

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2963

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Richard Sharp

Higher-Level Hardware Synthesis



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Author

Richard Sharp
Intel Research Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
E-mail: richard.sharp@intel.com

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at <<http://dnb.ddb.de>>.

CR Subject Classification (1998): B, C.1, D.2, D.3, F.3

ISSN 0302-9743

ISBN 3-540-21306-6 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag is a part of Springer Science+Business Media
springeronline.com

© Springer-Verlag Berlin Heidelberg 2004
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10988206 06/3142 5 4 3 2 1 0

For Kate

Preface

In the mid 1960s, when a single chip contained an average of 50 transistors, Gordon Moore observed that integrated circuits were doubling in complexity every year. In an influential article published by *Electronics Magazine* in 1965, Moore predicted that this trend would continue for the next 10 years. Despite being criticized for its “unrealistic optimism,” Moore’s prediction has remained valid for far longer than even he imagined: today, chips built using state-of-the-art techniques typically contain several million transistors. The advances in fabrication technology that have supported Moore’s law for four decades have fuelled the computer revolution. However, this exponential increase in transistor density poses new design challenges to engineers and computer scientists alike. New techniques for managing complexity must be developed if circuits are to take full advantage of the vast numbers of transistors available.

In this monograph we investigate both (i) the design of high-level languages for hardware description, and (ii) techniques involved in translating these high-level languages to silicon. We propose SAFL, a first-order functional language designed specifically for behavioral hardware description, and describe the implementation of its associated silicon compiler. We show that the high-level properties of SAFL allow one to exploit program analyses and optimizations that are not employed in existing synthesis systems. Furthermore, since SAFL fully abstracts the low-level details of the implementation technology, we show how it can be compiled to a range of different design styles including fully synchronous design and globally asynchronous locally synchronous (GALS) circuits.

We argue that one of the problems with existing high-level hardware synthesis systems is their “black-box approach”: high-level specifications are translated into circuits without any human guidance. As a result, if a synthesis tool generates unsuitable designs there is very little a designer can do to improve the situation. To address this problem we show how source-to-source transformation of SAFL programs “opens the black-box,” providing a common language in which users can interact with synthesis tools whilst exploring the different architectural tradeoffs arising from a single SAFL specification. We demonstrate this design methodology by presenting a number of transformations that facili-

tate resource-duplication/sharing and hardware/software co-design as well as a number of scheduling and pipelining tradeoffs.

Finally, we extend the SAFL language with (i) π -calculus style channels and channel-passing, and (ii) primitives for *structural*-level circuit description. We formalize the semantics of these languages and present results arising from the generation of real hardware using these techniques.

This monograph is a revised version of my Ph.D. thesis which was submitted to the University of Cambridge Computer Laboratory and accepted in 2003. I would like to thank my supervisor, Alan Mycroft, who provided insight and direction throughout, making many valuable contributions to the research described here. I am also grateful to the referees of my thesis, Tom Melham and David Greaves, for their useful comments and suggestions. The work presented in this monograph was supported by (UK) EPSRC grant GR/N64256 “A Resource-Aware Functional Language for Hardware Synthesis” and AT&T Research Laboratories Cambridge.

December 2003

Richard Sharp

Contents

1	Introduction	1
1.1	Hardware Description Languages	1
1.2	Hardware Synthesis	7
1.2.1	High-Level Synthesis	8
1.3	Motivation for Higher Level Tools	14
1.3.1	Lack of Structuring Support	14
1.3.2	Limitations of Static Scheduling	15
1.4	Structure of the Monograph	16
2	Related Work	19
2.1	Verilog and VHDL	19
2.2	The Olympus Synthesis System	23
2.2.1	The HardwareC Language	23
2.2.2	Hercules	25
2.2.3	Hebe	25
2.3	Functional Languages	26
2.3.1	μ FP: An Algebra for VLSI Specification	26
2.3.2	Embedding HDLs in General-Purpose Functional Languages	28
2.4	Term Rewriting Systems	30
2.5	Occam/CSP-Based Approaches	31
2.5.1	Handel and Handel-C	31
2.5.2	Tangram and Balsa	31
2.6	Synchronous Languages	33
2.7	Summary	34
3	The SAFL Language	35
3.1	Motivation	35
3.2	Language Definition	36
3.2.1	Static Allocation	37
3.2.2	Integrating with External Hardware Components	37
3.2.3	Semantics	38

3.2.4	Concrete Syntax	38
3.3	Hardware Synthesis Using SAFL	41
3.3.1	Automatic Generation of Parallel Hardware	42
3.3.2	Resource Awareness	42
3.3.3	Source-Level Program Transformation	44
3.3.4	Static Analysis and Optimisation	47
3.3.5	Architecture Independence	48
3.4	Aside: Dealing with Mutual Recursion	48
3.4.1	Eliminating Mutual Recursion by Transformation	49
3.5	Related Work	50
3.6	Summary	50
4	Soft Scheduling	51
4.1	Motivation and Related Work	52
4.1.1	Translating SAFL to Hardware	54
4.2	Soft Scheduling: Technical Details	55
4.2.1	Removing Redundant Arbiters	56
4.2.2	Parallel Conflict Analysis (PCA)	56
4.2.3	Integrating PCA into the FLaSH Compiler	58
4.3	Examples and Discussion	58
4.3.1	Parallel FIR Filter	58
4.3.2	Shared-Memory Multi-processor Architecture	59
4.3.3	Parallel Tasks Sharing Graphical Display	61
4.4	Program Transformation for Scheduling and Binding	62
4.5	Summary	63
5	High-Level Synthesis of SAFL	65
5.1	FLaSH Intermediate Code	66
5.1.1	The Structure of Intermediate Graphs	67
5.1.2	Translation to Intermediate Code	71
5.2	Translation to Synchronous Hardware	73
5.2.1	Compiling Expressions	73
5.2.2	Compiling Functions	75
5.2.3	Generated Verilog	79
5.2.4	Compiling External Functions	80
5.3	Translation to GALS Hardware	81
5.3.1	A Brief Discussion of Metastability	81
5.3.2	Interfacing between Different Clock Domains	83
5.3.3	Modifying the Arbitration Circuitry	85
5.4	Summary	86
6	Analysis and Optimisation of Intermediate Code	87
6.1	Architecture-Neutral versus Architecture-Specific	87
6.2	Definitions and Terminology	88
6.3	Register Placement Analysis and Optimisation	88
6.3.1	Sharing Conflicts	89

6.3.2	Technical Details	91
6.3.3	Resource Dependency Analysis	92
6.3.4	Data Validity Analysis	93
6.3.5	Sequential Conflict Register Placement	95
6.4	Extending the Model: Calling Conventions	97
6.4.1	Caller-Save Resource Dependency Analysis	97
6.4.2	Caller-Save Permanisation Analysis	99
6.5	Synchronous Timing Analysis	99
6.5.1	Technical Details	100
6.5.2	Associated Optimisations	101
6.6	Results and Discussion	104
6.6.1	Register Placement Analysis: Results	104
6.6.2	Synchronous Timing Optimisations: Results	109
6.7	Summary	110
7	Dealing with I/O	113
7.1	SAFL+ Language Description	113
7.1.1	Resource Awareness	115
7.1.2	Channels and Channel Passing	115
7.1.3	The Motivation for Channel Passing	117
7.2	Translating SAFL+ to Hardware	118
7.2.1	Extending Analyses from SAFL to SAFL+	120
7.3	Operational Semantics for SAFL+	121
7.3.1	Transition Rules	124
7.3.2	Semantics for Channel Passing	124
7.3.3	Non-determinism	126
7.4	Summary	126
8	Combining Behaviour and Structure	129
8.1	Motivation and Related Work	129
8.2	Embedding Structural Expansion in SAFL	130
8.2.1	Building Combinatorial Hardware in Magma	130
8.2.2	Integrating SAFL and Magma	134
8.3	Aside: Embedding Magma in VHDL/Verilog	136
8.4	Summary	138
9	Transformation of SAFL Specifications	141
9.1	Hardware Software CoDesign	142
9.1.1	Comparison with Other Work	142
9.2	Technical Details	143
9.2.1	The Stack Machine Template	144
9.2.2	Stack Machine Instances	144
9.2.3	Compilation to Stack Code	146
9.2.4	The Partitioning Transformation	148
9.2.5	Validity of Partitioning Functions	148
9.2.6	Extensions	149

9.3	Transformations from SAFL to SAFL+	151
9.4	Summary	153
10	Case Study	155
10.1	The SAFL to Silicon Tool Chain	155
10.2	DES Encrypter/Decrypter	160
10.2.1	Adding Hardware VGA Support	162
10.3	Summary	167
11	Conclusions and Further Work	169
11.1	Future Work	170
Appendix		
A	DES Encryption/Decryption Circuit	171
B	Transformations to Pipeline DES	177
C	A Simple Stack Machine and Instruction Memory	181
	References	185
	Index	193

List of Figures

1.1	A diagrammatic view of a circuit to compute $3.x.u.dx$	3
1.2	RTL code for a 3-input multiplexer	4
1.3	RTL code for the control-unit	5
1.4	RTL code to connect the components of the multiplication example together	6
1.5	A netlist-level Verilog specification of a 3-bit equality tester	7
1.6	Circuit diagram of a 3-bit equality tester	7
1.7	A categorisation of HLS systems and the synthesis tasks performed at each level of the translation process	8
1.8	Dataflow graph for expression: $12x + x^2 + y^3$	9
1.9	The results of scheduling and binding	10
1.10	(<i>left</i>) the dependencies between operations for an expression of the form $xy + z$. Operations are labelled with letters (a)–(e); (<i>centre</i>) an ASAP Schedule of the expression for a single adder and a single multiplier. (<i>right</i>) a List Schedule under the same resource constraints	11
2.1	VHDL code for a D-type flip-flop	22
2.2	Verilog code for the <code>confusing_example</code>	22
2.3	Running the <code>confusing_example</code> module in a simulator	22
2.4	HardwareC’s structuring primitives	24
2.5	The geometrical (circuit-level) interpretation of some μ FP combining forms. (<i>i</i>) $(/Lf)\langle x_1, x_2, \dots, x_n \rangle$; (<i>ii</i>) $(/Rf)\langle x_1, x_2, \dots, x_n \rangle$; (<i>iii</i>) $(\alpha f)\langle x_1, x_2, \dots, x_n \rangle$	27
2.6	The hardware-level realisation of the μ combinator— (<i>i</i>) function $f : \alpha \times \beta \rightarrow \gamma \times \beta$; (<i>ii</i>) the effect of applying the μ combinator, yielding a function $\mu f : \alpha \rightarrow \gamma$	28
2.7	Behavioural interpretation of basis functions AND, OR and NOT	28
2.8	Structural interpretation of basis functions AND, OR and NOT	29
3.1	A big-step transition relation for SAFL programs	39
3.2	Translating the <code>case</code> statement into core SAFL	41
3.3	Translating let barriers “---” into core SAFL	41
3.4	SAFL’s primitive operators	42
3.5	The SAFL Design-Flow	43
3.6	An application of the <i>unfold</i> rule to unroll the recursive structure one level	45

3.7	An application of the <i>abstraction</i> rule to <code>mult2</code>	46
3.8	The result of applying <i>fold</i> transformations to <code>mult3</code>	46
3.9	Three methods of implementing inter-block data-flow and control-flow	47
4.1	A Comparison Between Soft Scheduling and Soft Typing	52
4.2	A hardware design containing a memory device shared between a DMA controller and a processor	54
4.3	A table showing the expressivity of various scheduling methods	54
4.4	A structural diagram of the hardware circuit corresponding to a shared function, f , called by functions g and h . Data buses are shown as thick lines, control wires as thin lines	55
4.5	$\mathcal{C}[[e]]$ is the set of non-recursive calls which may occur as a result of evaluating expression e	57
4.6	$\mathcal{A}[[e]]$ returns the conflict set due to expression e	58
4.7	A SAFL description of a Finite Impulse Response (FIR) filter	59
4.8	Extracts from a SAFL program describing a shared-memory multi-processor architecture	60
4.9	The structure of a SAFL program consisting of several parallel tasks sharing a graphical display	61
4.10	A SAFL specification which computes the polynomial expression $12x + x^2 + y^3$ whilst respecting the binding and scheduling constraints shown in Figure 1.9	63
5.1	Structure of the FLaSH Compiler	66
5.2	Example intermediate graph	67
5.3	Nodes used in intermediate graphs	68
5.4	Translation of conditional expression: <code>if e_1 then e_2 else e_3</code>	70
5.5	Intermediate graph representing the body of <code>fun $f(x) = x+3$</code>	72
5.6	Expressions and Functions	73
5.7	Hardware blocks corresponding to <code>CONDITIONAL_SPLIT</code> (left) and <code>CONDITIONAL_JOIN</code> (right) nodes	74
5.8	Hardware block corresponding to a <code>CONTROL_JOIN</code> node	75
5.9	How to build a synchronous reset-dominant SR flip-flop from a D-type flip-flop	75
5.10	A Block Diagram of a Hardware Functional-Unit	77
5.11	The Design of the External Call Control Unit (ECCU)	78
5.12	The Design of a Fixed-Priority Synchronous Arbiter	79
5.13	The Design of a Combinatorial Priority Encoder with 4 inputs. (Smaller input numbers have higher priorities)	79
5.14	A dual flip-flop synchroniser. Potential metastability occurs at the point marked “M”. However, the probability of the synchroniser’s output being in a metastable state is significantly reduced since any metastability is given a whole clock cycle to resolve	82
5.15	An inter-clock-domain function call	83

5.16	Building an asynchronous RS latch out of two D-Type flip-flops with asynchronous resets (<code>c1r</code>)	85
5.17	Extending the inter-clock-domain call circuitry with an explicit arbiter release signal	85
6.1	A sequential conflict (left) and a parallel conflict (right). The horizontal dotted lines show the points where data may become invalid. These are the points where permanising registers are required	90
6.2	We insert permanisors on data-edges using this transformation. The dashed data-edges represent those which do not require permanisors; the solid data-edges represent those which do require permanisors	91
6.3	The nodes contained in the highlighted threads are those returned by $\pi(n, s_i)$	95
6.4	Diagrammatic explanation of $Succ_c^*(n) \cap Pred_c(n')$	96
6.5	Summary: Register Placement for Sequential Conflicts	98
6.6	Synchronous Timing Analysis	102
6.7	A block diagram of a circuit-level implementation of 3 parallel threads. Suppose that our analysis has detected that the “done” control outputs of the 3 threads will be asserted simultaneously. Thus we have no need for a <code>CONTROL_JOIN</code> NODE. Since signals “c_out1” and “c_out3” are no longer connected to anything we can optimise away the control circuitry of the shaded blocks	103
6.8	How various paramaters (area, number of permanisors, number of cycles, clock speeds and computation time) vary as the degree of resource sharing changes	105
6.9	SAFL programs with different degrees of resource sharing	106
6.10	Number of Permanising Registers	107
6.11	Chip area (as %-use of FPGA)	108
6.12	Number of clock cycles required for computation	108
6.13	Clock Speeds of Final Design	108
6.14	Time taken for design to perform computation	109
7.1	The abstract syntax of SAFL+ programs, p	114
7.2	Illustrating Channel Passing in SAFL+	116
7.3	Using SAFL+ to describe a lock explicitly	117
7.4	A Channel Controller. The synchronous RS flip-flops (R-dominant) are used to latch pending requests (represented as 1-cycle pulses). Static fixed priority selectors are used to arbitrate between multiple requests. The three data-inputs are used by the three writers to put data onto the bus	119
7.5	(i) A <code>READ</code> node connected to three channels; (ii) A <code>WRITE</code> node connected to two channels. The component marked <code>DMX</code> is a demultiplexer which routes the control signal to one of the three channels depending on the value of its select input (<code>ChSel</code>)	120

7.6	Extending PCA to deal with channel reads and writes	121
7.7	The Syntax of Program States, P , Evaluation States, e , and values, v	122
7.8	Structural congruence and structural transitions	123
7.9	A context, \mathbb{E} , defining which sub-expressions may be evaluated in parallel	124
7.10	Transition Rules for SAFL+	125
8.1	The definition of the BASIS signature (from the Magma library) . .	132
8.2	A simple ripple-adder described in Magma	133
8.3	A diagrammatic view of the steps involved in compiling a SAFL/Magma specification	134
8.4	A simple example of integrating Magma and SAFL into a single specification	135
9.1	A diagrammatic view of the partitioning transformation	144
9.2	The instructions provided by our stack machine	145
9.3	Compiling SAFL into Stack Code for Execution on a Stack Machine Instance	147
9.4	Top-level pipelining transformation	152
10.1	Using the FLaSH compiler to compile a SAFL specification to RTL Verilog	156
10.2	Using the RTL-synthesis tool <i>Leonardo</i> to map the Verilog generated by the FLaSH compiler to a netlist	157
10.3	Using the <i>Quartus II</i> package to map the netlist onto an Altera Apex-II FPGA	158
10.4	Using the <i>ModelSim</i> package to simulate FLaSH-generated code at the RTL-level	159
10.5	The Altera “Excalibur” Development Board containing an Apex-II FPGA with our simple VGA interface connected via ribbon cable	161
10.6	The Altera Development Board driving a test image onto a VGA monitor	163
10.7	The SAFL DES block connected to the VGA signal generation circuitry	164
10.8	The definition of function <code>write_hex</code>	165
10.9	Displaying the DES circuits inputs and outputs on a monitor whenever a micro-switch is pressed	166
10.10	A screenshot of the DES circuit displaying its inputs and outputs on a VGA monitor	166