-6	□ 0-7	□ 0-8	□ 0-9
□ 1-2	■ 1-3	□ 1-4	□ 1-5	□ 1-6	□ 1-7	□ 1-8	□ 1-9	
■ 2-3	□ 2-4	□ 2-5	□ 2-6	□ 2-7	□ 2-8	□ 2-9		
□ 3-4	□ 3-5	□ 3-6	□ 3-7	□ 3-8	□ 3-9			
□ 4-5	□ 4-6	□ 4-7	□ 4-8	□ 4-9				
□ 5-6	□ 5-7	□ 5-8	□ 5-9					
□ 6-7	□ 6-8	□ 6-9						
□ 7-8	□ 7-9							
□ 8-9								

FIGURE 105

DFG definition corresponding to figure 2.

A.3 Limits

Currently, the user can only define up to 10 operations. This is not due to a restriction of Power-Profiler, as it can handle DFGs with hundreds of operations (limited only by the RAM of the web server). The purpose is only to show how easy it is to enter a DFG specification.

The current web server is the machine Ro, a Sun Sparcstation 2. This machine is not the fastest, and its speed is even further reduced since it is shared by multiple users. Therefore, the performance obtained is far from optimal. Also note that the web server can be changed at any time by the system administrator.

To avoid overly bad performances, the CPU times that the user can select are those spent solely by the user's job, and not real time. However, the performance would still be much better on a faster machine.

END

1 4 1 1 9 5

FIN