Lecture Notes in Computer Science 2963
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Richard Sharp

Higher-Level Hardware Synthesis

Springer

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

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 μFP: 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 verses 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 (*left*) the dependencies between operations for an expression of the form $xy + z$. Operations are labelled with letters (a)–(e); (*centre*) an ASAP Schedule of the expression for a single adder and a single multiplier. (*right*) 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 `confusing_example` 22
2.3 Running the `confusing_example` module in a simulator 22
2.4 HardwareC's structuring primitives 24
2.5 The geometrical (circuit-level) interpretation of some μFP combining forms. (*i*) $(/Lf)\langle x_1, x_2, \ldots, x_n\rangle$; (*ii*) $(/Rf)\langle x_1, x_2, \ldots, x_n\rangle$; (*iii*) $(\alpha f)\langle x_1, x_2, \ldots, x_n\rangle$ 27
2.6 The hardware-level realisation of the μ combinator— (*i*) function $f : \alpha \times \beta \rightarrow \gamma \times \beta$; (*ii*) 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 `case` 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 *unfold* rule to unroll the recursive structure one level 45

3.7 An application of the *abstraction* rule to `mult2` 46
3.8 The result of applying *fold* transformations to `mult3` 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: `if` e_1 `then` e_2 `else` e_3 70
5.5 Intermediate graph representing the body of `fun f(x) = x+3` 72
5.6 Expressions and Functions 73
5.7 Hardware blocks corresponding to CONDITIONAL_SPLIT (left) and CONDITIONAL_JOIN (right) nodes 74
5.8 Hardware block corresponding to a CONTROL_JOIN 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 (`clr`) 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 $Succ_c^\star(n) \cap Pred_c(n')$ 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 CONTROL_JOIN 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 READ node connected to three channels; (*ii*) A WRITE node connected to two channels. The component marked DMX is a demultiplexer which routes the control signal to one of the three channels depending on the value of its select input (ChSel) 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 *Leonardo* to map the Verilog generated by the FLaSH compiler to a netlist 157
10.3 Using the *Quartus II* package to map the netlist onto an Altera Apex-II FPGA 158
10.4 Using the *ModelSim* 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 `write_hex` 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

1

Introduction

In 1975 a single Integrated Circuit contained several hundred transistors; by 1980 the number had increased to several thousand. Today, designs fabricated with state-of-the-art VLSI technology often contain several million transistors.

The exponential increase in circuit complexity has forced engineers to adopt higher-level tools. Whereas in the 1970s transistor and gate-level design was the norm, during the 1980s Register Transfer Level (RTL) Hardware Description Languages (HDLs) started to achieve wide-spread acceptance. Using such languages, designers were able to express circuits as hierarchies of components (such as registers and multiplexers) connected with wires and buses. The advent of RTL-synthesis led to a dramatic increase in productivity since, for some classes of design, time consuming tasks (such as floor-planning and logic synthesis) could be performed automatically.

More recently, *high-level synthesis* (sometimes referred to as *behavioural synthesis*) has started to have an impact on the hardware design industry. In the last few years commercial tools have appeared on the market enabling high-level, imperative programming languages (referred to as *behavioural languages* within the hardware community) to be compiled directly to hardware. Since current trends predict that the exponential increase in transistor density will continue throughout the next decade, investigating higher-level tools for hardware description and synthesis will remain an important field of research.

In this monograph we argue that there is scope for higher-level Hardware Description Languages and, furthermore, that the development of such languages and associated tools will help to manage the increasing size and complexity of modern circuits.

1.1 Hardware Description Languages

Hardware description languages are often categorised according to the level of abstraction they provide. We have already hinted at this taxonomy in the previous section. Here we describe their classification in more detail, giving concrete examples of each style.

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 1–18, 2004.

As a running example we consider designing a circuit to solve the differential equation $y'' + 3xy' + 3y = 0$ by the *forward Euler method* in the interval $[0, a]$ with step-size dx and initial values $x(0) = x$; $y(0) = y$; $y'(0) = u$. This example is similar to one proposed by Paulin and Knight in their influential paper on High-Level Synthesis [119]. It has the advantage of being small enough to understand at a glance yet large enough to allow us to compare and contrast the important features of the different classes of HDL.

Behavioural Languages

Behavioural HDLs focus on algorithmic specification and attempt to abstract as many low-level implementation issues as possible. Most behavioural HDLs support constructs commonly found in high-level, imperative programming languages (assignment, sequencing, conditionals and iteration). We discuss specific behavioural languages at length in Chapter 2; this section illustrates the key points of behavioural HDLs with reference to a generic, C-like language. In such a language our differential equation solver can be coded as follows:

```
while (x<a) {

  // Compute new values for x, y and u
  x1 = x + dx;
  u1 = u - (3*x*u*dx) - (3*y*dx);
  y1 = y + u*dx;

  // Update values of x, y and u
  x = x1; u = u1; y = y1;
}
```

Note that although this specification encodes the details of the algorithm to be computed it says very little about how it may be realised in hardware. In particular:

- the design-style of the final implementation is left unspecified (e.g. synchronous or self-timed);
- the number of functional-units to appear in the generated circuit is not specified (e.g. should separate multipliers be generated for the six '`*`' operations or should fewer, shared multipliers be used);
- the order in which operations within expressions will be evaluated is not specified;
- the execution time of the computation is unspecified (e.g. if we are considering a synchronous design, how many cycles does each multiplication take? How much parallelism should be exploited in the evaluation of the expressions?).

Even for this tiny example one can see that there is a large design-space to consider before arriving at a hardware implementation. To constrain this design-space behavioural HDLs often provide facility for programmers to annotate specifications with low-level design requirements. For example, a designer

may specify constraints which bound the execution time of the algorithm (e.g. < 5 clock cycles) or restrict the resource usage (e.g. one multiplier and three adders). These constraints are used to guide high-level synthesis packages (see Section 1.2.1).

Register-Transfer Level Languages

Register-Transfer Level (RTL) Languages take a much lower-level approach to hardware description. At the top-level an RTL specification models a hardware design as a directed graph in which nodes represent circuit blocks and edges correspond to interconnecting wires and buses. At this level of abstraction a number of design decisions that were left unspecified at the behavioural-level become fixed. In particular, an RTL specification explicitly defines the number of resources used (e.g. 3 multipliers and 1 adder) and the precise mechanism by which data flows between the building blocks of the circuit.

To give a concrete example of this style of programming let us consider specifying our differential equation solver in RTL Verilog. One of the first points to note is that, since many of the design decisions left open at the behavioural level are now made explicit, the RTL specification is a few orders of magnitude larger. For this reason, rather than specifying the whole differential equation solver, we will instead focus on one small part, namely computing the subexpression $3.x.u.dx$.

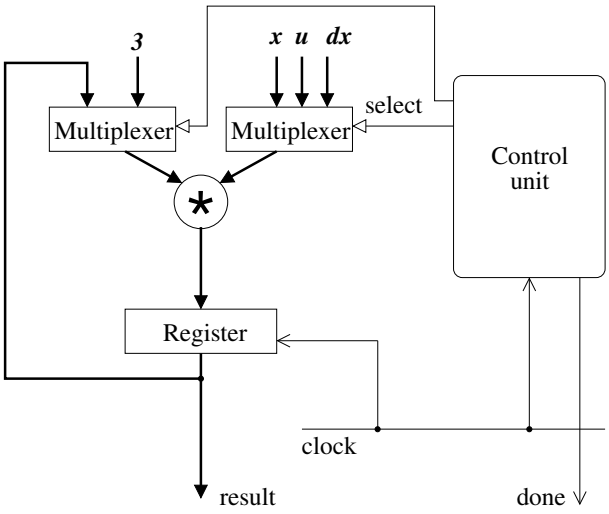


Fig. 1.1. A diagrammatic view of a circuit to compute $3.x.u.dx$

Let us assume that our design is synchronous and that it will contain only one 32-bit single-cycle multiplier. In this case, the circuit we require is shown diagrammatically in Figure 1.1. (We adopt the convention that thick lines represent data wires and thin lines represent control wires.) After being latched, the

```
module 3-input-mux(in1, in2, in3, select, mux_out);
 input in1,in2,in3,select;
 output mux_out;

 // declare data inputs and output as 32-bit values;
 wire [31:0] in1,in2,in3,mux_out;
 // declare select as a 2-bit value;
 wire [1:0] select;

 assign mux_out = (select==0)? in1 :
                  (select==1)? in2 : in3;

endmodule
```

Fig. 1.2. RTL code for a 3-input multiplexer

output of the multiplier is fed back into one of its inputs; in this way, a new term is multiplied into the cumulative product every clock cycle. The control-unit is a finite-state machine which is used to steer data around the circuit by controlling the select inputs of the multiplexers. For the purposes of this example we introduce a control signal, `done`, which is asserted when the result has been computed. We know that the circuit will take 4 cycles to compute its result: 1 cycle for each of the 3 multiplications required and an extra cycle due to the latency added by the register.

The first stage of building an RTL specification is to write definitions for the major components which feature in the design. As an example of a component definition Figure 1.2 gives the RTL-Verilog code for a 3-input multiplexer. The C-style '`?`' operator is used to select one of the inputs to connect to the output depending on the value of the 2-bit `select` input. Whilst the RTL-Verilog language is discussed in more depth in Section 2.1, for now it suffices to note that (*i*) each component is defined as a `module` parameterised over its input and output ports; and (*ii*) the `assign` keyword is used to drive the value of a given expression onto a specified wire/bus.

Let us now turn to the internals of the control-unit. In this example, since we only require 4 sequential control-steps, the state can be represented as a saturating divide-by-4 counter. At each clock-edge the counter is incremented by one; when the counter reaches a value of 3 then it remains there indefinitely. Although the precise details are not important here, RTL-Verilog for the control unit is presented in Figure 1.3. Control signal `arg1_select` is used to control the left-most multiplexer shown in Figure 1.1. In the first control step (when `state` = 0) it selects the value 3, otherwise it simply feeds the output of the register back into the multiplier. Similarly, control signal `arg2_select` is used to control the right-most multiplexer shown in Figure 1.1. In each control step, `arg2_select`, is incremented by one, feeding each of the multiplexer's 3 inputs into the multiplier in turn. Finally `done` is asserted once all three multiplications have been performed and the result latched.

```
module ctrl_unit(clock, arg1_select, arg2_select, done);
  input clock;
  output arg1_select, arg2_select, done;

  // 'arg1_select' and 'done' both single wires:
  wire arg1_select;

  // 'arg2_select' is a 2-bit quantity:
  wire [1:0] arg2_select;

  // declare a 2-bit register as a counter
  reg [1:0] state;
  initial state=0; // initialise register to 0

  // specify the behaviour of state register:
  always @(posedge clock)
      if (state<3) state <= state+1;

  // define control signals in terms of state register:
  assign arg1_select = (state == 0);
  assign arg2_select = state;
  assign done        = (state == 3);

endmodule
```

Fig. 1.3. RTL code for the control-unit

If we now assume `module`-definitions for the rest of the circuit's components (`multiplier`, `2-input-mux` and `32-bit-register`) we can complete our RTL-design by specifying the interconnections between these components as shown in Figure 1.4. We now have a module, `compute_product`, with a `result` output, a control-wire which signals completion of the computation (`done`), a clock input and input ports to read values of `x`, `u` and `dx`.

One can see from this example just how wide the gap between the behavioural-level and the RTL-level actually is. As well as the sheer amount of code one has to write, there are a number of other disadvantages associated with RTL-level specification. In particular, since so many design decisions are ingrained so deeply in the specification, it is difficult to make any architectural modifications at the RTL-level. For example, if we wanted to change the code shown in Figure 1.4 to use 2 separate multipliers (instead of 1), we would essentially have to re-write the code from scratch—not only would the data-path have to be redesigned, but the control-unit would have be re-written too.

RTL languages give a hardware designer a great deal of control over the generated circuit. Whilst, in some circumstances, this is a definite advantage it must be traded off against the huge complexity involved in making architectural changes to the design.

```
module compute_product(clock, x, u, dx, result, done);
  input clock, x, u, dx;
  output result, done;

  wire clock, done;
  wire [31:0] x, u, dx, result;

  // define the internal wires needed for connections:

  wire [31:0] value1, value3, mult_arg1, mult_arg2;
  wire [31:0] mult_out, reg_out;
  wire mux1_select, mux2_select;

  assign value1 = 1;
  assign value3 = 3;

  // instantiate the components, specifying their
  // interconnections:

  2-input-mux 2mux_inst(value3, reg_out, mux1_select,
                        mult_arg1);
  3-input-mux 3mux_inst(x, u, dx, mux2_select,
                        mult_arg2);
  multiplier mult_inst(mult_arg1, mult_arg2,
                        mult_out);
  32-bit-register reg_inst(clock, mult_out, reg_out);

  ctrl_unit ctrl_inst(clock, mux1_select, mux2_select, done);
endmodule
```

Fig. 1.4. RTL code to connect the components of the multiplication example together

Netlist Specification

At the netlist level, a specification is described in terms of an interconnection of gates. Thus, whereas in the RTL control-unit specification of Figure 1.3 we were able to use operators such as '`==`' and '`+`', at the netlist level these have to be specified explicitly in terms of their gate-level description.

For example, to specify a 3-bit equality tester (as used in the definition of control signals in Figure 1.3) in terms of primitive gates (`and`, `xor`, `not`) we can use the code shown in Figure 1.5. (The corresponding circuit diagram is shown in Figure 1.6.) Given this definition of `3bitEQ` we can replace the '`==`' operators of Figure 1.3 with:

```
3bitEQ eq1(state, 3'b000, arg1_select};
3bitEQ eq2(state, 3'b100, done);
```

where the notation n`'b`$\langle x \rangle$ represents the n-bit binary number $\langle x \rangle$. Of course, to complete the netlist specification we would also have to replace the mod-

```
module 3bitEQ (in1, in2, out);
 input in1, in2;
 output out;

 wire [2:0] in1, in2;
 wire out;

 xor(p1, in1[2], in2[2]);
 xor(p2, in1[1], in2[1]);
 xor(p3, in1[0], in2[0]);

 or(a1, p1, p2);
 or(a2, a1, p3);

 not(out, a2);
end module
```

Fig. 1.5. A netlist-level Verilog specification of a 3-bit equality tester

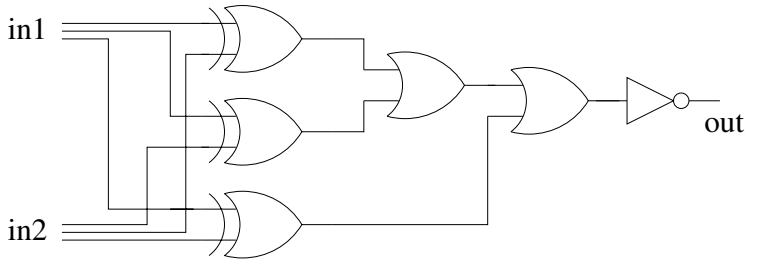


Fig. 1.6. Circuit diagram of a 3-bit equality tester

ules corresponding to the multiplexers, adder and register with their gate-level equivalents. For space reasons, the details are omitted.

Some HDLs support even lower-level representations than this. For example, the Verilog language facilitates the description of circuits in terms of the interconnections between individual transistors. Other HDLs also allow place-and-route information to be incorporated into the netlist specification.

1.2 Hardware Synthesis

Hardware synthesis is a general term used to refer to the processes involved in automatically generating a hardware design from its specification. Mirroring the classification of HDLs (outlined in Section 1.1), hardware synthesis tools are typically categorised according to the level of abstraction at which they operate (see Figure 1.7):

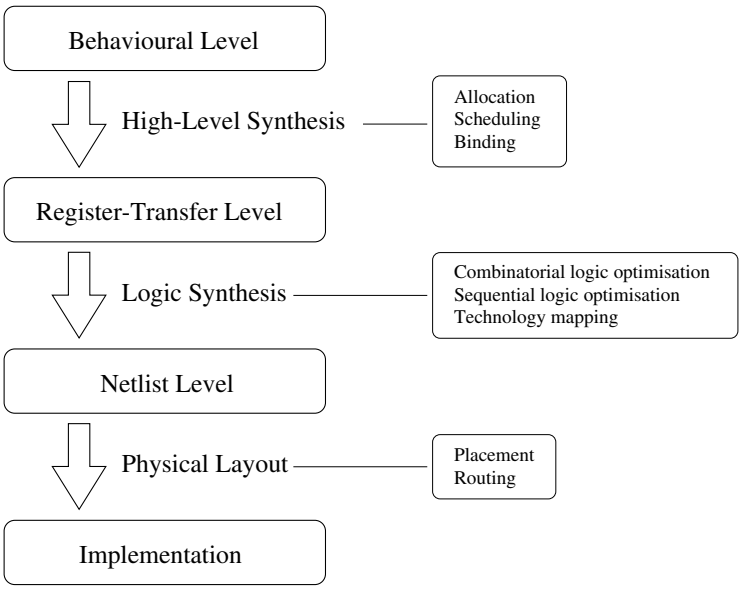


Fig. 1.7. A categorisation of HLS systems and the synthesis tasks performed at each level of the translation process

High-Level Synthesis is the process of compiling a behavioural language into a structural description at the register-transfer level. (We discuss high-level synthesis at length in Section 1.2.1.)

Logic Synthesis refers to the translation of an RTL specification into an optimised netlist. Tasks performed at this level include combinatorial logic optimisation (e.g. boolean minimisation), sequential logic optimisation (e.g. state-minimisation) and technology mapping (the mapping of generic logic onto the specific primitives provided by a particular technology).

Physical Layout involves choosing where hardware blocks will be positioned on the chip (*placement*) and generating the necessary interconnections between them (*routing*). This is an difficult optimisation problem; common techniques for its solution include simulated annealing and other heuristic function-minimisation algorithms.

Since this monograph is only concerned with High-Level Synthesis we do not discuss logic synthesis or place-and-route further. The interested reader is referred to surveys of these topics [41, 13, 127].

1.2.1 High-Level Synthesis

The roots of High-Level Synthesis (HLS) can be traced back further than one may expect. One pioneering system, ALERT [50], was developed at the IBM T. J. Watson Research Centre in the late 1960s. The package was used to automatically translate behavioural specifications written in APL [123] into logic-level

implementations. A complete IBM 1800 computer was synthesised automatically (albeit one that required twice as many components as the manually designed version).

Although in the 1970s most hardware synthesis research focused on lower-level issues, such as logic synthesis and place-and-route, some forward-looking researchers concentrated on HLS. For example, the MIMOLA system [146, 95], which originated at the University of Kiel in 1976, generates a CPU and microcode from a high-level input specification.

In the 1980s the field of high-level synthesis grew exponentially and started to spread from academia into industry. A large number of HLS systems were developed encompassing a diverse range of design-styles and applications (e.g. digital signal processing [92] and pipelined processors [117]). Today there is a sizable body of literature on the subject. In the remainder of this section we present a general survey of the field of high-level synthesis.

Overview of a Typical High-Level Synthesis System

The process of high-level synthesis is commonly divided into three separate subtasks [41]:

- *Allocation* involves choosing which resources will appear in the final circuit (e.g. three adders, two multipliers and an ALU).
- *Binding* is the process of assigning operations in the high-level specification to low-level resources—e.g. the `+` in line 4 of the source program will be computed by `adder_1` whereas the `+` in line 10 will be computed by the `ALU`.
- *Scheduling* involves assigning start times to operations in a given expression (e.g. for an expression, $x * y + z * w$, we may decide to compute $x * y$ and $z * w$ in parallel at time $t = 0$ and perform the addition at time $t = 1$.)

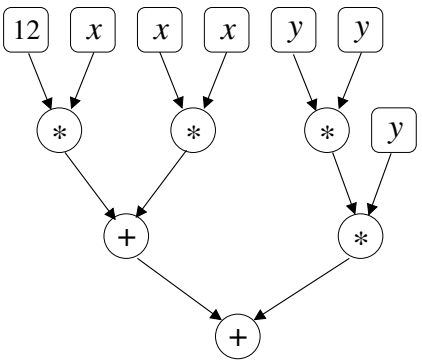


Fig. 1.8. Dataflow graph for expression: $12x + x^2 + y^3$

Let us illustrate each of these phases with a simple example. Consider a behavioural specification which contains the expression, $e = 12x + x^2 + y^3$,

where x and y are previously defined variables. Figure 1.8 shows the data-flow graph corresponding to this expression. In this example we assume that the user has supplied us with an allocation of two multipliers (we refer to the multiplier resources as $m1$, $m2$) and two adders ($a1$, $a2$).

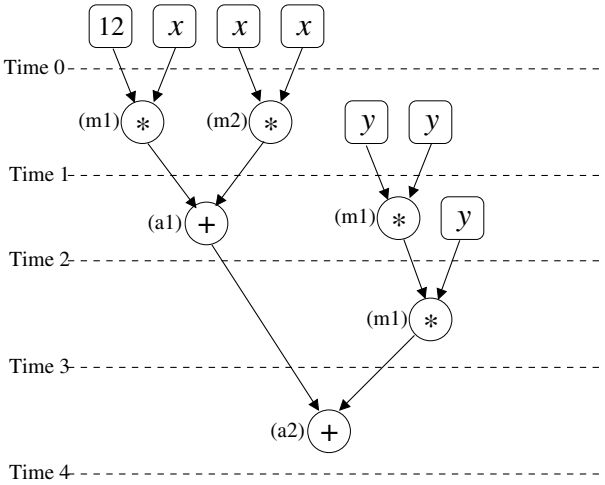


Fig. 1.9. The results of scheduling and binding

Figure 1.9 shows one possible schedule for e under these allocation constraints. In the first time step, we perform the multiplications $12 * x$ and $x * x$; in the second time step these products are added together and we compute $y * y$; the third time step multiplies a previous result by y to obtain y^3; finally, the fourth time step contains a single addition to complete the computation of e. Note that although the data-dependencies in e would permit us to compute $y * y$ in the first time step our allocation constraints forbid this since we only have two multipliers available. Each time step in the schedule corresponds to a single clock-cycle at the hardware level[1] (assuming that our multipliers and adders compute their results in a single cycle). Thus the computation of expression e under the schedule of Figure 1.9 requires four clock cycles.

After scheduling we perform binding. The result of the binding phase is also shown in Figure 1.9 where operations are annotated with the name of the hardware-level resource with which they are associated. (Recall that we refer to our allocated resources as $m1$, $m2$, $a1$ and $a2$). We are forced to bind the two multiplications in the first time-step to separate multipliers since the operations occur concurrently (and hence cannot share hardware). In binding the other operations more choices are available. Such choices can be guided in a number of ways—for example one may choose to minimise the number of resources used or attempt to bind operations in such a way as to minimise routing and multiplexing costs.

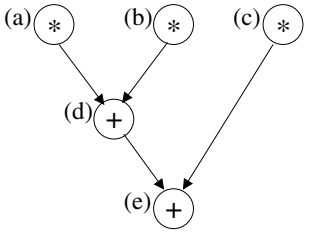
[1] Conventional HLS systems typically generate synchronous implementations.

The following sections discuss each of the phases of HLS in more detail and outline a few of the techniques and algorithms which have been most prevalent in each area.

Scheduling

Scheduling algorithms can be usefully divided into two categories as to whether they are *constructive* or *transformational* in their approach. Transformational algorithms start with some schedule (typically maximally parallel or maximally serial) and repeatedly apply transformations in an attempt to bring the schedule closer to the design requirements. The transformations allow operations to be parallelised or serialised whilst ensuring that dependency constraints between operations are not violated. A number of different search strategies governing the application of transformations have been implemented and analysed. For example, whereas Expl [18] performs an exhaustive search of the design space, the Yorktown Silicon Compiler [28] uses heuristics to guide the order in which transformations are performed. The use of heuristics dramatically reduces the search space, allowing larger examples to be scheduled at the cost of possibly settling for a sub-optimal solution.

In contrast, constructive algorithms build up a schedule from scratch by incrementally adding operations. The simplest example of the constructive approach is *As Soon As Possible* (ASAP) scheduling [98]. This algorithm involves topologically sorting the operations in the dependency graph and inserting them (in their topological order) into time steps under the constraints that (*i*) all predecessors in the dependency graph have already been scheduled in earlier timesteps and (*ii*) limits on resource usage (if any) are not exceeded. The MIMOLA [146] system employs this algorithm.



ASAP-Schedule:

Step	∗	+
1	(c)	–
2	(b)	–
3	(a)	–
4	–	(d)
5	–	(e)

List-Schedule:

Step	∗	+
1	(a)	–
2	(b)	–
3	(c)	(d)
4	–	(e)

Fig. 1.10. (*left*) the dependencies between operations for an expression of the form $xy + z$. Operations are labelled with letters (a)–(e); (*centre*) an ASAP Schedule of the expression for a single adder and a single multiplier. (*right*) a List Schedule under the same resource constraints

A problem with the ASAP method is that it ignores the global structure of an expression: whenever there is a choice of which operation to schedule one is chosen arbitrarily; the implication that this choice has on the *latency* (number of time steps required) of the schedule is ignored. Figure 1.10 highlights the

inadequacies of the ASAP algorithm. In this case we see that a non-critical multiplication, (c), has been scheduled in the first step, blocking the evaluation of the more critical multiplications, (a) and (b) until later time steps.

List Scheduling alleviates this problem. Whenever there is a choice between multiple operations a global evaluation function is used to choose intelligently. A typical evaluation function, $f(n)$, maps a node, n, onto the length of the longest path in the dependency graph originating from n. When a choice occurs, nodes with the largest values of $f(n)$ are scheduled first. Figure 1.10 shows an example of a list schedule using this heuristic function. Notice how the schedule is more efficient than the one generated by the ASAP algorithm since nodes on the critical path are prioritised. A number of HLS systems use List Scheduling (e.g. BUD [97] and Elf [55]). (As an aside, note that List Scheduling is also a common technique in the field of software compilers where it is used to reduce the number of pipeline stalls in code generated for pipelined machines with hardware interlocking [103]).

Allocation and Binding

In many synthesis packages, the tasks of allocation and binding are performed in a single phase. (Recall that allocation involves specifying how many of each resource-type will be used in an implementation and binding involves assigning operations in a high-level specification to allocated resources). This phase is further complicated if one considers *complex* resources: those capable of performing *multiple* types of operation [41]. An example of a complex resource is an Arithmetic Logic Unit (ALU) since, unlike a *simple* functional-unit (e.g. an adder), it is capable of performing a whole set of operations (e.g. addition, multiplication, comparison). The aim of allocation/binding is typically to minimise factors such as the number of resources used and the amount of wiring and steering logic (e.g. multiplexers) required to connect resources.

Let us start by considering the simplest case of minimising only the number of resources used (i.e. ignoring wiring and steering logic). In this case the standard technique involves building a *compatibility graph* from the input expression [98]. The compatibility graph has nodes for each operation in the expression and an undirected edge (n_1, n_2) iff n_1 and n_2 can be computed on the same resource (i.e. if they do not occur in the same time-step[2] and there is a single resource type capable of performing the operations corresponding to both n_1 and n_2). Each *clique*[3] in the compatibility graph corresponds to operations which can share a single resource. The aim of a synthesis tool is therefore to find the minimum number of cliques which covers the graph (or, phrased in a different way, to find the *maximal*[4] cliques of the graph). Unfortunately the *maximal clique problem* [6] is NP-complete so, to cope with large designs, heuristic methods are

[2] We assume scheduling has already been performed.

[3] Consider a graph, G, represented as sets of verticies and edges, (V, E). A *clique* of G is a set of nodes, $\mathcal{S} \subseteq V$, such that $\forall(n1, n2) \in \mathcal{S}.\ (n1, n2) \in E \vee (n2, n1) \in E$.

[4] A clique is *maximal* if it is not contained in any other clique.

often used to find approximate solutions. (Note the duality between this method and the "conflict-graph / vertex-colouring" technique used for register allocation in optimising software compilers [33].)

More complicated approaches to allocation/binding attempt to minimise both the number of resources *and* the amount of interconnect and multiplexer-logic required. This is often referred to as the *minimum-area* binding problem. Minimising wiring overhead is becoming increasingly important as the feature-size of transistors decreases; in modern circuits wiring is sometimes the dominant cost of a design. The compatibility graph (described above) can be extended to the minimum-area binding problem by adding weights to cliques [133]. The weights correspond to the cost of assigning the verticies in the clique to a single resource (i.e. the cost of the resource itself plus the cost of the necessary interconnect and steering logic). The aim is now to find a covering set of cliques with minimal total weight. This is, of course, still an NP-complete problem so heuristic methods are used in practice.

In contrast to graph-theoretic formulations, some high-level synthesis systems view allocation/binding as a search problem. Both MIMOLA [95] and Splicer [116] perform a directed search of the design space to choose a suitable allocation and binding. Heuristics are used to reduce the size of the search space.

The Phase Order Problem

Note that the scheduling, allocation and binding phases are deeply interrelated: decisions made in one phase impose constraints on subsequent phases. For example if a scheduler decides to allocate two operations to the same time-step then a subsequent binding phase is forbidden from assigning the operations to the same hardware resource. If a bad choice is unknowingly made in one of the early phases then poor quality designs may be generated. This is known as the *phase-order* problem (sometimes referred to as the *phase-coupling* problem). In our simple example (Figures 1.8 and 1.9), we perform scheduling first and then binding. This is the approach taken by the majority of hardware synthesis systems (including Facet [136], the System Architect's Workbench (SAW) [135] and Cathedral-II [92]). However, some systems (such as BUD [97], and Hebe [88]) choose to perform binding and allocation before scheduling. Each approach has its own advantages and shortcomings.

A number of systems have tried to solve the phase-order problem by combining scheduling, allocation and binding into a single phase. For example, the Yorktown Silicon Compiler [28] starts with a maximally parallel schedule where operations are all bound to separate resources. A series of transformations—each of which affects the schedule, binding and allocation—are applied in a single phase. Another approach is to formulate simultaneous scheduling and binding as an Integer Linear Programming (ILP) problem; a good overview of this technique is given by De Micheli [41]. Recent progress in solving ILP constraints and the development of reliable constraint-solving packages [78] has led to an increased interest in this technique.

1.3 Motivation for Higher Level Tools

Hardware design methodologies and techniques are changing rapidly to keep pace with advances in fabrication technology. The advent of System-on-a-Chip (SoC) design enables circuits which previously consisted of multiple components on a printed circuit board to be integrated onto a single piece of silicon. New design styles are required to cope with such high levels of integration. For example, the Semiconductor Industry Association (SIA) Roadmap [1] acknowledges that distributing a very high frequency clock across large chips is impractical; it predicts that in the near future chips will contain a large number of separate local clock domains connected via an asynchronous global communications network. It is clear that HLS systems must evolve to meet the needs of modern hardware designers:

- Facility must be provided to explore the different design styles arising from a single high-level specification. For example, a designer may wish to partition some parts of a design into multiple clock domains and map other parts to fully asynchronous hardware.
- HLS systems must be capable of exploring architectural tradeoffs at the system level (e.g. duplication/sharing of large scale resources such as processors, memories and busses).
- Hardware description languages must support the necessary abstractions to structure large designs (and also to support the *restructuring* of large designs without wholesale rewriting).

It is our belief that existing HLS tools and techniques are a long way from achieving these goals. In particular it seems that conventional HLS techniques are not well suited to exploring the kind of system-level architectural trade-offs described above. In this section we justify this statement by discussing some of the limitations of conventional hardware description languages and high-level synthesis tools.

1.3.1 Lack of Structuring Support

Although behavioural languages provide higher-level primitives for algorithmic description, their support for structuring large designs is often lacking. Many behavioural HDLs use structural *blocks* parameterised over input and output ports as a structuring mechanism. This is no higher-level than the structuring primitives provided at the netlist level. For example, at the top level, a Behavioural Verilog [74] program still consists of `module` declarations and instantiations albeit that the modules themselves contain higher-level constructs such as assignment, sequencing and while-loops.

Experience has shown that the notion of a block is a useful syntactic abstraction, encouraging structure by supporting a "define-once, use-many" methodology. However, as a *semantic abstraction* it buys one very little; in particular: (*i*) any part of a block's internals can be exported to its external interface; and

(*ii*) inter-block control- and data-flow mechanisms must be coded explicitly on an *ad-hoc* basis.

Point (*i*) has the undesirable effect of making it difficult to reason about the global (inter-module) effects of local (intra-module) transformations. For example, applying small changes to the local structure of a block (e.g. delaying a value's computation by one cycle) may have dramatic effects on the global behaviour of the program as a whole. We believe point (*ii*) to be particularly serious. Firstly, it leads to low-level implementation details scattered throughout a program—e.g. the definition of explicit control signals used to sequence operations in separate modules, or (arguably even worse) reliance on unwritten inter-module timing assumptions. Secondly, it inhibits compiler analysis: since inter-block synchronisation mechanisms are coded on an ad hoc basis it is very difficult for the compiler to infer a system-wide ordering on events (a prerequisite for many global analyses—see Chapter 6). Based on these observations, we contend that structural blocks are not a high-level abstraction mechanism.

1.3.2 Limitations of Static Scheduling

We have seen that conventional high-level synthesis systems perform scheduling at compile time. In this framework mutually exclusive access to shared resources is enforced by statically serialising operations. While this approach works well for simple resources (e.g. arithmetic functions) whose execution time is statically bounded, it does not scale elegantly to system-level resources (e.g. IO-controllers, processors and busses). In real hardware designs it is commonplace to control access to shared system-level resources *dynamically* through the use of arbitration circuitry [67]. However, existing synthesis systems require such arbitration to be coded explicitly on an *ad-hoc* basis at the structural level. This leads to a design-flow where individual modules are designed separately in a behavioural synthesis system and then composed manually at the RT-level. It is our belief that a truly high-level synthesis system should not require this kind of low-level manual intervention.

Another problem with conventional scheduling methods is that they are only applicable to synchronous circuits—the act of scheduling operations into system-wide control steps assumes the existence of a single global clock. Thus, conventional scheduling techniques cannot be performed across multiple clock domains and are not applicable to asynchronous systems. Such limitations make it impossible to explore alternative design styles (e.g. multiple clock-domains or asynchronous implementations) in the framework of conventional HLS.

The Black-Box Approach

Although some researchers have investigated the possibility of performing high-level synthesis interactively [52, 145] the majority of HLS tools take a black-box approach: behavioural specifications are translated into RTL descriptions without any human guidance. The problem with black-box synthesis is that when

unsuitable designs are generated there is very little a designer can do to improve the situation. Often one is often reduced to blindly changing the behavioural specification/constraints whilst trying to second guess the effects this will have on the synthesis tool.

A number of researchers have suggested that source-level transformation of behavioural specifications may be one way to open the black-box, allowing more user-guidance in the process of architectural exploration [53]. However, although a great deal of work has been carried out in this area [93, 142, 107] behavioural-level transformations are currently not used in industrial high-level synthesis. Other than the lack of tools to assist in the process, we believe that there are a number of reasons why behavioural-level transformation has not proved popular in practice:

- Many features commonly found in behavioural HDLs make it difficult to apply program-transformation techniques (e.g. an imperative programming style with low-level circuit structuring primitives such as Verilog's `module` construct).
- It is difficult for a designer to know what impact a behavioural-level transformation will have on a generated design.

We see these issues as limitations in conventional high-level hardware description languages.

1.4 Structure of the Monograph

In this monograph we focus on HLS, addressing the limitations of conventional hardware description languages and synthesis tools outlined above. Our research can be divided into two inter-related strands: (*i*) the design of new high-level languages for hardware description; and (*ii*) the development of new techniques for compiling such languages to hardware.

We start by surveying related work in Chapter 2 where, in contrast to this chapter which gives a general overview of hardware description languages and synthesis, a number of *specific* HDLs and synthesis tools are described in detail. (Note that this is not the only place where related work is considered: each subsequent chapter also contains a brief 'related work' section which summarises literature of direct relevance to that chapter.)

The technical contributions of the monograph start in Chapter 3 where the design of SAFL, a small functional language, is presented. SAFL (which stands for Statically Allocated Functional Language) is a behavioural HDL which, although syntactically and semantically simple, is expressive enough to form the core of a high-level hardware synthesis system. SAFL is designed specifically to support:

1. high-level program transformation (for the purposes of architectural exploration);
2. automatic compiler analysis and optimisation—we focus especially on *global analysis and optimisation* since we feel that this is an area where existing HLS systems are currently weak; and
3. structuring techniques for large SoC designs.

Having defined the SAFL language we use it to investigate a new scheduling technique which we refer to as *Soft Scheduling* (Chapter 4). In contrast to existing static scheduling techniques (see Section 1.3.2), Soft Scheduling generates the necessary circuitry to perform scheduling dynamically. Whole-program analysis of SAFL is used to statically remove as much of this scheduling logic as possible. We show that Soft Scheduling is more expressive than static scheduling and that, in some cases, it actually leads to the generation of faster circuits. It transpires that Soft Scheduling is a strict generalisation of static scheduling. We demonstrate this fact by showing how *local* source-to-source program transformation of SAFL specifications can be used to represent any static scheduling policy (e.g. ASAP or List Scheduling—see Section 1.2.1).

In order to justify our claim that "the SAFL language is suitable for hardware description and synthesis" a high-level synthesis tool for SAFL has been designed and implemented. In Chapter 5 we describe the technical details of the *FLaSH Compiler*: our behavioural synthesis tool for SAFL. The high-level properties of the SAFL language allow us to compile specifications to a variety of different design styles. We illustrate this point by describing how SAFL is compiled to both purely synchronous hardware and also to GALS (Globally Asynchronous Locally Synchronous) [34, 77] circuits. In the latter case the resulting design is partitioned into a number of different clock domains all running asynchronously with respect to each other. We describe the intermediate code format used by the compiler, the two primary design goals of which are (*i*) to map well onto hardware; and (*ii*) to facilitate analysis and transformation.

In Chapter 6 we demonstrate the utility of the FLaSH compiler's intermediate format by presenting a number of global analyses and optimisations. We define the concept of *architecture-neutral* analysis and optimisation and give an example of this type of analysis. (Architecture-neutral analyses/optimisations are applicable regardless of the design style being targeted.) We also consider *architecture-specific* analyses which are able to statically infer some timing information for the special case of synchronous implementation. A number of associated optimisations and experimental results are presented.

Whilst SAFL is an excellent vehicle for high-level synthesis research we recognise that it is not expressive enough for industrial hardware description. In particular the facility for I/O is lacking and, in some circumstances, the "call and wait for result" interface provided by the function model is too restrictive. To address these issues we have developed a language, SAFL+, which extends SAFL with process-calculus features including synchronous channels and channel-passing in the style of the π-calculus [100]. The incorporation of channel-passing allows a style of programming which strikes a balance between

the flexibility of structural blocks and the analysability of functions. In Chapter 7 we describe both the SAFL+ language and the implementation of our SAFL+ compiler. We demonstrate that our analysis and compilation techniques for SAFL (Chapters 4 and 6) naturally extend to SAFL+.

A contributing factor to the success of Verilog and VHDL is their support for *both* behavioural *and* structural-level design. The ability to combine behavioural and structural primitives in a single specification offers engineers a powerful framework: when the precise low-level details of a component are not critical, behavioural constructs can be used; for components where finer-grained control is required, structural constructs can be used. In Chapter 8 we present a single framework which integrates a structural HDL with SAFL. Our structural HDL, which is embedded in the functional subset of ML [101], is used to describe acyclic combinatorial circuits. These circuit fragments are instantiated and composed at the SAFL-level. Type checking is performed across the behavioural-structural boundary to catch a class of common errors statically. As a brief aside we show how similar techniques can be used to embed a functional HDL into Verilog.

Chapter 9 justifies our claim that "the SAFL language is well-suited to source-level program transformation". As well as presenting a large global transformation which allows a designer to explore a variety of hardware/software partitionings. We also describe a transformation from SAFL to SAFL+ which converts functions into pipelined stream processors.

Finally, a realistic case-study is presented in Chapter 10 where the full 'SAFL/SAFL+ to silicon' design flow is illustrated with reference to a DES encryption/decryption circuit. Having shown that the performance of our DES circuit compares favourably with a hand-coded RTL version we give an example of interfacing SAFL to external components by integrating the DES design with a custom hardware VGA driver written in Verilog.

We include brief summaries and conclusions at the end of each chapter. Global conclusions and directions for further work are presented in Chapter 11.

2

Related Work

The previous chapter gave a general overview of languages and techniques for hardware description and synthesis. The purpose of this chapter is to provide a more detailed overview of work which is directly relevant to this monograph. A number of languages and synthesis tools are discussed in turn; a single section is devoted to each topic.

2.1 Verilog and VHDL

The Verilog HDL [74] was developed at Gateway Design Automation and released in 1983. Just two years later, the Defence Advanced Research Agency (DARPA) released a similar language called VHDL [73]. In contrast to Verilog, whose primary design goal is to support efficient simulation, the main objective of VHDL is to cope with large hardware designs in a more structured way.

Today Verilog and VHDL are by far the most commonly used HDLs in industry. Although the languages have different syntax and semantics, they share a common approach to modelling digital circuits, supporting a wide range of description styles ranging from behavioural specification through to gate-level design. For the purposes of this survey, a single section suffices to describe both languages.

Initially Verilog and VHDL were designed to facilitate only the *simulation* of digital circuits; it was not until the late 1980s that automatic synthesis tools started to appear. The expressiveness of the languages makes it impossible for synthesis tools to realise all VHDL/Verilog programs in hardware and, as a consequence, there are many valid programs which can be simulated but not synthesised. Those programs which can be synthesised are said to be *synthesisable*. Although there has been much recent effort to precisely define and standardise the synthesisable subsets of VHDL/Verilog, the reality is that synthesis tools from different commercial vendors still support different constructs. (At the time of writing VHDL is ahead in this standardisation effort: a recently published IEEE standard defines the syntax and semantics of synthesisable RTL-VHDL [75]. Future RTL-VHDL synthesis tools are expected to adhere to this standard.)

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 19–34, 2004.

The primary abstraction mechanism used to structure designs in both Verilog and VHDL is the *structural block*. Structural blocks are parameterised over input and output ports and can be instantiated hierarchically to form circuits. The Verilog language, whose more concise syntax is modelled on C [83], uses the `module` construct to declare blocks. For example, consider the following commented description of a half-adder circuit:

```
module HALF_ADDER (a, b, carry, sum);
     input a, b;         // declare inputs
     output carry, sum; // declare outputs

     and GATE1 (carry, a, b);
       // instantiate an and-gate (instance-name: GATE1)
       // the output of the gate is connected to carry
       // the inputs of the gate are connected to a and b

     xor GATE2 (sum, a, b)
       // instantiate an xor-gate (instance-name: GATE2)
endmodule
```

The more verbose syntax of VHDL is modelled on ADA [9]. In contrast to Verilog, the VHDL language is strongly typed, supports user-defined datatypes and forces the programmer to specify interfaces explicitly. In VHDL, each structural block consists of two components: an *interface description* and an *architectural body*. The half-adder circuit (see above), has the following VHDL interface description:

```
entity HALF_ADDER is
     port(a, b: in bit; sum,carry: out bit);
end HALF_ADDER
```

and the following architectural body:

```
architecture STRUCTURE of HALF_ADDER is
     component AND2   port (x, y: in bit; o: out bit);
     component EXOR2  port (x, y: in bit; o: out bit);
begin
     GATE1: AND2  port map (a,b,carry);
     GATE2: EXOR2 port map (a,b,sum);
end STRUCTURE;
```

Both VHDL and Verilog use the *discrete-event* model to represent hardware. In this paradigm a digital circuit is modelled as a set of concurrent *processes* connected with *signals*. Signals represent wires whose values change over time. For each *time-frame* (unit of simulation time) a signal has a specific value; typical values include 0, 1, X (undefined) and Z (high-impedance). Processes may be *active* (executing) or *suspended*; suspended processes may be reactivated by *events* generated when signals' values change.

To see how the discrete-event model can be used to describe hardware consider modelling a simple D-type flip-flop (without reset). The flip-flop has two inputs: data-input (`d`), clock (`clk`); and a single data-output (`q`). On the rising edge of each clock pulse the flip-flop copies its input to its output after a propagation delay of 2 time-frames. In Verilog the flip-flop can be modelled as follows:

```
module DFF(d,clk,q);
   input d,clk;
   output q;
   reg q;
   always @(posedge clk) q <= #2 d;
endmodule
```

This Verilog specification contains a single process declaration (introduced with the `always` keyword). The body of the process is executed every time the event "`posedge clk`" occurs (i.e. every time the `clk` signal changes from 0 to 1). The body of the process contains a *non-blocking* assignment, `q <= #2 d`, which assigns the value of input `d` into register `q` after a delay of 2 time-frames. A non-blocking assignment, x `<= #`n e, causes no delay in a process' execution, but schedules the *current* value of e to be assigned to register x after n time-frames[1]. To see how this differs from conventional imperative assignment consider the following:

```
x <= #1 y;
y <= #1 x;
```

This sequence of non-blocking assignments swaps the values of registers `x` and `y`. The key point to note is that there is no control-dependency between the two assignments: both the right hand sides are evaluated *before* either `x` or `y` are updated. For the sake of comparison the equivalent VHDL code for the flip-flop is given in Figure 2.1.

Although the discrete-event timing model is apposite and powerful for hardware description it is often criticised for being difficult to reason about. To see some of the issues which can cause confusion consider the Verilog program shown in Figure 2.2. Although the program is short, its behaviour is not immediately obvious. To understand the workings of this code one must observe that the body of the main process *always* causes a transition on signal, `q`, which in turn re-activates the process. Hence an infinite loop is set up in which signal `q` is constantly updated. Also recall that although the effect of the non-blocking assignments is delayed, it is the *current* value of the right-hand-side expression that is written to `q`. Thus the statement, `q <= #5 q`, which at first sight appears to assign `q` to itself, does in fact change the value of `q`. Figure 2.3 shows this Verilog code fragment running in a simulator. Signal `q` repeatedly remains low for 3 time frames and then goes high for 2 subsequent time frames.

[1] Writing x `<=` e is simply shorthand for x `<= #0` e.

```
entity DFF is
port (d, clk: IN  bit; q: OUT bit);
end DFF;

architecture BEHAVIOR of DFF is
begin
   process (clk)
   begin
       if clk = '1' then q <= d after 2ns;
       end if;
   end process;
end BEHAVIOR;
```

Fig. 2.1. VHDL code for a D-type flip-flop

```
module confusing_example;
   reg q;
   initial q <= 0;   // set initial value of q to 0

   // main process: executed whenever signal q changes
   always @(q)
      begin  q <= #3 !q;
             q <= #5 q;
      end
endmodule
```

Fig. 2.2. Verilog code for the `confusing_example`

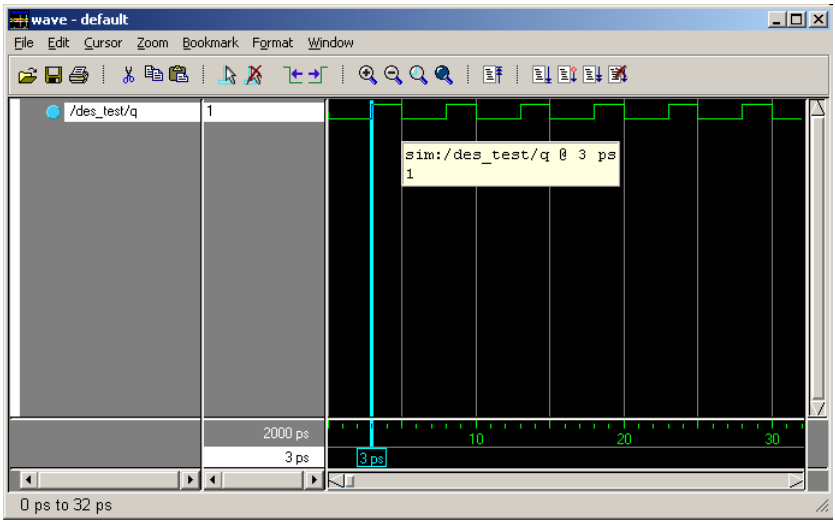


Fig. 2.3. Running the `confusing_example` module in a simulator

The difficulty of reasoning about Verilog/VHDL has inspired a number of researchers to develop formal semantics for the languages. Chapman and Hwang [124] give a translation of a subset of behavioural VHDL into Hoare's CSP process algebra [31]. Their translation essentially models the structure of a discrete-event simulator—user defined processes are composed in parallel along with a *kernel process* which co-ordinates their activity. Other researchers have developed semantics using a wide range of different formalisms including higher-order logic [140], Petri nets [111] and temporal-logic [144]. Although this research represents a significant step in specifying the behaviour of VHDL/Verilog, to date only small, tractable subsets of the languages have been formalised. Whether or not any of these techniques will scale to formalising the full languages remains to be seen.

For more information on the Verilog/VHDL languages the reader is referred to HDL textbooks [91, 115].

2.2 The Olympus Synthesis System

Developed as a research project at Stanford University, the *Olympus Synthesis System* [43] is a vertically integrated set of tools for the synthesis of digital circuits. Following the Olympus theme, each of the tools is named after a character in ancient Greek mythology: high-level synthesis is performed by *Hercules* and *Hebe*; simulation is performed by *Ariadne*; logic synthesis is carried out by *Mercury* and technology mapping is the job of *Ceres*[2]. Since this monograph is only concerned with high-level synthesis we choose to focus only on *Hercules* and *Hebe*.

2.2.1 The HardwareC Language

The input to Olympus is a behavioural specification written in a language called *HardwareC* [86]. Although the language has a C-like syntax this is where the similarity with C [83] ends. HardwareC supports the four fundamental structuring primitives outlined in Figure 2.4. Integers and booleans are provided as primitive types; the only structured type supported is the array. The usual imperative constructs (sequencing, assignment, iteration and conditionals) are also provided.

Synchronous (blocking) channels are used for communication between parallel processes. Channel operations `send` and `receive` have their usual meanings; the `msgwait` operation takes a channel, c, as an argument and returns a boolean flag indicating whether c has pending messages. Channels are uni-directional: as

[2] Purists may complain that the Olympus metaphor is shattered by the observation that whereas some the tools take on the identity of characters from Greek mythology, others are named after their *Roman* counterparts. However, since the Romans borrowed much of their mythology from the Greeks this is really only a point of pedantry.

Abstraction	Description
`block`	A structural block parameterised over input/output ports. The body of a `block` contains instantiations of other `blocks` and `processes` and defines their interconnections.
`process`	A structural block parameterised over input/output ports. A `process` contains imperative statements which are executed repeatedly—on completion of the last statement, the process restarts.
`function`	A `function` has formal-parameters. The `function` body contains imperative statements which are executed when the function is called. A result is returned to the caller.
`procedure`	A `procedure` is a `function` which does not return a result.

Fig. 2.4. HardwareC's structuring primitives

in the Occam language [76], each channel must be connected to a single reading process and a single writing process.

An interesting feature of the HardwareC language is its facility for describing shared resources. The `instance` statement allows named instances of functions/procedures to be declared. At the hardware level an *instance* of a procedure is represented as a shared resource; multiple calls to the same instance corresponds to resource sharing. A call to a procedure or a function may be *bound* or *unbound*. An unbound call to a function (or procedure), $f(x)$, does not specify which instance of f is used to perform the computation. In contrast, a bound call specifies the name of a specific instance which will be used to perform the computation. For example, the following HardwareC code contains two unbound calls to a function called `adder`:

```
a = adder(3,4);
b = adder(5,6);
```

Since the calls are unbound the compiler is free to choose whether the two calls to `adder` are synthesised into two separate adders, or whether a shared resource is used. Demonstrating the use of bound calls, the following code shows how the programmer can specify explicitly that a single, shared adder should be used:

```
/* declare an instance called my_adder */
instance adder my_adder;

a = my_adder(3,4);
b = my_adder(5,6);
```

Note that the programmer must be careful when using unbound calls to functions/procedures with internal state. In this case the language semantics are undefined.

HardwareC provides constructs to express *timing-constraints* and *resource-constraints* which are used to guide the synthesis process. Timing-constraints specify upper and lower bounds on times (in terms of implementation-level clock cycles) between labelled program points. All timing-constraints must be *intra-procedural*; that is for each constraint the labelled program points must be in the

same function, procedure or process. Resource constraints specify upper bounds on the number of resource instances that the synthesis system may generate to implement unbound calls.

2.2.2 Hercules

The *Hercules* [42] tool provides the front-end of the behavioural synthesis subsystem. Its job is to parse HardwareC descriptions, perform high-level optimisations and generate intermediate code.

The optimisations performed at this level are similar to those which would be applied by an optimising software compiler including loop unrolling, constant propagation, common sub-expression elimination and dead-code elimination [10]. Note that all the optimisations are intra-procedural. No inter-procedural optimisations are performed.

The intermediate code (referred to as *Sequencing Intermediate Format*—SIF) is structured as a directed graph with nodes representing operations (such as assignments, conditionals etc.) and edges representing control- and data-dependencies. Parallelism is represented explicitly in SIF. The Hercules system analyses the dependencies in the HardwareC source and attempts to generate SIF with as much parallelism as possible.

2.2.3 Hebe

Hebe [87] takes its input in SIF form and performs the tasks of resource-binding and scheduling. Hebe chooses to perform resource-binding first and then scheduling. The whole process is repeated multiple times to investigate the effects of different possible binding alternatives. The tasks performed by Hebe can be characterised more precisely as follows:

- *Select Binding Configuration* – a binding configuration is a mapping between verticies of the SIF graph and hardware-level resources. (Note that the binding configuration must satisfy the resource constraints specified in the HardwareC specification.) Hebe supports both exact and heuristic search of binding configurations. In the latter case, the evaluation function attempts to minimise the chip-area, interconnect and latency of the final implementation.
- *Resolve Resource Conflicts* [88] – once a resource binding is selected, operations bound to the same resource are statically serialised to ensure that they will be accessed under mutual exclusion. A branch-and-bound search is used to explore the ordering alternatives. If a serialisation cannot be found then the binding configuration is discarded and the whole process is repeated.
- *Relative Scheduling* [89] – the scheduling phase has special provision for dealing with operations whose execution times are not statically bounded. We discuss relative scheduling further in Chapter 4 where we contrast its expressivity with our *Soft Scheduling* technique.

One of the key differences between HardwareC and Verilog/VHDL is that, whereas the latter were initially intended for simulation, the former was designed specifically with synthesis in mind. As a result, all valid HardwareC programs are synthesisable (subject to the tools finding an implementation which matches the specified constraints).

HardwareC's treatment of functions as shared resources is of particular relevance to our research. We adopt a similar technique in our SAFL hardware description language (defined in Chapter 3).

2.3 Functional Languages

There is a large body of work on using functional programming languages to describe circuits at the *structural* level; this methodology offers a number of potential advantages over structural Verilog/VHDL:

- Function composition is a very useful notion, allowing circuits to be composed directly (without the clutter of defining intermediate signal names explicitly).
- Higher-order functions provide a concise and flexible notation for abstracting commonly used circuit structures (e.g. carry chains).
- The polymorphic type systems commonly used in functional languages offer a greater degree of flexibility than simply typed VHDL and a greater degree of security than untyped Verilog.
- The layout of a circuit can be represented elegantly and concisely by using higher-order combinators which encapsulate a notion of geometry.

In the remainder of this section we explore these issues with reference to a number of functional hardware description languages. (Chapter 8 extends the ideas described here by embedding a functional language for structural hardware description into SAFL, our functional language for behavioural hardware description.)

2.3.1 μFP: An Algebra for VLSI Specification

Sheeran's μFP [132] is a structural HDL based on Backus' FP [15, 16]. Whereas functions in FP take a single input and produce a single output, functions in μFP map a *stream* of inputs onto a stream of outputs. The nth element of a stream corresponds to the value of a signal at time $t = n$. In this model time is a discrete quantity, providing a convenient mapping between time intervals and hardware-level clock cycles[3].

Like FP, the μFP language provides a set of higher-order *combining forms*: functions which map functions to functions. Some of the important combining forms (referred to as $\circ$, α, $/R$ and $/L$) are given in the table below:

[3] μFP was designed with synchronous circuits in mind.

Combining Form	Description
$(f \circ g)(x)$	Function composition: $f(g(x))$
$(\alpha f)\ \langle x_1, \ldots, x_n \rangle$	Map f onto each element of sequence $\langle x_1, \ldots, x_n \rangle$
$(/R\ f)\ \langle x_1, \ldots, x_n \rangle$	Fold (binary) function f onto sequence $\langle x_1, \ldots, x_n \rangle$ to compute $f(x_1, f(\ldots, f(x_{n-1}, x_n)\ldots)$
$(/L\ f)\ \langle x_1, \ldots, x_n \rangle$	Fold (binary) function f onto sequence $\langle x_1, \ldots, x_n \rangle$ to compute $f(\ldots f(f(x_1, x_2), x_3)\ldots), x_n)$

As well as the stream semantics [132] (defined by means of a translation from μFP to FP) the combining forms of μFP also have a circuit-level geometric interpretation. Figure 2.5 shows how circuits corresponding to the $/L$, $/R$ and α combinators are constructed.

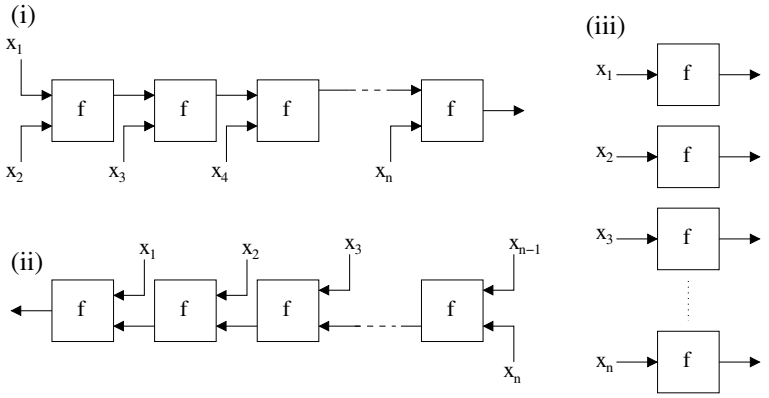


Fig. 2.5. The geometrical (circuit-level) interpretation of some μFP combining forms. (i) $(/L\ f)\ \langle x_1, x_2, \ldots, x_n \rangle$; (ii) $(/R\ f)\ \langle x_1, x_2, \ldots, x_n \rangle$; (iii) $(\alpha f)\ \langle x_1, x_2, \ldots, x_n \rangle$

The μFP language augments Backus' FP with a new combining form μ which represents state. Given a function f of type $\alpha \times \beta \rightarrow \gamma \times \beta$, μf is a function of type $\alpha \rightarrow \gamma$ which has internal state. Figure 2.6 shows how μf is realised at the circuit-level. A feedback loop is created using a clocked-latch to hold state.

The simplicity of the μFP language makes it ideal for program transformation. Transformations are expressed as a series of algebraic identities. The μFP design philosophy involves *deriving* an efficient implementation from an abstract specification by repeated application of transformation rules.

In later work, μFP was refined into a *relational* VLSI specification language called Ruby [80]. In Ruby a circuit is modelled as a binary relation between input and output signals. Guo and Luk [59] present a method for compiling Ruby specifications to FPGAs. This exploits the structure of Ruby specifications in order to generate efficient layouts using a syntax-directed translation function.

Fig. 2.6. The hardware-level realisation of the μ combinator— (*i*) function $f : \alpha \times \beta \rightarrow \gamma \times \beta$; (*ii*) the effect of applying the μ combinator, yielding a function $\mu f : \alpha \rightarrow \gamma$

2.3.2 Embedding HDLs in General-Purpose Functional Languages

A number of researchers have investigated embedding HDLs in existing functional languages. For example, the HDRE (pronounced "hydra") system [109] was originally implemented on top of the functional programming language Daisy [85]; the Lava HDL [25] is embedded in Haskell [3].

Both HDRE and Lava use *alternative interpretations of basis functions* in order to perform both simulation and net-list generation from the same specification [110]. To illustrate this technique we give ML [101] code for boolean basis functions (`AND`, `OR` and `NOT`) in both a *behavioural interpretation* and a *structural interpretation*. We then demonstrate the effect of executing a simple program under each of these interpretations.

```
fun AND (true,true) = true
  | AND _ = false

fun OR (false,false) = false
  | OR _ = true

fun NOT true = false
  | NOT false = true
```

Fig. 2.7. Behavioural interpretation of basis functions `AND`, `OR` and `NOT`

The behavioural interpretation of the basis functions is simply the usual interpretation of the function in the boolean domain (see Figure 2.7). In contrast, the structural interpretation of the basis functions is presented in Figure 2.8. Note that Figure 2.8 assumes the existence of a function, `fresh_wire_name()`, which when invoked returns a string representing the name of a fresh wire.

Let us now consider a simple specification of a simple combinatorial circuit defined in terms of our basis functions `AND`, `OR` and `NOT`:

```
fun xor(x,y) = OR( AND(NOT(x), y), AND(x, NOT(y)) )
```

Under the behavioural interpretation of our basis functions we can execute this specification as a boolean function. For example:

```
- xor(false,true);
val it = true : bool

- xor(true,true);
val it = false : bool
```

Under the structural interpretation of the basis functions, execution of the specification generates a Verilog-style netlist and returns the name of the function's output wire:

```
- xor("input1","input2");

     not(wire1,input1)
     and(wire2,wire1,input2)
     not(wire3,input2)
     and(wire4,input1,wire3)
     or(wire5,wire2,wire4)

val it = "wire5" : string
```

Compare the simplicity of the functional `xor` specification with the equivalent Verilog-style net-list. One of the advantages of using functional languages to describe the structure of hardware is that function composition can be used to connect hardware blocks without explicitly declaring intermediate wires.

```
fun diadic fname (in_wire1,in_wire2) =
  let val out_wire : string = fresh_wire_name()
   in print (fname ^ "(" ^ out_wire ^ "," ^ in_wire1 ^
              "," ^ in_wire2 ^ ")\n");
      out_wire
  end

fun AND (in_wire1,in_wire2) = diadic "and" (in_wire1,in_wire2)
fun OR  (in_wire1,in_wire2) = diadic "or" (in_wire1,in_wire2)

fun NOT in_wire =
  let val out_wire : string = fresh_wire_name()
   in print ("not(" ^ out_wire ^ "," ^ in_wire ^ ")");
      print "\n"  (* new line *);
      out_wire
  end
```

Fig. 2.8. Structural interpretation of basis functions `AND`, `OR` and `NOT`

Although for the purposes of this example we embedded a simple HDL in the Call-By-Value (CBV) language ML, work on embedding HDLs usually focuses on lazy functional languages. For describing combinatorial (acyclic) hardware, CBV and lazy languages are equally appropriate. However, lazy evaluation has

advantages when circuits contain feedback loops. The reason for this is that circuits with feedback loops are typically represented as mutually recursive stream equations. Although infinite-streams can be represented in a CBV language, simulating delayed evaluation by explicitly constructing *thunks*[4] [120], cyclic circuit definitions are more elegant in a naturally lazy language. The flip side is that lazy languages have more difficulty operating on graphs with shared nodes (a particularly useful feature when describing circuits containing feedback). Classen and Sands propose an extension for Haskell which makes graph sharing observable [37]. Their work is based on immutable references which support test for equality and is motivated by a desire to embed an HDL in Haskell.

A number of large designs have been simulated and synthesised using HDLs embedded in functional programming languages. Classen et al describe the synthesis of a parameterisable sorter core [38] using the Lava system. Their implementation (on a Xilinx FPGA) is significantly smaller than when state-of-the-art tools were used to synthesise an equivalent sorter described in Verilog.

The Hawk [96] system is another HDL embedded in Haskell designed specifically to facilitate the description and simulation of complex microprocessors. The system has been used to specify a modern microarchitecture based on the Intel P6 [39].

Johnson's DDD (Digital Design Derivation) system [79] manipulates functional S-expressions in the Lisp dialect *Scheme* [125]. The system allows engineers to refine a behavioural specification into a structural implementation by means of a library of correctness-preserving transformations. DDD has been tested on a number of large case-studies, including the derivation of a complete microprocessor [27].

2.4 Term Rewriting Systems

Hoe and Arvind describe TRAC [69]: a hardware synthesis system which generates synchronous hardware from a high-level specification expressed in a term-rewriting system [14]. Broadly speaking, terms correspond to states and rules correspond to combinatorial logic which calculates the next state of the system. Restrictions imposed on the structure of rewrite rules facilitate the static allocation of storage. These closely correspond to the tail-recursion restriction imposed on SAFL programs to achieve static allocation (see Chapter 3).

As a motivating example, term-rewriting systems are used to specify and verify the equivalence of two simple RISC processors: a simple processor without any advanced features, and a more complicated variant with pipelining and speculative execution [12]. Two languages based on term rewriting systems are investigated [68], one based on the syntax of Parallel Haskell [108], the other based on the syntax of C [83].

[4] Thunks are closures with a `unit` argument (e.g. $\lambda().5$).

2.5 Occam/CSP-Based Approaches

Hoare's CSP process calculus [31] and its related programming language Occam [76] have proved popular for describing parallel systems. A number of researchers have investigated the use of Occam/CSP-based languages for hardware description and synthesis.

2.5.1 Handel and Handel-C

The *Handel* language is a subset of Occam used for hardware synthesis research at Oxford University during the early nineties. A compiler was built to translate Handel into hardware, specifically targeting Xilinx FPGAs [112].

The compilation techniques employed are entirely syntax-directed. No scheduling or binding is performed—each occurrence of an operator in the source (e.g. '$*$') leads to the generation of a new combinatorial functional-unit at the hardware-level (e.g. a new combinatorial multiplier). All function calls are inlined, leading to the possibility of an exponential blow-up in code size; all expressions are translated into purely combinatorial hardware.

Despite the crude compilation strategy, the Handel system was used in a number of interesting hardware/software co-design projects. Transformations for converting a Handel program into a specialised micro-processor and an associated machine-code program were investigated [113]. Dynamically reconfiguring the parameterised processor (by reprogramming an FPGA at run-time) was also considered [114].

More recently, the Handel language has evolved into a commercial product called *Handel-C* [2]. The marketing literature is careful to distance itself from Occam, describing Handel-C as "a subset of C with extensions"[5]. However, in practice *subset* corresponds to removing the features of C that do not appear in Handel and *with extensions* corresponds to adding the features of Handel that are not in C (i.e. synchronous channels and parallel composition). Thus we can safely deduce that Handel-C is in fact Occam with a C-like syntax.

Although the Handel-C development environment is significantly more sophisticated than the Handel equivalent, the hardware compilation process is still essentially syntax-directed; no scheduling or binding is performed.

2.5.2 Tangram and Balsa

The *Tangram* system started life as a PhD Thesis from Eindhoven University of Technology [137] after which it was adopted by Philip's Research Laboratories and developed further [138]. The Tangram compiler takes a circuit specification expressed in a CSP-like language (with parallel composition and synchronous channels) and generates an *asynchronous* circuit.

[5] Of course, all languages are extended subsets of each other. For language extensions, $\mathcal{X}$, we observe: $\forall l_1 l_2 \exists \mathcal{X}.l_1 \subseteq (l_2 \cup \mathcal{X})$.

One of the major contributions of the Tangram project was the development of *handshake circuits*: an intermediate representation which both (*i*) abstracts low-level technology-specific details and (*ii*) supports an elegant translation to asynchronous hardware. A handshake circuit is built by composing a set of primitive *handshake components* to form a graph. The terminals of handshake components incorporate request/acknowledge signals which explicitly signal the presence/receipt of data (respectively). Since handshake components all rely on explicit request/acknowledge signalling, they can be connected together to form a system which does not require a global clock for synchronisation.

The compilation process consists of two phases: the Tangram language is first compiled into handshake circuits and then from this form into an asynchronous circuit. The first phase (compilation to handshake circuits) is syntax directed; the second phase (compilation to asynchronous circuits) involves a one-to-one mapping from *handshake components* to their circuit-level realisation.

In a sense the Tangram compiler is quite simple; it does not perform allocation, binding or scheduling (the tasks usually associated with high-level synthesis systems). What makes Tangram interesting and differentiates it from the vast majority of high-level synthesis systems is that it targets asynchronous circuits. A number of asynchronous ICs have been compiled using Tangram including a Compact Disc error decoding circuit [84] and an asynchronous version of the Intel 80C51 microcontroller [139] which consumes about a quarter of the power of its synchronous counterpart.

In 1998 a redesigned version of the Tangram language and its associated silicon compiler was released within Philips. Based on the experiences of Philips' engineers, who had by that time been using Tangram for several years, a number of new language features were added. Most importantly, primitives such as `sample`, `wait`, and `write` were provided to manipulate external signals without the requirement for an external handshake [121].

The Balsa synthesis system [19], developed at the University of Manchester, is similar to Tangram, translating CSP-like input descriptions into asynchronous circuits for implementation on FPGAs or standard-cell technologies. The striking similarity between Balsa and Tangram is not coincidental: Balsa was developed shortly after the EXACT project (EXplotation of Asynchronous Circuit Techniques—ESPIRIT project 6143, 1992–94) during which Manchester had access to the Tangram tools [45]. Balsa has been used to synthesise the DMA controller for the AMULET3i macrocell (which consists of an AMULET3 asynchronous microprocessor [51], 8Kb RAM, 16Kb ROM and a simple peripheral bus interface). The most important difference between Tangram and Balsa from the perspective of the research community is political rather than technical: whereas Tangram is entirely Philips proprietary, the Balsa tools and associated research papers are freely available.

2.6 Synchronous Languages

Computerised systems can be divided into three broad categories [64]:

- *Transformational Systems* compute output values from input values and then stop (e.g. numerical computation programs, compilers).
- *Interactive Systems* constantly interact with their environment, but do so at a speed governed by the system (rather than the environment). Examples of interactive systems include operating systems: an operating system listens to the user when it can and delivers services as and when they are available.
- *Reactive systems* (also known as *reflex systems*) also constantly interact with their environment, but do so at a pace that is dictated by the environment. Signal processors are a typical example of reactive systems—a stream of data comes into the system (at a speed determined by the environment); the signal processor must produce corresponding outputs at as fast as the input data arrives.

Given this taxonomy, concurrent programming languages can also be categorised accordingly. The synchronous language community divide languages in two distinct classes that they call the *synchronous* class and the *asynchronous* class. The class of asynchronous languages contains classical concurrent languages such as CSP [31], Occam [76] and Ada [9]. In such languages concurrent processes are viewed as independent execution units, each proceeding at their own pace. Inter-process communication is facilitated by mechanisms such as message passing or rendezvous. In this framework communication is asynchronous in the sense that an arbitrary amount of time can pass between the desire for communication and its actual completion[6]. Asynchronous languages are well suited to describing transformational and interactive systems but are often not so appropriate where reactive systems are concerned [20].

The most notable examples of synchronous languages are Esterel [23], Lustre [61], Signal [58] and Statecharts [65]. Lustre and Signal are declarative languages which are designed with real-time DSP systems in mind. In contrast Esterel and Statecharts are imperative languages which focus instead on describing real-time controllers. Although the problem domains are varied, the common factor that unifies all synchronous languages is that they deal with *reactive* systems subject to strict timing constraints. In this section we choose to give an overview of synchronous languages in general rather than focusing on the specifics of particular languages. (For a good description of the specifics the reader is referred to Halbwachs' book [60].)

Programs written in synchronous languages are viewed as reacting *instantaneously* to their inputs by producing the required outputs. Of course in practice this is an idealised assumption as one cannot construct infinitely fast computers. In reality synchronous programs are implemented as cycle-based systems where

[6] Note that there is a conflict of terminology here as rendezvous is sometimes referred to as synchronous communication—to avoid confusion the synchronous language community uses the term *synchronising communication* rather than synchronous communication in this context.

a cycle corresponds to "read inputs; perform computation; write outputs". However, the assumption of an instantaneous response to the environment (sometimes referred to as the *perfect synchrony hypothesis*) simplifies the semantic model of the languages considerably and removes issues regarding computation time from specifications: the designer hopes that the compiler will produce a design with a cycle time short enough to meet the system's real-time constraints. There is a direct analogy with synchronous circuit design here. When designing a synchronous circuit at the netlist level one often makes the assumption that gates and wires have no delays[7]. It is only when CAD tools map the specification to hardware that the engineer can determine whether the resulting clock speed is sufficient to meet real timing constraints.

Another distinction between synchronous and asynchronous languages can be seen in the style of their Inter-Process Communication (IPC) primitives. Whereas asynchronous languages rely on rendezvous communication or message passing, synchronous languages perform IPC by means of instantaneous broadcast, the receiver receiving the message at exactly the same time it is sent. In contrast to asynchronous languages, which usually support non-determinism, (e.g. Occam's `ALT` construct), synchronous languages are fully deterministic: for any given input sequence there is exactly one corresponding output sequence that can be generated.

Complex systems often require the capabilities of both synchronous and asynchronous languages. For example, a robot driver must use a specific reactive program to control each articulation, but the global robot control may be asynchronous due to limitations of networking capabilities. To address this issue Berry *et al* motivate *Communicating Reactive Processes* [24]: a formalism which supports the capabilities of asynchronous and synchronous languages by unifying Esterel and CSP.

Research has shown that synchronous languages can be compiled to hardware efficiently [21, 62]. Although current compilers for synchronous languages only target synchronous hardware, there is no fundamental reason why synchronous languages should not be compiled to asynchronous hardware. However, it may prove to be the case that hard real-time guarantees are less forthcoming in the asynchronous domain (e.g. for every input, does the circuit always generate its outputs in a timely manner?). Further research is required to investigate whether synchronous languages can be usefully implemented as asynchronous hardware.

2.7 Summary

In this chapter we have reviewed a number of existing hardware synthesis methodologies. We continue by presenting SAFL, our functional language for hardware synthesis.

[7] Of course, this is a bit of an over-simplification as a good engineer will be thinking about critical paths etc. whilst drawing up a netlist. Despite this, however, there is still no *precise* timing information known at this stage—the design philosophy is very much "suck it and see"!

3

The SAFL Language

SAFL, which stands for Statically Allocated Functional Language, is a behavioural HDL which is used throughout this monograph as a vehicle to explore high-level synthesis. Although the language is syntactically and semantically simple, it is expressive enough to form the core of a behavioural hardware synthesis system supporting high-level analysis, optimisation and transformation.

The methodology that underlies our research involves using a small and elegant language to explore new possibilities in the field of high-level hardware description and synthesis. The simplicity of SAFL provides two key benefits: firstly it allows us to be more productive by minimising implementation and development times; secondly it enables us to present our ideas clearly and concisely without becoming entangled in the complexity of a fully-featured programming language. Of course, an obvious concern with this methodology is that using an unrealistically simplistic language to investigate analysis and compilation methods may result in the development of techniques which are *only* applicable to such unrealistic languages. To rebut this argument we dedicate Chapters 7 and 8 to extending SAFL with the capabilities one would expect from an industrial-strength HDL and demonstrate that the analysis and compilation techniques of Chapters 4 and 6 scale accordingly.

In this chapter we introduce the SAFL language and outline how it is used for hardware description and synthesis. Having described the motivation behind SAFL (Section 3.1) the language's syntax and semantics are presented (Section 3.2). A number of important concepts are defined including *static allocation* (Section 3.2.1) and *resource awareness* (Section 3.3.2). We introduce the idea of using source-to-source program transformations on SAFL specifications to facilitate architectural exploration. Some simple transformations are described and small examples, based on Burstall/Darlington's *fold/unfold* transformations [29], are presented (Section 3.3.3). Finally we show how mutual-recursion can be dealt with in the SAFL framework.

3.1 Motivation

Recall that in Chapter 1 we argued that structural blocks (employed as an abstraction mechanism by many behavioural HDLs) are not suitable for high-

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 35–50, 2004.

level hardware specification. SAFL is based on the observation that many of the problems associated with structural blocks (see Section 1.3.1) can be alleviated by structuring code as a series of function definitions. Firstly, the properties of functions make it easier to reason about the effects of program transformations. As a result, source-level program transformation becomes a viable technique for architectural exploration. Secondly, the "invoke and wait for result" interface provided by functions removes the programmer from the burden of explicitly specifying ad hoc inter-module control- and data-flow mechanisms. As well as making the code shorter and easier to read, the implicit control-flow information encapsulated in the function-abstraction increases the scope for global compiler analysis and optimisation (see Section 3.3.4).

A principal aim of this work is to adopt an aggressively high-level stance with respect to hardware specification. Our goal is to build a system in which (*i*) the programming language is clean (e.g. no 'implementation-defined subsets'); (*ii*) the *logical* structure can be specified early in the design process but its realisation as a *physical* structure can easily be modified even at late stages of development; and (*iii*) programs are susceptible to compiler analysis and optimisation facilitating the automatic synthesis of efficient circuits. These requirements are clearly reflected in the design of SAFL, a first-order, functional, concurrent language which:

- can be *statically allocated*—all variables are allocated to fixed storage locations at compile time—there is no stack or heap; and
- has independent sub-expressions evaluated in parallel.

While this concept might seem rather odd in terms of the capabilities of modern processor instruction sets, the view is that it neatly abstracts the primitives available to a hardware designer. The desire for static allocation is motivated by an observation that dynamically-allocated storage does not map well onto silicon: an addressable global store leads to a von Neumann bottleneck which inhibits the natural parallelism of a circuit.

3.2 Language Definition

SAFL is essentially a language of first-order recurrence equations with an ML-like syntax [101] and a call-by-value semantics. A user program consists of a number of function definitions declared using the `fun` keyword.

Let c range over a set of constants, x over variables (occurring in `let` declarations or as formal parameters), a over primitive functions (such as addition) and f over user-defined functions. For typographical convenience we abbreviate formal parameter lists $(x_1, \ldots, x_k)$ and actual parameter lists $(e_1, \ldots, e_k)$ to $\vec{x}$ and $\vec{e}$ respectively; the same abbreviations are used in `let` definitions. The core SAFL language has an *abstract syntax* of:

– terms e given by:

$$e ::= c \mid x \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{let}\ \vec{x} = \vec{e}\ \texttt{in}\ e_0 \mid a(e_1, \ldots, e_{arity(a)}) \mid f(e_1, \ldots, e_{arity(f)})$$

– programs p given by:

$$p ::= \texttt{fun}\ f^1(\vec{x}) = e_1 \quad \ldots \quad \texttt{fun}\ f^n(\vec{x}) = e_n$$

Programs have a distinguished function `main` (normally f^n) which represents an external world interface—at the hardware level it accepts values on an input port and may later produce a value on an output port.

Throughout this document SAFL's *abstract syntax* (given above) is used to simplify the definition of analysis and compilation techniques. In contrast, SAFL program fragments are given using the *concrete syntax* defined in Section 3.2.4.

3.2.1 Static Allocation

We say that SAFL is *statically allocatable*, meaning that we can allocate all storage required for a SAFL program at compile-time. In order to achieve static allocability we impose the restriction that all recursive calls must be tail-recursive. This is formalised as a well-formedness check: define the *tailcall contexts*, $\mathcal{TC}$ by

$$\mathcal{TC} ::= [\,] \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ \mathcal{TC} \mid \texttt{if}\ e_1\ \texttt{then}\ \mathcal{TC}\ \texttt{else}\ e_3 \mid \texttt{let}\ \vec{x} = \vec{e}\ \texttt{in}\ \mathcal{TC}$$

The well-formedness condition is then that, for every user-function application $f^i(\vec{e})$ within function definition $f^j(\vec{x}) = e_j$, we have that:

$$i < j \vee (i = j \wedge \exists (C \in \mathcal{TC}).e_j = C[f^i(\vec{e})])$$

The first part ($i < j$) is merely static scoping (definitions are in scope if previously declared) while the second part says that if the call is recursive ($i = j$) then it must be in tailcall context.

To emphasise the hardware connection we define the *area* of a SAFL program to be the total space required for its execution. Due to static allocation we see that *area* is $O(\textit{length of program})$.

3.2.2 Integrating with External Hardware Components

Although the SAFL language in itself is powerful enough to specify many hardware designs in full (e.g. Chapter 10 contains a DES Encrypter/Decrypter written in SAFL) there are often circumstances where a SAFL hardware design must interface with external hardware components. In the SAFL language, as implemented, facility is provided to access externally defined hardware via a function call interface. The signature of external functions is given using the `extern` keyword. For example, one may declare an interface to an external RAM block as follows:

```
extern memory(address:16, value:32, write:1):32
```

where the $:\langle n \rangle$ annotations are used to specify the bit-widths of the argument and result types (see Section 3.2.4).

Whilst the details of the interfacing mechanism are described in Chapter 5, it is worth noting at this stage that external calls may be side-effecting. We take an ML-style view of side-effects: since SAFL is a strict language it is easy to reason about where side-effects will occur in a program's execution. Of course, we intend that programmers write as much as possible of their specification in SAFL, only relying on external calls when absolutely necessary.

3.2.3 Semantics

At an abstract level, the semantics of SAFL is straightforward; Figure 3.1 presents a Structural Operational Semantics [122] for the language in only seven rules. However, although this semantics is able to determine the result of a SAFL program's execution, it is does not model many of the important properties of the language such as concurrency or calls to external (possibly side-effecting) functions. These issues are dealt with formally in Chapter 7 where a finer-grained semantics based on the *Chemical Abstract Machine* [22] is presented. For now, since most of our SAFL examples are purely functional (i.e. do not rely on `extern`al functions), the semantics of Figure 3.1 suffices to define the meaning of SAFL programs.

One aspect of SAFL which is highlighted by the operational semantics given here is its call-by-value nature: the ($app\Downarrow$) rule specifies that a function call's arguments must be evaluated fully before the body of the called function can be executed. Our decision to use eager evaluation, as opposed to lazy evaluation, is based on an observation that a strict semantics offers more opportunity for concurrent execution. Whereas with lazy evaluation one always has a single (i.e. sequential) idea of 'the next redex'[1], call-by-value execution permits all function-call arguments to be evaluated in parallel. Concurrency is a very important consideration in the design of an HDL, since, in contrast to software, hardware is inherently parallel.

3.2.4 Concrete Syntax

Whilst for many constructs SAFL's concrete syntax mirrors the abstract syntax given earlier in this section, there are a few differences worth highlighting. Firstly, as in `ML`, the `val` keyword is used to introduce `let` declarations, and the `end` keyword delimits the end of the body. For example:

```
let val x = f(3)
    val y = h(4)
 in g(x,y)
end
```

[1] Note that *strictness analysis* [36] of a lazy language can be used to determine which of a function's parameters may safely be called by value (and hence those parameters which may be executed concurrently). However, decidability issues limit the accuracy of the analysis meaning that in practice opportunity for parallelism is not exploited fully.

Let S be a function mapping (immutable) variables onto their integer values. As before we use $\vec{x}$ as shorthand for $x_1, \ldots, x_k$ (similarly for $\vec{e}$ and $\vec{v}$). $S[\vec{x} \mapsto \vec{v}]$ represents a function which is as S but maps each x_i onto v_i for $1 \leq i \leq k$. Let $\mathcal{F}$ be a global *function environment* mapping function names onto the SAFL expressions representing their bodies. We use $x_1, \ldots, x_k$ to refer to a function's formal parameters. Then a big-step transition relation, $\Downarrow$, for SAFL can be defined with the following rules:

$$\frac{}{\langle S, c\rangle \Downarrow c}\,(con \Downarrow)$$

$$\frac{}{\langle S, x\rangle \Downarrow v}\,(var \Downarrow) \quad \text{if } x \in dom(S) \wedge v = S(x)$$

$$\frac{\langle S, e_1\rangle \Downarrow v_1 \quad \ldots \quad \langle S, e_k\rangle \Downarrow v_k}{\langle S, a(e_1, \ldots, e_k)\rangle \Downarrow v}\,(op \Downarrow) \quad \text{where } v = a(v_1, v_2, \ldots, v_n)$$

$$\frac{\langle S, e_0\rangle \Downarrow 0 \quad \langle S, e_2\rangle \Downarrow v}{\langle S, \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2\rangle \Downarrow v}\,(if_0 \Downarrow)$$

$$\frac{\langle S, e_0\rangle \Downarrow r \quad \langle S, e_1\rangle \Downarrow v}{\langle S, \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2\rangle \Downarrow v}\,(if_1 \Downarrow) \quad \text{provided } r \neq 0$$

$$\frac{\langle S, e_1\rangle \Downarrow v_1 \quad \ldots \quad \langle S, e_k\rangle \Downarrow v_k \quad \langle S[\vec{x} \mapsto \vec{v}], \mathcal{F}(f)\rangle \Downarrow r}{\langle S, f(e_1, \ldots, e_k)\rangle \Downarrow r}\,(app \Downarrow)$$

$$\frac{\langle S, e_1\rangle \Downarrow v_1 \quad \ldots \quad \langle S, e_k\rangle \Downarrow v_k \quad \langle S[\vec{x} \mapsto \vec{v}], e_0\rangle \Downarrow r}{\langle S, \texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0\rangle \Downarrow r}\,(let \Downarrow)$$

Fig. 3.1. A big-step transition relation for SAFL programs

specifies that the values of `x` and `y` are to be computed and then used as arguments to the function `g`. Variables defined in a `let` construct are only in scope in the body of the `let`[2].

Type System

For most of the examples in the text, we will omit type information. However, the types of all variables must clearly be known to compile the appropriate width

[2] Note that we adopt a different syntactic convention from ML in this respect. In ML, `var`-declared variables are bound in subsequent declarations within the same `let`-block (unless the `and` keyword is used explicitly).

of hardware register. All SAFL variables are explicitly typed with a type bit_n; in concrete syntax we use the form $\texttt{x:}bit_n$ (where n is an integer constant) to annotate variables at their introduction. All constants also have a type (explicitly given or inferred from their value). Built-in functions, such as addition and concatenation, may have a family of types (e.g. $bit_n * bit_n \mapsto bit_n$ or $bit_m * bit_n \mapsto bit_{m+n}$); the result type (and hardware generated) depends on the types of the arguments. User-defined functions also require the type of result to be provided when it cannot be inferred from the type of result; this is only really necessary for odd (and useless) forms like $\texttt{fun f(x : 8)} = \texttt{f(x)}$. The value, `()`, of width 0 plays a special rôle and has type *unit* (which is synonymous with bit_0). We adopt the convention that all functions return a single result; side-effecting functions which do not need to return a value can return `()` instead.

To make examples more readable we introduce the notion of record types, extending the top-level syntax of SAFL programs (p) as follows:

$$
\begin{aligned}
d ::= {} & \texttt{fun } f(\vec{x}) : t = e_1 \\
\mid {} & \texttt{type } t = \{\mathit{field}_1 : t_1,\ \ldots,\ \mathit{field}_k : t_k\}
\end{aligned}
$$

$$
p ::= d \ldots d
$$

Type variables t_k range over both integers (in which case they represent bit widths) and other (previously defined) record types. We use the usual dot-notation ($r.x$) to select field x from record r.

We choose not to focus on type systems for HDLs in this monograph: the type system employed here is just the "bare minimum" required to make SAFL usable. A topic for future work is to investigate the impact of applying more sophisticated type systems (e.g. the introduction of parametric polymorphism [99]) to SAFL.

Operators

Figure 3.4 shows SAFL's arithmetic, logical and relational primitive operators. A number of other operations are also provided. These are discussed briefly below. Note that these operators are merely concrete syntax (providing convenient special cases of the abstract syntax's $a(e_1, \ldots, e_n)$ production).

- We write `e[n:m]` to select a bit-field `[n..m]` from the result of expression `e` (where `n` and `m` are integer constants).
- Conversely, we use the special operator *join* to concatenate values at the bit-level. The result of

  $$\texttt{join}(e_1, e_2, \ldots, e_n)$$

  is a single value which is the bit-level concatenation of the results of expressions $e_1, \ldots, e_n$.

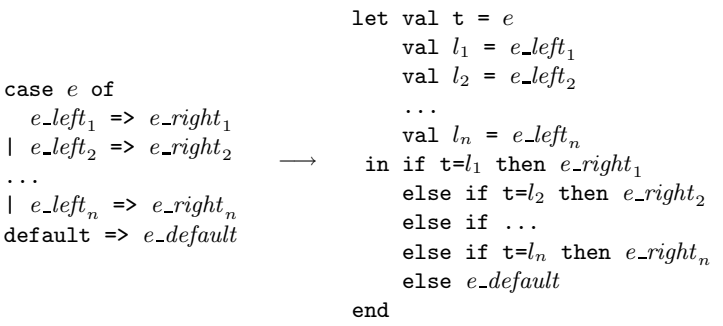


Fig. 3.2. Translating the `case` statement into core SAFL

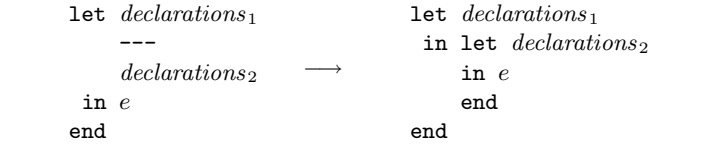


Fig. 3.3. Translating let barriers "`---`" into core SAFL

Syntactic Sugarings

It is convenient to extend SAFL with a number of syntactic sugarings:

- We define a `case` construct which is translated into code which evaluates the left-hand expressions in parallel and then performs a series of iterated tests (see Figure 3.2).
- We define sequential composition of expressions $e_1; e_2$ as:

  `let val _ =` e_1 `in` e_2 `end`

  and data-flow parallel composition, e_1 `||` e_2, as:

  `let val x =` e_1 `val y =` e_2 `in y end`

  Where `x` and `y` are fresh variable names.
- We define the lexical symbol "`---`" as a *let-barrier* (see Figure 3.3). The let-barrier is useful because it provides us with a concise way to control the order of execution of `let`-declarations. All expressions before a let-barrier are evaluated fully before any subsequent expressions are evaluated. Our primary motivation for the let-barrier sugaring is that it enables static scheduling of shared resources to be described concisely in SAFL (see Section 4.4).

3.3 Hardware Synthesis Using SAFL

Having defined the syntax and semantics of SAFL we continue by describing the core principles involved in building a high-level synthesis system around the

Symbol(s)	Meaning
`+`, `-`, `*`, `/`, `%`	Usual arithmetic operators (unsigned)
`<=`, `>=`, `<`, `>`	Usual relational operators (unsigned)
`<>`	Test for inequality
`=`	Test for equality
`and`, `or`, `xor`, `not`	Usual bitwise logic operators
`<<`, `>>`	Logical (i.e. no sign-extension) shift operators

Fig. 3.4. SAFL's primitive operators

language. Here we introduce and motivate the concept of *resource-awareness*, present a SAFL-to-silicon design-flow and outline some of the advantages of generating hardware from SAFL specifications.

3.3.1 Automatic Generation of Parallel Hardware

Hammond [63] observes:

> "It is almost embarrassingly easy to partition a program written in a strict [functional] language [into parallel threads]. Unfortunately, the partition that results often yields a large number of very fine-grained tasks."

He uses the word *unfortunately* because his discussion takes place in the context of software, where fairly course-grained parallelism is required to ensure the overhead of `fork`/`join` does not outweigh the benefits of parallel evaluation.

In contrast, we consider the existence of "a large number of very fine-grained tasks" to be a very *fortunate* occurrence: in a silicon implementation, very fine-grained parallelism is provided with virtually no overhead[3]. The referential transparency of SAFL allows us to evaluate function-call arguments and `let`-declarations concurrently.

3.3.2 Resource Awareness

The static allocation properties of SAFL allow our compiler to enforce a direct mapping between a function definition:

$$\texttt{fun}\ \ f(\vec{x}) = e$$

and a hardware block, H_f, with output port, P_f, consisting of:

- a fixed amount of storage (registers holding values of the arguments $\vec{x}$)
- a circuit to compute e to P_f.

[3] Hardware is the exact opposite of software in this respect: separate components of a circuit naturally operate in parallel—special effort must be taken to force them to execute sequentially!

We say that our SAFL compiler is *resource aware* since each function declaration at the source-level is translated into a *single* resource at the hardware level. In this framework, multiple calls to a function f corresponds directly to sharing the resource H_f at the hardware level. We believe that resource awareness offers a number of benefits:

1. The function-resource correspondence allows SAFL to express both logical specification and physical structure without requiring any extra language features.
2. A designer is able to visualise how a SAFL program will be structured at the hardware-level. This intuitive understanding prevents one from having to "second guess" the compiler.
3. Source-to-source program transformation becomes a powerful technique. (See Section 3.3.3).

At first sight there appears to be contention between our goal to develop a truly high-level specification language on the one hand, and the decision to enforce a 1-1 correspondence between function definitions and hardware resources on the other. Recall that in Section 1.3.1 we claimed that a high-level hardware specification language should not fix low-level implementation details. Thus, at least on the surface, it appears that by advocating resource-awareness we have violated our own design criteria.

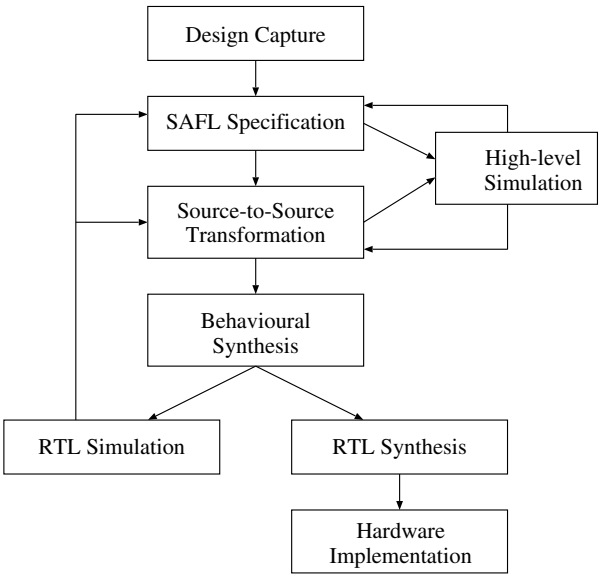


Fig. 3.5. The SAFL Design-Flow

These apparently conflicting beliefs are resolved via an extra program-transformation step in the SAFL-to-silicon design-flow (see Figure 3.5). We intend

that designers initially write SAFL specifications as clearly as possible, without considering any implementation-level issues. A source-level program transformation phase is then used to refine a given specification towards a suitable implementation whilst preserving its logical semantics. Hence SAFL frees a designer from low-level concerns in the initial phases of development but is still expressive enough to encode important implementation-level choices later in the design process.

Hardware designers frequently complain that it is difficult to visualise the circuits that will be generated by black-box HLS tools. However, the desire for an intuitive understanding of the compilation process must be traded off against the benefits of automatic analysis and optimisation: a compiler which performs significant optimisation is necessarily harder to predict than one which performs only syntax-directed translation. A strong argument in favour of resource awareness is that it explicitly defines the boundary between human specification and compiler optimisation—a SAFL program fixes the top-level circuit structure but leaves a compiler free to optimise function-internals and inter-function communication structures.

It is important to observe the interaction between parallel execution and resource-awareness. Since our compiler generates concurrent hardware we have to be careful to ensure that multiple accesses to a shared hardware resource will not occur simultaneously. This is related to the *scheduling* problem described in the first chapter. However whereas existing silicon compilers statically serialise access to shared resources we take a contrasting approach, resolving conflicts *dynamically* via arbiters generated as part of the synthesis process. (Chapter 4 considers this technique in detail and presents a global static analysis which allows redundant arbiters to be optimised away.)

3.3.3 Source-Level Program Transformation

Source-level program transformation of SAFL specifications is a powerful technique. We argue that SAFL is better suited to source-to-source manipulation than conventional HDLs for the following reasons:

- The functional properties of SAFL allow equational reasoning and hence make a wide range of transformations applicable.
- Resource-awareness gives transformations precise meaning at the design-level (e.g. we know that duplicating a function definition in the source is guaranteed to duplicate the corresponding resource in the generated circuit).

A designer can explore a wide range of hardware implementations by repeatedly transforming an initial specification. Here we illustrate this technique by showing how Burstall and Darlington's fold/unfold transformations [29] can be used to trade area for time. (Chapter 9 investigates more complex transformations which can be used to explore a wide range of architectural tradeoffs.)

Consider the following SAFL specification:

```
fun f x = ...
fun main(x,y) = g(f(x),f(y))
```

The two calls to `f` are serialised by mutual exclusion before `g` is called. Now use fold/unfold to duplicate `f` as `f'`, replacing the second call to `f` with one to `f'`. This can be done using an unfold, a definition rule and a fold yielding

```
fun f  x = ...
fun f' x = ...
fun main(x,y) = g(f(x),f'(y))
```

The second program has more area than the original (by the size of `f`) but runs more quickly because the calls to `f(x)` and `f'(y)` execute in parallel.

```
fun mult2(x, y, acc) =
   if (x=0 or y=0) then acc
   else
     if (x<<1=0 or y>>1=0) then
           if y[0:0] then acc+x else acc
     else
           mult2(x<<2, y>>2,
                 if (y>>1)[0:0] then
                    (if y[0:0] then acc+x else acc)+(x<<1)
                 else
                    if y[0:0] then acc+x else acc)
```

Fig. 3.6. An application of the *unfold* rule to unroll the recursive structure one level

Although the example given above is trivial, we find fold/unfold to be a useful technique in choosing a hardware implementation of a given specification. Note that fold/unfold allows us to do more than resource/duplication sharing tradeoffs. For example, consider the SAFL description of a shift-add multiplier:

```
fun mult(x, y, acc) =
   if (x=0 or y=0) then acc
   else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
```

This corresponds to a single multiply unit containing an adder, two shifters and some control logic. If we now *unfold* the recursive call one level our specification is transformed into the one shown in Figure 3.6. The function `mult2` corresponds to a multiply unit with 4 adders, 6 shifters and twice as much control logic as our previous example. We have increased the area, and in return gained performance. When this specification is fed through our synchronous compiler, `mult2` is able to do 'twice as much per clock-cycle' as `mult` at the cost of a higher gate count[4]. Note, however, that `mult2` is doing needless work computing some expressions repeatedly. An application of the *abstraction* rule solves this (see Figure 3.7). Using `let` to share the output of expressions leads to a decrease in time and area: we only need to compute expressions once (less time) and we only require a single circuit to compute a single expression (less area).

[4] Of course the clock frequency may be reduced if `mult2` is on the critical path of the design.

```
fun mult3(x, y, acc) =
   if (x=0 or y=0) then acc
   else
      let val new_x = x<<1
          val new_y = y>>1
          val new_acc = if y[0:0] then acc+x else acc
      in  if (new_x=0 or new_y=0) then new_acc
               else mult3(new_x<<1, new_y>>1,
                          if new_y[0:0] then new_acc+new_x
                                        else new_acc)
      end
```

Fig. 3.7. An application of the *abstraction* rule to `mult2`

Finally, consider the application of *fold* transformations to `mult3` yielding the code shown in figure 3.8. The *fold* transformation has reduced the area of

```
fun add(x,y)  = x + y
fun lshift(x) = x << 1
fun rshift(x) = x >> 1

fun mult4(x, y, acc) =
   if (x=0 or y=0) then acc
   else
      let val new_x = lshift(x)
          val new_y = rshift(x)
          val new_acc = if y[0:0] then add(acc,x) else acc
      in  if (new_x=0 or new_y=0) then new_acc
            else mult4(lshift(new_x),rshift(new_y),
                       if new_y[0:0] then add(new_acc,new_x)
                                     else new_acc)
      end
```

Fig. 3.8. The result of applying *fold* transformations to `mult3`

the generated circuit by specifying that the `adder`, `lshift` and `rshift` units are to be shared. As a result of this, `mult4` may take longer to compute its result than `mult3` since parallelism is inhibited by the shared resources: if two expressions require access to the same shared resource then they must be evaluated sequentially. Note that in practice one must be careful that the cost of sharing a resource—i.e. the extra multiplexing and control-logic—does not lead to an *increase* in area. (An example where resource sharing inadvertently leads to an increase in chip area can be seen in the experimental results of Section 6.6.)

3.3.4 Static Analysis and Optimisation

Recall that one of the key differences between structural blocks and function definitions is that, unlike structural blocks, the function abstraction implicitly encapsulates a notion of control- and data-flow. We have already observed that the function call-return interface removes low-level details from specifications, reducing program length and making code easier to read (see Section 3.1). Here we argue that functional hardware specification greatly increases the scope for *global* compiler analysis and optimisation.

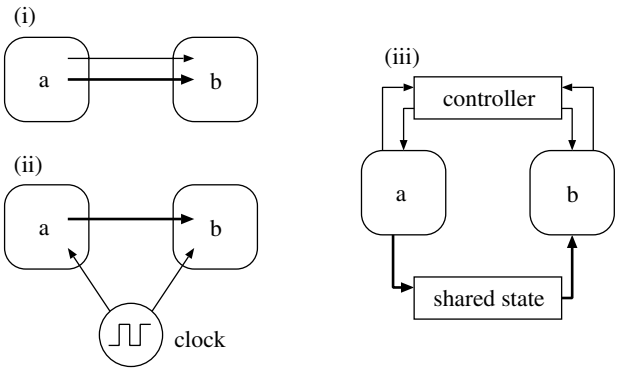


Fig. 3.9. Three methods of implementing inter-block data-flow and control-flow

Consider the case where a hardware component, a, wishes to synchronise with a component, b, in order to transfer a value into one of b's registers. There are many ways that this rendezvous could be realised at the hardware level. Figure 3.9 shows three possible implementation techniques. (Note that in this figure we adopt the convention, followed throughout this document, of using thin lines to represent control signals and thicker lines to represent data signals.) Figure 3.9(i) involves an explicit control signal which allows a to inform b when the data is ready to be latched—in this case it is assumed that b will always be ready to latch the data so no explicit backwards acknowledgement is required. Figure 3.9(ii) shows the two blocks synchronising the transfer with respect to a global clock (e.g. both a and b may adopt the convention that a transfer will take place every 15 cycles). Finally, Figure 3.9(iii) shows a and b communicating via a shared register. A separate control unit is employed to co-ordinate the transfer (i.e. to inform b when data is waiting and inform a when the data has been read). The three implementation techniques shown here are by no means exhaustive—one can easily imagine a plethora of other circuits which achieve the same effect.

Using structural blocks to define the separate components a and b inevitably results in specifications which fix low-level implementation choices (through explicit definition of control-signals, data busses, registers etc.). Although in some

sense each of the designs of Figure 3.9 perform the same high-level data-transfer, their structural specifications look radically different. This makes global analysis problematical: a pre-requisite for global static analysis is a model of how separate components of a design communicate either in terms of control-flow, data-flow or possibly both. Clearly, the problem of inferring the high-level control structures and data-flow information from structural-level specifications is very difficult in general. This is why existing HLS tools for languages such as Verilog/VHDL only perform *local* analysis and optimisation at the behavioural level.

In contrast, an equivalent SAFL specification for our communicating components would represent both a and b as separate function definitions, with function a containing a call, $b(x)$. At this higher level the intention is clear: a compiler is able to automatically infer control-flow and data-flow information across the resource boundary. Chapters 4 and 6 present global control- and data-flow analyses which allow us to optimise SAFL-generated hardware.

3.3.5 Architecture Independence

Although we try to make SAFL well-suited to describing hardware in general, we are careful not to favour the description of any particular circuit design paradigm. We say that SAFL is *architecture neutral* to mean that it abstracts a number of implementation styles. For example, a single SAFL specification could be compiled into either synchronous or asynchronous hardware; or, as a less extreme example, we could choose to implement inter-function communication either over a shared bus or as point-to-point links.

In particular, resource-awareness provides a useful abstraction since it opens up the possibility of compiling different source-level function definitions into different design styles (e.g. synchronous or asynchronous). Hardware to interface between these different design styles can then be generated automatically. We illustrate this technique in Chapter 5, where we show how SAFL can be translated into both synchronous and GALS (Globally Asynchronous Locally Synchronous) design styles; and in Chapter 9 where a source-to-source transformation for hardware/software partitioning is presented.

3.4 Aside: Dealing with Mutual Recursion

In previous work [105] we showed how SAFL could be extended to allow mutual recursion whilst maintaining both static allocation and a resource-aware compilation strategy. To show that the decision to ignore mutual recursion does not result in a loss of expressivity, Section 3.4.1 outlines a simple semantics-preserving transformation which translates mutually-recursive SAFL programs into non-mutually-recursive SAFL programs.

Let us start by extending the syntax of SAFL programs, p, with an ML-style `and` keyword:

$$p ::= \begin{array}{l} \texttt{fun } f^{11}(\vec{x}) = e_{11} \texttt{ and } \ldots \texttt{and } f^{1r_1}(\vec{x}) = e_{1r_1} \\ \ldots \\ \texttt{fun } f^{n1}(\vec{x}) = e_{n1} \texttt{ and } \ldots \texttt{and } f^{nr_n}(\vec{x}) = e_{nr_n}. \end{array}$$

We refer to a phrase of the form

$$\texttt{fun } f^{i1}(\vec{x}) = e_{i1} \texttt{ and } \ldots \texttt{and } f^{ir_i}(\vec{x}) = e_{ir_i}$$

as a (mutually recursive) function *group*. The notation f^{ij} just means the jth function of group i.

By stratifying a program into mutually-recursive function groups we are able to generalise our resource-aware compilation strategy: whereas previously a function definition constituted a single hardware-level resource, now a single function *group* corresponds to a hardware-level resource. Thus a compiler generates arbiters to enforce mutual exclusion at the granularity of groups rather than function definitions (see Chapter 4). At the hardware-level each group has a single output-port which gives the result of a call to any function within the group.

We extend the static allocation restriction presented in Section 3.2.1. For every user-function application $f^{ij}(\vec{e})$ within function definition $f^{gk}(\vec{x}) = e_{gk}$ in group g:

$$i < g \vee (i = g \wedge \exists (C \in \mathcal{TC}).e_{gk} = C[f^{ij}(\vec{e})])$$

Thus we require that mutually-recursive calls (i.e. calls between members of the same group) must be in tail-call context.

In a hardware implementation, calls to other functions in the same group correspond to a simple transfer of control and data (no locking required). In contrast calls between groups involve (*i*) acquiring mutually exclusive access to the called group; (*ii*) transferring data into the called function's registers; and (*iii*) returning the result of the call (via the group's single output port).

3.4.1 Eliminating Mutual Recursion by Transformation

The transformation for mutual-recursion elimination simply involves replacing each function group:

$$\texttt{fun } f^{i1}(\vec{x}) = e_{i1} \texttt{ and } \ldots \texttt{and } f^{ir_i}(\vec{x}) = e_{ir_i}$$

with a single function:

$$\begin{array}{l} \texttt{fun } f^{i}(s, \vec{x}) = \texttt{if } s = 1 \texttt{ then } e_{i1} \\ \qquad\qquad\quad \texttt{else if } s = 2 \texttt{ then } e_{i2} \\ \qquad\qquad\quad \ldots \\ \qquad\qquad\quad \texttt{else if } s = r_i - 1 \texttt{ then } e_{i(r_i-1)} \\ \qquad\qquad\quad \texttt{else } e_{ir_i} \end{array}$$

and replacing each call of the form, $f^{ij}(\vec{e})$, with a call $f^{i}(j, \vec{e})$. Before applying this transformation we need to ensure that all functions in the group have the

same formal parameters by (*i*) renaming and (*ii*) padding function definitions and their associated calls with extra formal parameters as necessary.

Note that if a mutually-recursive SAFL program satisfies the static allocation restriction given in the previous section, then this transformation ensures that the resulting non-mutually-recursive SAFL program satisfies the static allocation restriction given in Section 3.2.1.

3.5 Related Work

The motivation for static allocation is not new. Gomard and Sestoft [57] describe *globalization* which detects when stack or heap allocation of function parameters can be implemented more efficiently with global variables. However, whereas globalization is an optimisation which may in some circumstances improve performance, in our work static allocation is a fundamental property of SAFL enforced by the syntactic restrictions described in Section 3.2.1.

Hofmann [70] describes a type system which allows space pre-allocated for argument data-structures to be re-used by in-place update. Boundedness there means that no new heap space is allocated although stack space may be unbounded. As such our notion of static allocatability is more restrictive. In a similar vein sized types have been proposed as a means of statically verifying that a program runs in bounded space [72]. In this scheme a list containing n elements each of type α would have the type $List_n(\alpha)$. Sized types have been integrated into a variant of ML designed for programming embedded systems [71].

Johnson's Digital Design Derivation (DDD) system [26] uses a scheme-like language to describe circuit behaviour. A series of semantics-preserving transformations are presented which can be used to refine a behavioural specification into a circuit structure; the transformations are applied manually by an engineer. In some ways this is similar to our use of semantics-preserving transformations on SAFL specifications for architectural exploration. However, although we advocate the use of source-level transformations to explore architectural tradeoffs (including allocation, binding and scheduling), SAFL specifications are translated to hardware automatically using our optimising silicon compiler. In contrast, in the DDD framework it is the manually applied transformations themselves that *are* the compilation process.

3.6 Summary

We have defined syntax and semantics of SAFL and motivated the particular combination of features which comprise the language. Our methodology for the high-level synthesis of SAFL has also been presented with a particular emphasis on the (pre-compilation) program transformation phase. The following three chapters expand on this foundation by describing the technical details of our HLS tool for SAFL.

4

Soft Scheduling

We have seen that the SAFL language facilitates the description of hardware as a set of interconnected resources. These resources run concurrently and may be shared. The interaction between parallelism and resource-sharing leads to an obvious problem: how does one ensure that shared resources are accessed mutually exclusively?

Recall that existing silicon compilers solve the mutual exclusion problem by *statically* serialising operations during a compile-time scheduling phase (see Chapter 1). Whilst this approach is well-suited to fine-grained resources (e.g. arithmetic circuits) it does not scale to *system-level* resources (e.g. processors, IO controllers, busses). At the system-level, choices regarding the acquisition and use of resources are typically governed by *dynamic* considerations such as "the particular program that a processor is executing" or the actions of a user.

One alternative to static serialisation is to employ arbitration circuitry to perform scheduling dynamically. This technique is well-suited to system-level resources and is commonly used in practice to control access to shared busses[1]. Unfortunately, the use of dynamic arbitration is often inappropriate for fine-grained resources. The problem with attaching arbiters to fine-grained resources is that the time/area overhead of arbitration can become unacceptably large compared with the time/area overhead of the resource itself.

It seems therefore that if a scheduling technique is to be appropriate for *both* system-level *and* fine-grained resources a compromise must be reached between static serialisation and dynamic arbitration. Based on this observation we motivate a new approach to the scheduling problem in which (*i*) circuitry to perform scheduling *dynamically* is generated automatically; and (*ii*) *static* analysis is employed to detect cases where arbiters can be optimised away.

Our method is to scheduling as Soft Typing [32] is to type checking (see Figure 4.1): the aim is to retain the flexibility of dynamic scheduling whilst using static analysis to remove as many dynamic checks as possible. To highlight this analogy we choose to call our method *Soft Scheduling*.

[1] Note that even though this technique is commonly used in real hardware designs, arbiters are coded explicitly at the structural level. Existing high-level synthesis tools still only perform *static* scheduling.

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 51–64, 2004.

	Typing	Scheduling
Static	No dynamic checks required in object code. Not all valid programs pass type checker.	No scheduling logic required in final circuit. Not all valid programs can be scheduled statically.
Dynamic	Dynamic checking of argument types required each time a function is called. All valid programs can be run.	Scheduling logic required on each shared resource in the final circuit. All valid programs can be scheduled.
Soft	Fewer dynamic checks required (some removed statically). All valid programs can be run.	Less scheduling logic required (some removed statically). All valid programs can be scheduled.

Fig. 4.1. A Comparison Between Soft Scheduling and Soft Typing

In this chapter we describe Soft Scheduling and show how it is implemented as part of our SAFL silicon-compiler. We start by exploring related work (Section 4.1) after which we describe technical details: outlining how arbiters are generated to schedule access to shared resources (Section 4.2) and presenting a SAFL-based static analysis to remove redundant arbiters (Section 4.2.2). Three motivating examples are then presented demonstrating that (*i*) the removal of superfluous arbitration is critical in achieving efficient circuits (Section 4.3.1); (*ii*) soft scheduling is more expressive than static scheduling (Section 4.3.2); and (*iii*) dynamic scheduling can offer performance benefits over static serialisation (Section 4.3.3). We show how the Soft Scheduling framework allows conventional static scheduling algorithms to be represented as SAFL-level source-to-source transformations (Section 4.4).

Note that although, for expository purposes, this chapter describes Soft Scheduling in the framework of SAFL, the technique is applicable to any high-level HDL which allows function definitions to be treated as shared resources (e.g. HardwareC [86], Balsa [44], Tangram [138]). Indeed, in Chapter 7 we extend SAFL with π-calculus [100] style channels and assignment and demonstrate that the Soft Scheduling technique scales accordingly.

4.1 Motivation and Related Work

Traditional high-level synthesis packages perform scheduling using a data-structure called a *sequencing graph*—a partial ordering which makes dependencies between operations explicit. In this context, scheduling is performed by assigning a start time to each operation in the graph such that operations which invoke a shared resource do not occur in parallel [41] (see Section 1.2.1). There are a number of problems with this approach:

1. The time taken to execute each operation in the sequencing graph must be bounded statically (and in general padded to this length). This restriction means that conventional scheduling techniques are not expressive enough to handle a large class of practical designs. For example, it is impossible to statically schedule an operation to perform a bus transaction of unknown length.
2. Since operations are scheduled statically one must be pessimistic about what *may* be invoked in parallel in order to achieve safety. This can inhibit parallelism in the final design by unnecessarily serialising operations.

Ku and De Micheli have proposed Relative Scheduling [89] which extends the method outlined above to handle operations with statically unbounded computation times. Their technique partitions a flow-graph into statically-schedulable segments separated by *anchor nodes*—nodes which have unbounded execution delays. Each segment is scheduled separately at compile-time. Finally, the compiler connects segments together by generating logic to signal the completion of anchor nodes dynamically.

In [88] Ku and De Micheli show how Relative Scheduling of shared resources is integrated into their Olympus Hardware Synthesis System [43]. Their method permits the scheduling of operations whose execution time is not statically bounded, hence alleviating Problem 1 (above). However, potential contention for shared resources is still resolved by serialising operations at compile time so Problem 2 remains. Furthermore, there is still a class of practical designs which cannot be scheduled by Olympus:

Figure 4.2(i) shows a processor and a DMA (Direct Memory Access) controller both connected to a shared memory. In a high-level HDL, such as SAFL, we would essentially like to describe this system as three functions, `Processor(...)`, `DMA_Controller(...)` and `Memory(...)`, where `Processor` and `DMA_Controller` operate in parallel and both access the shared (external) function `Memory`. Note that since the `Processor` and `DMA_Controller` functions both access a shared resource, Olympus-style static scheduling requires that calls to these functions must be serialised. However, if neither the call to `Processor` nor the call to `DMA_Controller` terminate[2], attempting to sequentialise the operations is futile; the correct operation of the system relies on their parallel operation.

In contrast, Soft Scheduling is expressive enough to cope with such scenarios: an arbiter is automatically generated to ensure mutually exclusive access to the `Memory` whilst allowing the `Processor` and `DMA_Controller` to operate in parallel (see Figure 4.2.ii). The table of Figure 4.3 summarises the expressivity of various scheduling methods.

Of course, existing HDLs are capable of describing arbiters but they require arbitration to be coded explicitly at the *structural* level on an *ad hoc* basis. Since arbitration can impose an overhead both in terms of chip area and time,

[2] This is not merely a contrived example. In real designs both Processors and DMA Controllers are typically non-terminating processes which constantly update the machine state.

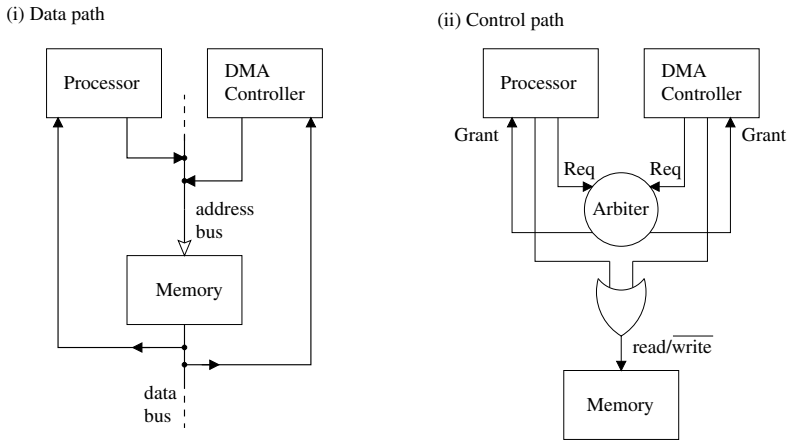


Fig. 4.2. A hardware design containing a memory device shared between a DMA controller and a processor

	Bounded execution delays	Unbounded execution delays	Non-terminating operations
Static	✓	✗	✗
Relative	✓	✓	✗
Soft	✓	✓	✓

Fig. 4.3. A table showing the expressivity of various scheduling methods

designers aim to eliminate unnecessary locking operations manually. For large designs this is a tedious and error-prone task which often results in badly structured and less reusable code. In contrast the Soft Scheduling approach analyses a *behavioural* specification, automatically inserting arbiters on a where-needed basis. This facilitates readable and maintainable source code without sacrificing efficiency.

Although we only consider the application of Soft Scheduling to hardware synthesis, the technique is also applicable to software compilation. Aldrich *et al.* [8] advocate a similar approach which uses static analysis to remove redundant synchronisation from Java programs.

4.1.1 Translating SAFL to Hardware

The full details of SAFL's translation to hardware are presented in the next chapter. However, in order to understand how arbiters are generated (the topic of this chapter), it is helpful to outline a little of the translation process at this stage. For the moment we assume that the generated hardware is synchronous; other design styles are considered in Chapter 5.

As in Relative Scheduling [89] control-signals are generated to signal the completion of operations explicitly. More precisely, each SAFL function definition, $f(\vec{x}) = e$, is compiled into a single resource, H_f, consisting of:

- logic to compute its body expression, e
- multiple control and data inputs: one control/data input-pair for each call site
- multiple control outputs (one to return control to each caller)
- a single data output (which is shared between all callers)

An example of function connectivity is given in Figure 4.4. In this example resource H_f is shared between H_g and H_h. Notice how H_f's data output is shared, but the control structure is duplicated on a per call basis.

To perform a call to resource H_f the caller places the argument values on its data input into H_f before triggering a call event on the corresponding control input. Some point later, when H_f has finished computing, the result of the call is placed on H_f's shared data-output and an event is generated on the corresponding control output.

4.2 Soft Scheduling: Technical Details

To protect shared resources the FLaSH compiler automatically generates scheduling logic to resolve conflicts dynamically (see Figure 4.4). The scheduling circuitry consists of two parts: (*i*) an arbiter to select which caller to service; and (*ii*) a locking mechanism to ensure the resource is accessed mutually exclusively. For the sake of brevity, we use the term *arbiter* to refer to both the arbiter and locking structure. Note that the insertion of arbiters is essentially

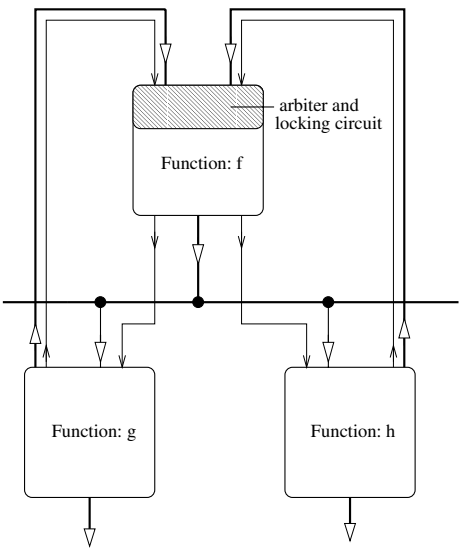


Fig. 4.4. A structural diagram of the hardware circuit corresponding to a shared function, f, called by functions g and h. Data buses are shown as thick lines, control wires as thin lines

the hardware equivalent of using binary semaphores to protect critical regions in multi-threaded software.

4.2.1 Removing Redundant Arbiters

Just because a resource is shared does not necessarily mean that arbitration is required. For example consider the following SAFL program:

```
fun f(x) = ...
fun g(x) = f(f(x))
```

In this case, the two calls to `f` cannot occur in parallel: the innermost call must complete before the outermost call can begin (recall that SAFL is a call-by-value language). We do not need to generate an arbiter to serialise the calls to $H_{\texttt{f}}$: from the structure of the program we can statically determine that the two calls will not try to access `f` simultaneously.

We use *Parallel Conflict Analysis* (see Section 4.2.2) in order to detect redundant arbiters. Removing unnecessary arbitration is important for two reasons:

1. Arbitration takes time: in the current version of the FLaSH compiler arbitration adds one cycle latency to a call even if the requested resource is available at the time of call. Although we may accept this latency if it is small in comparison to the callee's average execution time, consider the case where the callee is a frequently used resource with a small execution delay. In this case an arbiter may significantly degrade the performance of the whole system (see Example 4.3.1).
2. Arbitration uses chip area: although the gate-count of an arbiter is typically small compared to the resource as a whole, the extra wiring complexity required to represent request and grant signals adds to the area of the final design.

Arbiters are inserted at the granularity of calls. This offers increased performance over inserting arbiters on a per-resource basis. For example, in a design containing a function, f, shared between five callers, we may infer that only two calls to f require an arbiter—the other three calls need not suffer the overhead of arbitration.

4.2.2 Parallel Conflict Analysis (PCA)

Parallel Conflict Analysis (PCA) is performed over the structure of a whole SAFL program in order to determine which function calls may occur in parallel. If a group of calls to the same function may occur in parallel then we say that the group is *conflicting*. In the presentation of PCA it it is essential to differentiate between function definitions and function calls. In order to distinguish distinct calls we assume that each abstract-syntax node is labelled with a unique identifier, α, writing $f^{\alpha}(e_1, \ldots, e_k)$ to indicate a call to function f at abstract-syntax node α.

The result of PCA is a *conflict set*: a set of calls which require arbiters. For example, if the resulting conflict set is $\{f^1, f^2, f^5, g^{10}, g^{14}\}$ then we would synthesise two arbiters: one for the conflicting group $\{f^1, f^2, f^5\}$, the other for conflicting group $\{g^{10}, g^{14}\}$.

We now proceed to define PCA. Let e_f represent the body of function f. Let the predicate *RecursiveCall*(f^α) hold iff f^α is a recursive call (i.e. f^α occurs within the body of f). Figure 4.5 defines $\mathcal{C}[\![e]\!]$, the set of non-recursive calls which may occur as a result of evaluating expression e. $PC(\mathcal{S}_1, \ldots, \mathcal{S}_n)$ takes sets of

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &= \emptyset \\
\mathcal{C}[\![c]\!] &= \emptyset \\
\mathcal{C}[\![a(e_1, \ldots, e_k)]\!] &= \bigcup_{1 \le i \le k} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![f^\alpha(e_1, \ldots, e_k)]\!] &= \Big(\bigcup_{1 \le i \le k} \mathcal{C}[\![e_i]\!] \Big) \cup \begin{cases} \emptyset & \text{if } \mathit{RecursiveCall}(f^\alpha) \\ \{f^\alpha\} \cup \mathcal{C}[\![e_f]\!] & \text{otherwise} \end{cases} \\
\mathcal{C}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \bigcup_{1 \le i \le 3} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] &= \bigcup_{0 \le i \le k} \mathcal{C}[\![e_i]\!]
\end{aligned}
$$

Fig. 4.5. $\mathcal{C}[\![e]\!]$ is the set of non-recursive calls which may occur as a result of evaluating expression e

calls, $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$, and returns the conflict set resulting from the assumption that calls in each $\mathcal{S}_i$ are evaluated in parallel with calls in each $\mathcal{S}_j$ $(j \neq i)$:

$$PC(\mathcal{S}_1, \ldots, \mathcal{S}_n) = \bigcup_{i \neq j} \{f^\alpha \in \mathcal{S}_i \mid \exists \beta.\ f^\beta \in \mathcal{S}_j\}$$

The conflict set due to expression e, $\mathcal{A}[\![e]\!]$, is defined in Figure 4.6.

Finally, for a program, p, consisting of a sequence of user-function definitions:

$\texttt{fun } f_1(\ldots) = e_1; \ \ldots; \ \texttt{fun } f_n(\ldots) = e_n$

$\mathcal{A}[\![p]\!]$ returns the conflict set resulting from program, p. The letter $\mathcal{A}$ is used since $\mathcal{A}[\![p]\!]$ represents the calls which require arbiters:

$$\mathcal{A}[\![p]\!] = \bigcup_{1 \le k \le n} \mathcal{A}[\![e_k]\!]$$

Notice that the equation for $\mathcal{C}[\![f^\alpha(e_1, \ldots, e_k)]\!]$ is a little unusual in that it is not defined compositionally. This reflects the fact that PCA depends on the global structure of a whole SAFL program as opposed to just the local structure of a function definition. $\mathcal{C}[\![\cdot]\!]$ is well-defined due to the predicate *RecursiveCall* and the source restrictions on SAFL which ensure that the call-graph is acyclic (see Section 3.2.1).

$$
\begin{aligned}
\mathcal{A}[\![x]\!] &= \emptyset \\
\mathcal{A}[\![c]\!] &= \emptyset \\
\mathcal{A}[\![a(e_1,\ldots,e_k)]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\le i\le k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![f(e_1,\ldots,e_k)]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\le i\le k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \bigcup_{1\le i\le 3} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{0\le i\le k} \mathcal{A}[\![e_i]\!]
\end{aligned}
$$

Fig. 4.6. $\mathcal{A}[\![e]\!]$ returns the conflict set due to expression e

4.2.3 Integrating PCA into the FLaSH Compiler

After computing $\mathcal{A}[\![p]\!]$ at the abstract-syntax level the FLaSH Synthesis System translates p into an intermediate flow-graph representation which makes both control and data paths explicit. At this level, the `Call`-nodes (see Section 6) which require arbitration are tagged (i.e. we tag node, n, iff n represents `Call` f^α and $f^\alpha \in \mathcal{A}[\![p]\!]$).

When the circuit for H_f is generated only tagged calls to f are fed through an arbiter, other calls are merely multiplexed. If none of the calls to f are in $\mathcal{A}[\![p]\!]$ then H_f's arbiter is eliminated completely. As Section 4.3.1 shows, using Parallel Conflict Analysis to remove redundant arbitration can significantly improve the performance of a large class of designs.

4.3 Examples and Discussion

We provide three practical examples of applying Soft Scheduling to SAFL hardware designs. Each example illustrates a different point: Example 4.3.1 demonstrates that using static analysis to remove redundant arbiters is critical to achieving efficient circuits; Example 4.3.2 highlights the extra expressivity of Soft Scheduling over static scheduling techniques; Example 4.3.3 shows that dynamically controlling access to shared resources can lead to better performance than generating a single schedule statically.

4.3.1 Parallel FIR Filter

Finite Impulse Response (FIR) filters are commonly used in Digital Signal Processing where they are used to remove certain frequencies from a discrete-time sampled signal. Assuming the existence of (external) functions `read_next_value` and `write_value`, Figure 4.7 gives SAFL code corresponding to an integer arithmetic FIR filter.

```
fun mult1(x,y) = x*y
fun mult2(x,y) = x*y

fun FIR(x,y,z,w) =
   let val o1 = mult1(x,2)
       val o2 = mult2(y,3)
       val next = read_next_value()
    in
       let val o3 = mult1(x,7)
           val o4 = mult2(y,9)
       in write_value(o1 + o2 + o3 + o4);
          FIR(y,z,w,next)
       end
    end
```

Fig. 4.7. A SAFL description of a Finite Impulse Response (FIR) filter

Recall that the semantics of the `let` statement requires all `val`-declarations to be computed fully before the body is executed (see the SAFL semantics in Figure 3.1). Although this design contains two shared combinatorial multipliers, `mult1` and `mult2`, the outermost `let` statement ensures that the calls to the shared multipliers do not occur in parallel. As a result Parallel Conflict Analysis infers that no arbitration is required.

The shared combinatorial multipliers, `mult1` and `mult2` take a single cycle to compute their result. Generating an arbiter for a shared resource adds an extra cycle latency to each call (irrespective of whether the resource is busy at the time of call). Thus, in this case, if we naively generated arbiters for all shared resources, the performance of the design would be degraded by a factor of two.

This example illustrates the importance of using static analysis to remove redundant arbiters. For this design, using Parallel Conflict Analysis to remove unnecessary arbiters leads to a 50% speed increase over a policy which simply inserts arbiters on each shared resource.

4.3.2 Shared-Memory Multi-processor Architecture

Figure 4.8 contains SAFL code fragments describing a simple shared-memory multi-processor architecture. The system consists of two processors which have separate instruction memories but share a data memory. Such architectures are common in control-dominated embedded systems where multiple heterogenous processors perform separate tasks using a common memory to synchronise on shared data structures.

The example starts by defining the type of instructions (records containing 4-bit opcodes and 12-bit operands), declaring 2 constants and specifying the signatures of various (externally defined) memory functions. The function `Shared_memory` takes three arguments: `WriteSelect` indicates whether a read or a write is to be performed; `Address` specifies the memory location concerned;

```
type Instruction = {opcode:4,operand:12}
const WRITE=1, READ=0

extern Shared_memory(WriteSelect:1, Address:12, Data:16) : 16

extern instruction_mem1(Address:12) : 16
extern instruction_mem2(Address:12) : 16

(* Processor 1: Loads instructions from instruction_mem1 *)
fun proc1(PC:12, RX:16, RY:16, A:16) : unit =
  let val instr:Instruction = instruction_mem1(PC)
      val incremented_PC = PC + 2
   in
      case instr.opcode of
        1 => (* Load Accumulator From Register *)
             if instr.operand=1
               then proc1(incremented_PC,RX,RY,RX)
               else proc1(incremented_PC,RX,RY,RY)
      | 2 => (* Load Accumulator From Memory *)
             let val v = Shared_memory(READ, instr.operand, 0)
              in proc1(incremented_PC,RX,RY,v)
             end
      | 3 => (* Store Accumulator To Memory *)
             (Shared_memory(WRITE, instr.operand, A);
              proc1(incremented_PC,RX,RY,v))
       ... etc
  end

(* Processor 2: Loads instructions from instruction_mem2 *)
fun proc2(PC:12, RX:16, RY:16, A:16) : unit =
  let val instr:Instruction = instruction_mem2(PC)
      val incremented_PC = PC + 2
   in
      case instr.opcode of
       ....
      | 2 => (* Load Accumulator From Memory *)
             let val v = Shared_memory(READ, instr.operand, 0)
              in proc2(incremented_PC,RX,RY,v)
             end
      | 3 => (* Store Accumulator To Memory *)
             (Shared_memory(WRITE, instr.operand, A);
              proc2(incremented_PC,RX,RY,v))
       ... etc
  end

fun main() : unit = proc1(0,0,0,0) || proc2(0,0,0,0)
```

Fig. 4.8. Extracts from a SAFL program describing a shared-memory multi-processor architecture

`Data` gives the value to be written (this argument is ignored if a read operation is performed). It always returns the value of memory location `Address`.

Functions `proc1` and `proc2` define two simple 16-bit processors. Argument `PC` represents the program counter, `RX` and `RY` represent processor registers and `A` is the accumulator. The processor state is updated on recursive calls—neither processor terminates.

The `main` function initialises the system by calling `proc1` and `proc2` in parallel with `PC`, `RX`, `RY` and `A` initialised to 0.

Since the SAFL code contains parallel non-terminating calls to `proc1` and `proc2` both of which share a single resource, neither static nor relative scheduling are applicable (see Section 4.1): this example cannot be synthesised using conventional silicon compilers.

Soft Scheduling is expressive enough to deal with non-terminating resources: a circuit is synthesised which contains an arbiter protecting the shared memory whilst allowing `proc1` and `proc2` to operate in parallel.

4.3.3 Parallel Tasks Sharing Graphical Display

Consider a hardware design which can perform a number of tasks in parallel with each task having the facility to update a graphical display. Many real-life systems have this structure. For example in preparation for printing an ink-jet printer performs a number of tasks in parallel: feed paper, reset position of print head, check ink levels etc. Each one of these tasks can fail in which case an error code is printed on the graphical display.

```
extern display (data : 16) : unit

fun reset_head()  : unit =
          ...
            if head_status <> 0 then
                 display(4) (* Error code 4 *)
            else ...

fun feed_paper()  : unit = ... display(5) ...
fun check_ink()   : unit = ... display(6) ...

fun main():unit  =
        (feed_paper() || reset_head() || check_ink());
         do_print();
         wait_for_next_job(); main()
```

Fig. 4.9. The structure of a SAFL program consisting of several parallel tasks sharing a graphical display

A controller for such a printer in SAFL may have a structure similar to that shown in Figure 4.9. Let us assume that each of the tasks terminates in a

statically bounded time. Given this assumption, both static scheduling and Soft Scheduling can be used to ensure mutually exclusive access to `display`. It is interesting to compare and contrast the circuits resulting from the application of these different techniques.

Since the tasks invoke a common resource (`display`), applying static scheduling techniques results in the tasks being serialised. In contrast, Soft Scheduling allows the tasks to operate in parallel and automatically generates an arbiter which dynamically schedules access to the shared `display` function.

Errors occur infrequently and hence contention for the display is rare. Under this condition, and assuming that the tasks take all roughly the same amount of time, Soft Scheduling yields a printer whose initialisation time is three times faster than an equivalent statically scheduled printer. More generally, for a system of this form with n balanced tasks, Soft Scheduling generates designs which are n times faster.

4.4 Program Transformation for Scheduling and Binding

In this section we show how existing static scheduling and binding algorithms can be represented as SAFL-level transformations. By representing static scheduling/binding choices at the source level we gain a significant advantage over traditional "black-box" synthesis tools, since the engineer is able to "see what the tool has done".

We return to the example of Section 1.2.1 which involves statically scheduling the polynomial expression, $12x + x^2 + y^3$, using two multipliers and two adders. Let us assume that, as part of a larger specification, a designer has written a SAFL program to compute this polynomial expression:

```
fun poly(x,y) = 12*x + x*x + y*y*y
```

If we synthesise `poly` directly the resulting design will contain two adders and five multipliers. Imagine, however, that we instead we require a design which uses only two multipliers. We start to transform the program by adding two fresh `fun`-definitions corresponding to these two multiplier resources:

```
fun m1(x,y) = x*y
fun m2(x,y) = x*y
```

It is now clear that we can transform `poly` into a specification which computes the same function, but respects our required binding constraints (two multipliers). For example:

```
fun poly2(x,y) = m1(12,x) + m2(x,x) + m1(m1(y,y),y)
```

Function `poly2` requires arbiters to dynamically control access to `m1`. However, in this instance a static schedule would be more appropriate. Figure 4.10 shows how `poly2` can be further transformed into `poly3`, a function which implements the static schedule shown in Figure 1.9. (Note the use of the "let-barrier" syntactic sugar defined in Section 3.2.4. We are able to avoid the multiple nested

```
fun m1(x,y) = x*y
fun m2(x,y) = x*y

fun poly3(x,y) =
  let val t1 = m1(12,x)   val t2 = m2(x,x)
      ---
      val t3 = t1+t2      val t4 = m1(y,y)
      ---
      val t5 = m1(t4,y)
  in
     t5 + t3
  end
```

Fig. 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

`let` declarations by using the "`---`" construct.) When we apply soft scheduling to the specification of Figure 4.10 no arbiters will be generated since the analysis detects that neither shared resource will be subjected to multiple concurrent accesses.

4.5 Summary

Soft Scheduling is a powerful technique which provides a number of advantages over current scheduling technology:

More expressive: in contrast to existing scheduling methods, Soft Scheduling can handle arbitrary networks of shared inter-dependent resources.

Increased efficiency: in some circumstances, controlling access to shared resources dynamically yields significantly better performance than statically choosing a single schedule (see Example 4.3.3).

Higher level of abstraction: current hardware synthesis paradigms require a designer to code arbiters explicitly at the structural level. Soft Scheduling abstracts mutual exclusion concerns completely, increasing the readability of source code without sacrificing efficiency.

One of the aims of the FLaSH Synthesis System is to facilitate the use of source-level program transformation in order to investigate a range of possible designs arising from a single specification. The simplicity of our transformation system is partly due to the resource abstraction provided by Soft Scheduling—transformations involving shared resources would be much more complex if locking and arbitration details had to be considered at the SAFL-level. Furthermore Soft Scheduling allows us to use SAFL-level program transformation to represent static scheduling under resource constraints.

The program transformation given in Section 4.4 contrasts the local nature of existing static scheduling algorithms with the global nature of our Soft

Scheduling approach. Whereas static scheduling techniques (such as ASAP, List-scheduling etc.) can only be applied to a single expression within a SAFL function, Soft Scheduling operates over the structure of a whole program and is capable of scheduling calls to global resources originating from different functions.

Another advantage of Soft Scheduling over traditional static scheduling methodologies is that, since it does not statically partition operations into global time-steps, it is applicable to other design-styles than purely synchronous hardware. Not only can we design arbiters to cope with a number of different design styles (see Section 5.3.3), but Parallel Conflict Analysis itself is applicable regardless of the design-style targetted (since it makes no assumptions about the underlying target technology). We say that Parallel Conflict Analysis is *architecture-neutral* and explore the concept of architecture-neutral analyses further in Chapter 6.

When the parallel interleaving of non-terminating resources is required dynamic scheduling is essential (see Example 4.3.2); in other cases dynamic scheduling can offer increased performance (see Example 4.3.3). However, for fine-grained sharing of smaller resources whose execution delays are known at compile-time (such as arithmetic units), static scheduling techniques are more appropriate. Soft Scheduling provides a powerful framework which strikes a compromise between the two approaches. The designer has the flexibility either to describe a single static schedule (see Example 4.3.1) in which case dynamic arbitration is optimised away; or to leave scheduling details to the compiler (see Example 4.3.3) in which case dynamic arbitration is inserted where needed. Representing local scheduling decisions as semantics-preserving program transformations alleviates the black-box problem associated with other high-level synthesis methodologies (see Section 1.3.2).

5

High-Level Synthesis of SAFL

In order to justify our claim that "the SAFL language is suitable for hardware description and synthesis" a high-level synthesis tool for SAFL has been designed and implemented. We refer to our HLS tool for SAFL as *The FLaSH compiler*[1].

As can be seen in the block diagram of Figure 5.1, the FLaSH compiler consists of a single front-end and multiple back-ends. The tasks performed by the front-end are as follows:

- SAFL source is lexed and parsed into an abstract syntax tree. We check that the source complies with the restrictions described in Section 3.2.1; invalid SAFL is rejected.
- Simple type-checking is performed. This ensures that the bit-widths of function call arguments match those specified in the function's signature.
- We perform *parallel conflict analysis* (as described in Chapter 4).
- The abstract syntax tree is translated into intermediate code. Our intermediate representation is based on a control/data-flow graph model (see Section 5.1). At this level *architecture-neutral* analysis and optimisation is performed (see Chapter 6).

Each of the FLaSH compiler's separate back-ends targets a different design-style. We have implemented a back-end to generate synchronous hardware (Section 5.2) and propose another which targets GALS hardware (Section 5.3). To ensure the generation of efficient hardware, *architecture-specific* analyses and optimisations are sometimes performed in the back-end before the code generation phase (see Section 6.5). The compiler is designed in a modular fashion to allow new back-ends and additional analysis and optimisation phases to be incorporated with minimal effort.

In this chapter we describe the technical details of how the FLaSH Compiler translates SAFL programs into structural hardware descriptions. The interme-

[1] Initially, FLaSH, which stands for Functional Languages for Synthesising Hardware, was a term that we coined to refer to the whole research project. However, as time went on it gradually evolved into being the name of the compiler.

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 65–86, 2004.

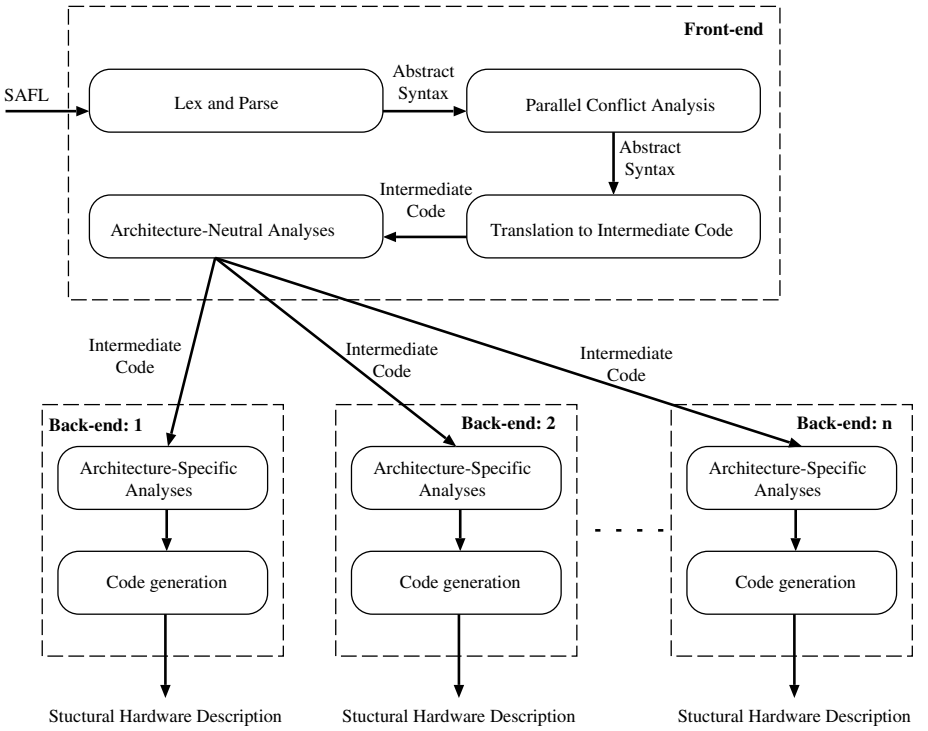


Fig. 5.1. Structure of the FLaSH Compiler

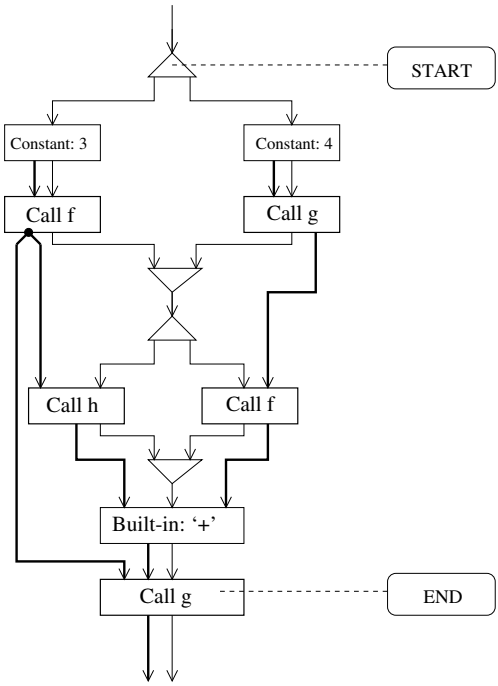
diate code format is presented in Section 5.1 and two back-end implementations are described in Sections 5.2 and 5.3.

5.1 FLaSH Intermediate Code

There are many analysis and optimisation techniques which are more suited to a lower level of representation than abstract syntax. For this reason the FLaSH compiler translates designs into intermediate code. The intermediate code is designed with the following aims:

- to map well onto hardware (many of the intermediate operations can be represented directly by simple circuit templates);
- to be *architecture neutral* (that is, not to favour the description of one particular design style—e.g. synchronous hardware);
- to make all control and data dependencies explicit; and
- to facilitate analysis and transformation.

As in many compilers, the intermediate representation is a graph-like structure best modelled as sets of nodes and edges.



This intermediate graph represents the following expression:

```
let var x = f(3)
    var y = g(4)
 in g(x, h(x) + f(y))
end
```

Fig. 5.2. Example intermediate graph

5.1.1 The Structure of Intermediate Graphs

An intermediate graph is a triple $(\mathcal{N}, E_c, E_d)$ where:

$\mathcal{N}$ is a set of nodes

$E_c \subseteq \mathcal{N} \times \mathcal{N}$ is a set of *control edges*

i.e. $(n, n') \in E_c \Leftrightarrow$ control flows out of n into n'

$E_d \subseteq \mathcal{N} \times \mathcal{N}$ is a set of *data edges*

i.e. $(n, n') \in E_d \Leftrightarrow$ data flows out of n into n'

Intermediate graphs are best viewed pictorially. We adopt the convention that thin lines represent control edges and thick lines represent data edges. Figure 5.2 gives an example of an intermediate graph and the SAFL expression it represents. The types of node used in intermediate graphs are summarised in Figure 5.3. Given a node n, we define the formula $(n : \textsc{call}\ f)$ to hold iff n is a call node (and similarly for other node forms).

As can be seen from Figure 5.3, nodes have at most one data output-port. We say that a node n is a *data-producer* if it has a data output-port. If n is a data-producer then we define $n.DataOut$ to refer to n's (single) data output-port.

When compiling an expression to an intermediate graph, we mark two distinguished intermediate nodes: *start* and *end*. The node marked *start* represents an expression's entry point and the node marked *end* is a data-producer whose value will ultimately yield the result of the expression.

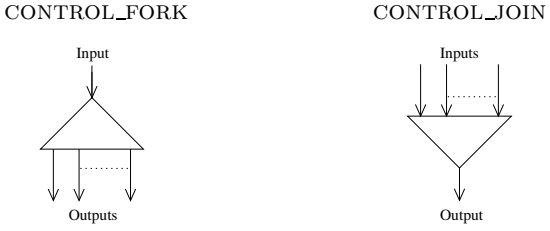
The meanings of nodes (and their pictorial representations) are outlined informally in the following sections. In order to describe the semantics of nodes we often talk of *control* and the act of *propagating control*. This will become clearer when we outline the translation to hardware, but until then it may help the reader to imagine control edges propagating events cf. asynchronous circuit design.

Node Type	*Number of Control*		*Number of Data*	
	Inputs	*Outputs*	*Inputs*	*Outputs*
CONTROL_FORK	1	≥ 2	0	0
CONTROL_JOIN	≥ 2	1	0	0
CONSTANT	1	1	0	1
BUILT_IN	1	1	≥ 1	1
CALL	1	1	≥ 0	1
JUMP	1	1	≥ 0	1
READ_FORMAL	1	1	0	1
CONDITIONAL_SPLIT	1	2	1	0
CONDITIONAL_JOIN	2	1	3	1

Fig. 5.3. Nodes used in intermediate graphs

Fork/Join Parallelism

Parallelism is made explicit through control fork/join nodes:

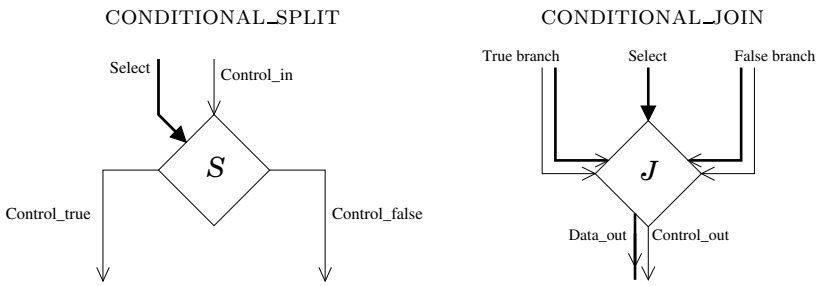


- When control reaches a CONTROL_FORK node's single input then control is propagated to its many control outputs.
- Conversely, a CONTROL_JOIN node waits for control to arrive at all its inputs, before propagating control to its single output.

Examples of the use of CONTROL_FORK and CONTROL_JOIN can be seen in Figure 5.2 where they are used to facilitate the parallel evaluation of `let`-declarations and function arguments.

Conditionals

We represent conditional execution using two nodes: CONDITIONAL_SPLIT and CONDITIONAL_JOIN. The style of these nodes may seem unusual to people familiar with software compilers, but they map well onto hardware. In particular the CONDITIONAL_JOIN node mimics a multiplexer.

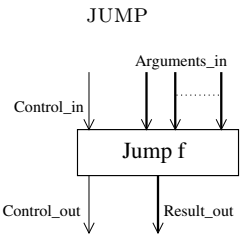


- A CONDITIONAL_SPLIT node channels control from Control_in to either Control_true or Control_false output depending on the boolean data value on the Select input.
- The CONDITIONAL_JOIN node has two control/data input pairs corresponding to the true and false branches of a corresponding CONDITIONAL_SPLIT. When control arrives at either control input it is propagated to Control_out. Similarly, data on one of the two data-inputs is propagated to Data_out. The boolean value on Select is used to determine which of the data values to forward.

Figure 5.4 shows how conditionals are translated into intermediate structures.

Recursive Calls

We use the JUMP node to represent recursive calls. (Recall that we can always implement recursive calls as jumps because all recursive calls are restricted to tail-calls.)



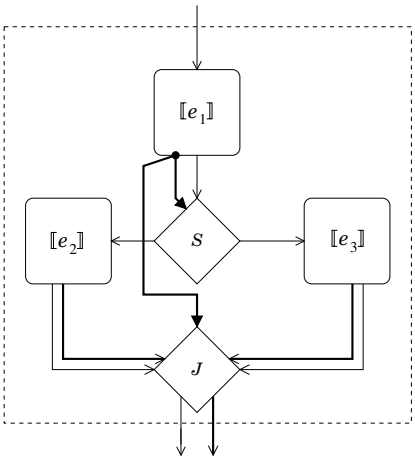
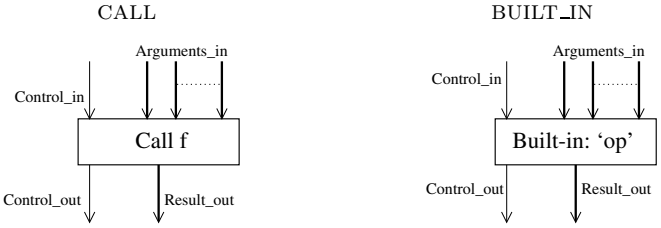


Fig. 5.4. Translation of conditional expression: `if` e_1 `then` e_2 `else` e_3

A JUMP returns no data and never propagates control—hence its outputs are ignored; we can treat the data output as some random undefined value and the control output as never asserted. In practice, its control and data outputs are not realised at the hardware level. (We only include them at the intermediate level because it allows us to maintain the invariant that a closed expression is represented by an intermediate graph with a control input, a control output and a data output.)

Primitive and User-Defined Function Calls

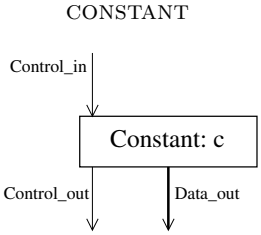
Although they are treated very differently at the hardware level, primitive function calls and user-defined function calls look similar at the intermediate level:



Both these nodes read their data-inputs when control reaches their single control-input, perform their operation and then return their result on the single data-output, propagating control forwards when the operation is complete. (Note that the CALL node is only used to represent non-recursive calls to external functional-units. We have already shown how we use JUMP to represent recursive calls.)

Constants

The CONSTANT node simply propagates control whilst continually writing its value, c, onto its data-output.

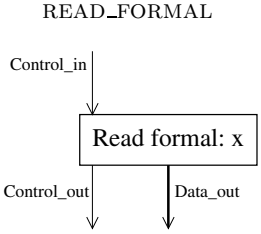


Variables

Variables can be subdivided into two separate categories:

- `let`-bound variables
- formal parameters

Although `let`-bound variables are represented implicitly by sharing a node's data-output (see Figure 5.2) we require a special node to deal with formal parameters:



This node propagates control whilst writing the value of the local function's formal parameter, x, to its data-output. Figure 5.5 shows an example of the READ_FORMAL node by translating the body of `fun f(x) = x+3`.

5.1.2 Translation to Intermediate Code

The main translation phase is implemented by a recursive function which walks the abstract-syntax tree, constructing parts of intermediate graph and then gluing them together. There are a number of issues worth mentioning explicitly:

1. Intermediate graphs represent expressions rather than programs. Thus, in order to compile a SAFL program into intermediate form we compile each of the function bodies separately and maintain a list of (function name, intermediate expression) pairs. The *start* and *end* nodes of the intermediate expression then correspond to the entry and exit points of the function.

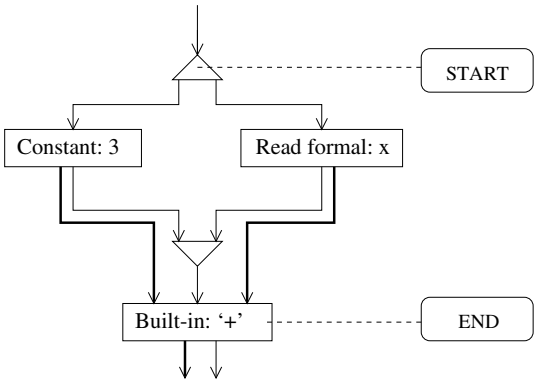


Fig. 5.5. Intermediate graph representing the body of `fun f(x) = x+3`

2. We translate abstract syntax expressions into intermediate graphs where all `let`-declarations, primitive function call arguments and user-defined function call arguments are evaluated in parallel. (This parallelism is made explicit through the use of CONTROL_SPLIT and CONTROL_JOIN nodes.)
3. We perform a simple reachability analysis and dead-code elimination phase to remove redundant nodes from the graph before any analysis takes place. For example consider the intermediate graph corresponding to the body of:

```
fun f(x) = if ... then f(x+1)
                  else f(x-1)
```

   In this case, since both of the conditional branches contain tail recursive calls (represented as JUMP nodes) we know that the corresponding CONDITIONAL_JOIN node will never be reached and is hence unnecessary.

It is useful to briefly compare our intermediate format with other program graphs which have been developed previously [129]. The structure of FLaSH intermediate graphs is similar to that of Program Dependence Graphs (PDGs) [47] in the sense that both control- and data-dependencies are represented explicitly. Like the PDG, we essentially combine the Control Flow Graph (CFG) and the Data Dependence Graph (DDG) [5] in a single unified framework. Our CONTROL_JOIN node essentially performs the same function as γ-nodes in the Gated Single-Assignment (GSA) framework [66].

The explicit representation of parallelism through CONTROL_FORK and CONTROL_JOIN nodes has some similarities with the Parallel Program Graph (PPG) [128]. The CONTROL_FORK node, used to explicitly represent parallelism performs the same function as the PPG's *MGOTO* node. However, unlike the PPG, we do not provide facility for arbitrary synchronisation between threads. The only way threads can synchronise in our model is via a CONTROL_JOIN node.

5.2 Translation to Synchronous Hardware

After SAFL programs have been translated into intermediate code, various analyses and optimisations are performed. The details of intermediate-code-level analyses are found in Chapter 6. Here, we continue by describing the internals of the FLaSH compiler's synchronous back-end focusing on how intermediate graphs are mapped onto circuit templates. Our compiler targets hierarchical RTL-Verilog; we use existing RTL-compilers to map the generated Verilog to hardware (see Chapter 10). In this way we avoid having to consider low-level issues which are not directly relevant to this monograph (e.g. place-and-route and logic minimisation).

The synchronous hardware generated is based on a matched-delay protocol where each group of data-wires is bundled with a control wire. Control wires propagate events which, in this framework, are represented as one-cycle pulses. The circuits are designed so that control events propagate at the same speed as valid data. When the control wire is high the corresponding data wires are guaranteed to be valid. Firstly we discuss how expressions are compiled; Section 5.2.2 explains how function definitions are compiled into hardware functional-units.

5.2.1 Compiling Expressions

An expression is compiled into a hardware-block with one control input, one control output and one data output—see Figure 5.6 (left). Signalling an event on the control input triggers the expression which computes its value and places its result on the data output, signalling an event on its control output when the data output becomes valid. Our method of translating expressions to synchronous hardware has some similarities with Page's scheme for the high-level synthesis of Occam [113]. The major difference is that since Page adopts the convention that every expression will be computed in a single cycle his expressions do not need explicit control signals. We take a more general view: SAFL expressions can take an arbitrary number of cycles. (Of course, if we can statically determine the number of cycles in which a particular expression will execute, then we can optimise the control signals away—see Section 6.5.)

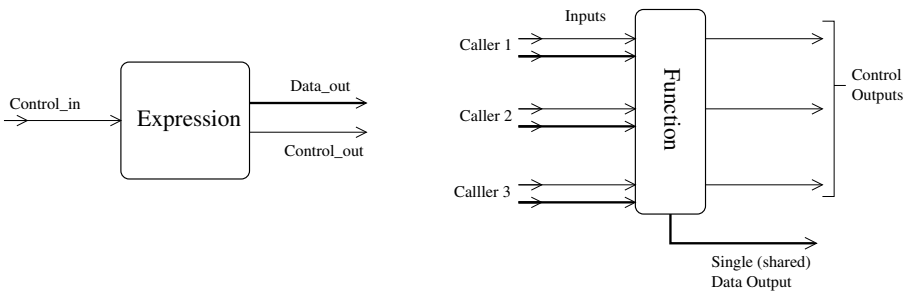


Fig. 5.6. Expressions and Functions

Our compiler maintains the following invariant: once an expression, e, has been computed (i.e. its control output has been asserted), $e.DataOut$, remains valid indefinitely as long as all its data inputs remain valid. One point worth considering further here is our decision to give the CONDITIONAL_JOIN node an explicit select input (see Figure 5.4). After all, in some sense the select input is redundant: why not just let the CONDITIONAL_JOIN select which data input to propagate based solely on which of the two control inputs is asserted?

Our motivation for the explicit select input is that it provides an efficient way of ensuring that conditional expressions maintain the aforementioned invariant. If, at the hardware level, we used the control inputs as select lines into the multiplexer used to implement a CONDITIONAL_JOIN then the node's data output would only be valid for a single cycle (since the active control input would only be asserted for a single cycle). Whilst we could alleviate this problem by explicitly latching the control signals, the need for extra latches is eliminated if we simply use the data output of the conditional test expression as the select input for the CONDITIONAL_JOIN multiplexer.

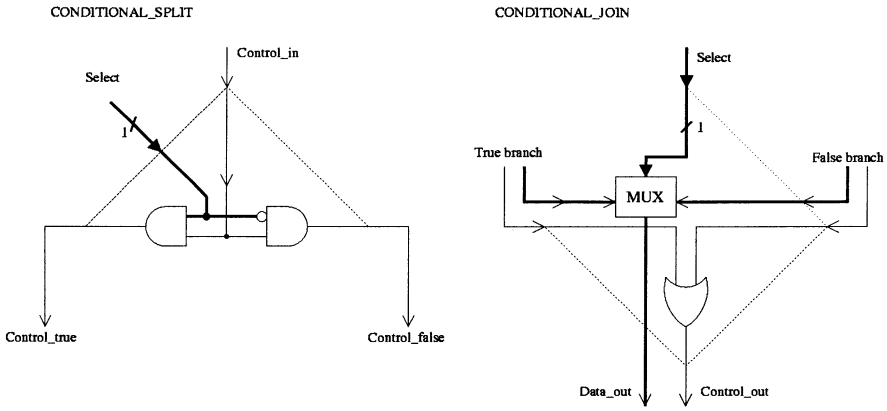


Fig. 5.7. Hardware blocks corresponding to CONDITIONAL_SPLIT (left) and CONDITIONAL_JOIN (right) nodes

CALL nodes are synthesised into connections to other hardware level functional-units; JUMP nodes (representing recursive calls) are synthesised into a connection back into the current functional unit; all other intermediate nodes are translated into a corresponding circuit template. Figure 5.7 shows the circuit templates corresponding to conditional split and join nodes; Figure 5.8 shows the hardware block corresponding to a CONTROL_JOIN node. Some of our schematics use synchronous reset-dominant SR flip-flops. Figure 5.9 shows how these can be constructed from the more familiar D-type flip-flop.

Other nodes have their obvious translations. For example a CONTROL_SPLIT node is just a wire connecting its control input to all its control outputs, and a BUILT_IN: `<op>` node contains combinatorial logic to perform operation `<op>`.

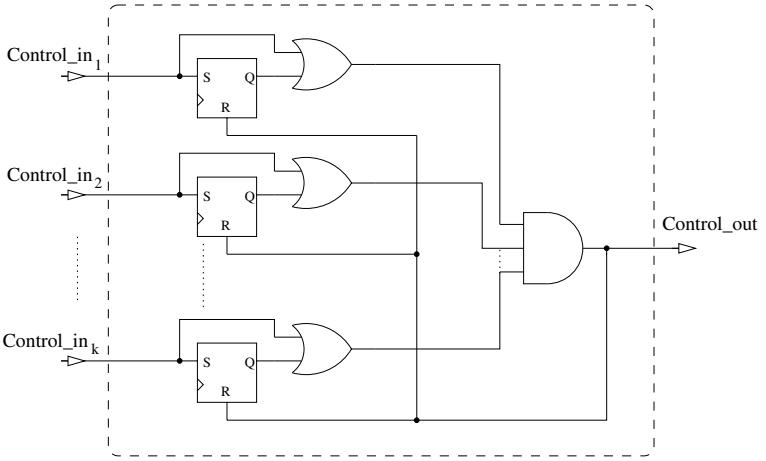


Fig. 5.8. Hardware block corresponding to a CONTROL_JOIN node

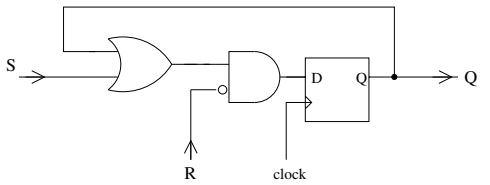


Fig. 5.9. How to build a synchronous reset-dominant SR flip-flop from a D-type flip-flop

5.2.2 Compiling Functions

A function *definition* is compiled into a single hardware-block (functional-unit) with multiple control and data inputs: one control/data input-pair for each call—see Figure 5.6 (right). There are multiple control outputs (one to return control to each caller), but only a single data output (which is shared between all callers). Each function contains logic to compute its body expression.

Calling Protocol

We have already seen how functional-units are composed to form larger structures in the previous chapter. Recall that Figure 4.4 showed a functional-unit H_f shared between resources H_g and H_h and described the calling protocol. The important point to remember is that although H_f's data output is shared, the control structure is duplicated on a per call basis.

There is another important point which is worth mentioning here. Consider the following SAFL expression which involves computing `f(4)`, then computing `f(5)` (once the first call has terminated) and finally computing the sum of the two results:

```
let var x = f(4)
 in let var y = f(5)
     in x + y
    end
end
```

Since `f` represents a shared resource, H_f, with a single output, the value of `f(4)` must be latched into a temporary register (since the subsequent call `f(5)` will change the value on H_f's shared output). We call these temporary registers *permanising registers* since they make the result of computing an expression permanent, decoupling the caller from the callee. In Chapter 6 we present an analysis which allows us to eliminate redundant permanising registers. For now we assume a naïve translation scheme which simply inserts a permanising register on the output of each call to a shared resource.

Function Unit Internals

The internals of a functional-unit are shown in Figure 5.10. First consider the control path. Each control/data input-pair to a functional-unit, H_f, corresponds to a single call. If any of the control inputs may trigger calls in parallel (as inferred by the soft scheduling analysis presented in Chapter 4) then these control wires are passed through an arbiter which ensures that only one call is dealt with at a time.

Having been through the arbiter, the control inputs are fed into the *External Call Control Unit* (ECCU—see Figure 5.11), which:

1. remembers which of the control inputs triggered the call;
2. invokes the function body expression (by generating an event on the `FB_invoke` wire);
3. waits for completion (signalled by the `FB_finished` wire); and finally
4. generates an event on the corresponding control output, signalling to the caller that the result is waiting on H_f's shared data output.

Now let us consider the data-path. The data inputs are fed into a multiplexer which uses the corresponding control inputs as select lines. The selected data input is latched into the argument registers. (Obviously, the *splitter* is somewhat arbitrary; it is included in the diagram to emphasise that multiple arguments are all placed on the single data-input.) Note that recursive calls feed back into the multiplexer to create loops, re-triggering the body expression as they do so. The D-type flip-flop is used to insert a 1-cycle delay onto the control path to match the 1-cycle delay of latching the arguments into the registers on the data-path.

The function body expression contains connections to the other functional-units that it calls. These connections are the ones marked "calls to other functions" in Figure 5.10 and are seen in context in Figure 4.4.

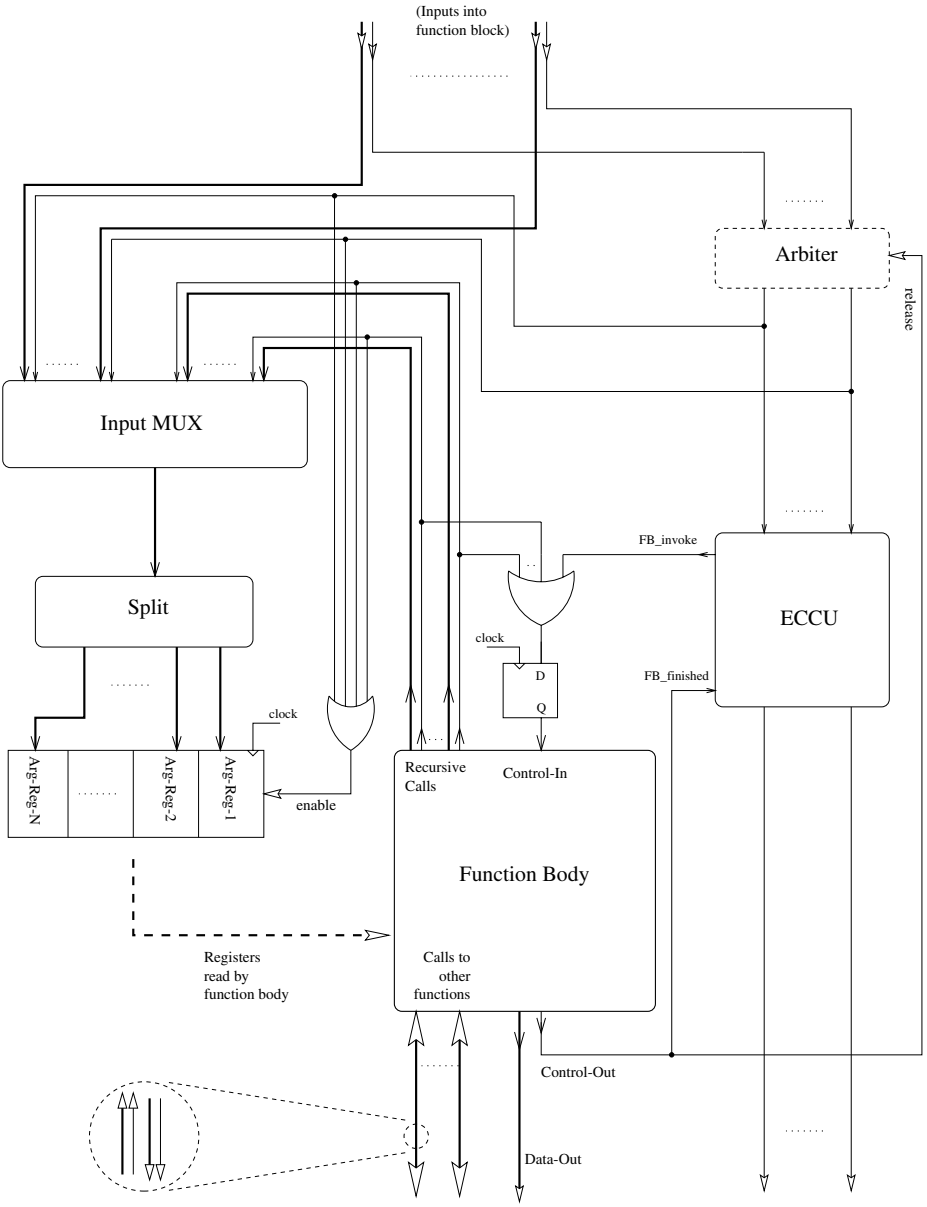


Fig. 5.10. A Block Diagram of a Hardware Functional-Unit

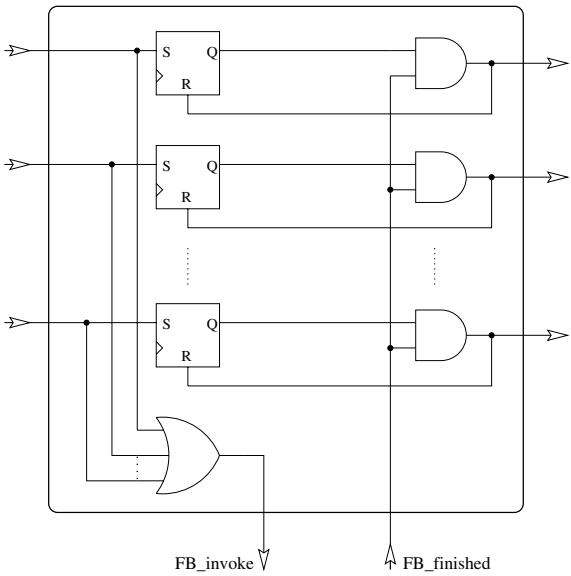


Fig. 5.11. The Design of the External Call Control Unit (ECCU)

Arbiter Design

The SAFL language semantics do not dictate a specific arbitration policy. As a result there are a number of different arbitration policies that could be implemented by the compiler (e.g. round-robin, static fixed priority etc.). In the FLaSH compiler we chose to implement static fixed priority arbitration in the synchronous back-end. Our motivation behind this choice is primarily because it yields the simplest circuits—in particular the arbiter does not require state that persists between separate calls[2].

Figure 5.12 shows the circuit corresponding to a fixed-priority arbiter with 3 control-inputs. Once latched the incoming control pulses are fed into a priority encoder which selects one of them. Figure 5.13 shows the simple combinatorial circuit we use as a priority encoder. The flip-flop whose data output is marked "locked" stores whether or not the function resource is currently busy. By forming the conjunction of the locked signal and the outputs of the priority encoders we ensure that (*i*) at most one call can be active at a time; and (*ii*) a call can only proceed when the function is not locked. As control leaves the arbiter the "lock flip-flop" is set, ensuring that further calls cannot proceed until "release" is signalled.

[2] The reader may be worried about the lack of *fairness* with this arbitration policy. We concede that this may prove problematical where state and IO are concerned (see Chapter 7). A topic for future work is to investigate other dynamic scheduling policies and assess the efficiency of their hardware implementation.

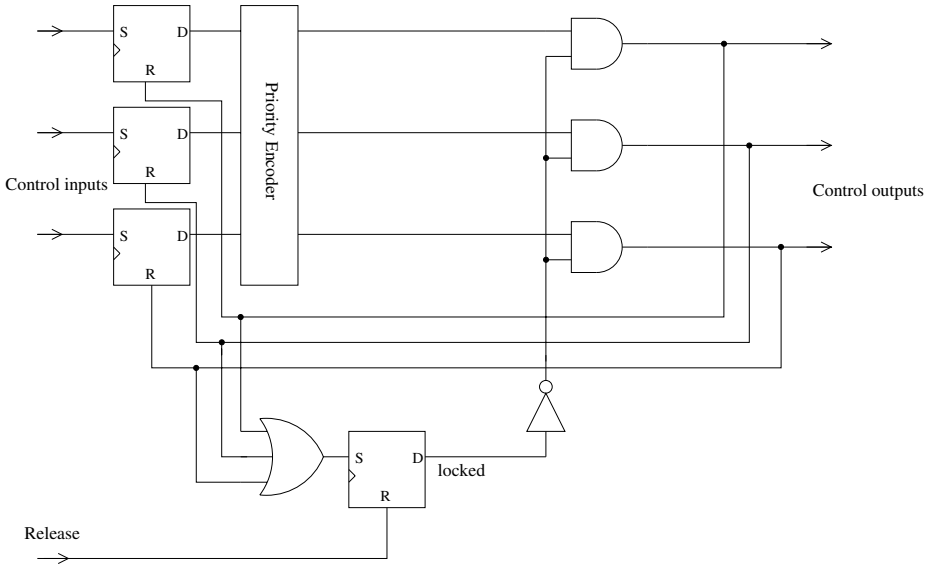


Fig. 5.12. The Design of a Fixed-Priority Synchronous Arbiter

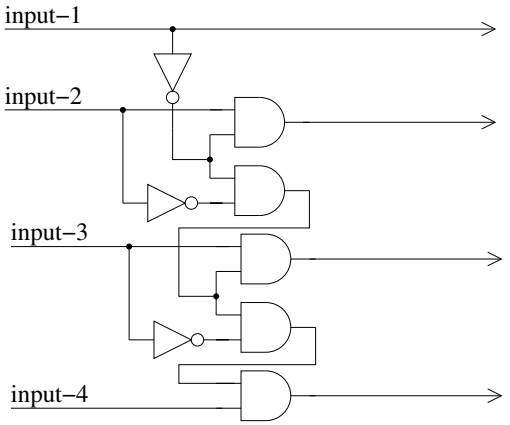


Fig. 5.13. The Design of a Combinatorial Priority Encoder with 4 inputs. (Smaller input numbers have higher priorities)

5.2.3 Generated Verilog

The FLaSH compiler produces RTL Verilog to express the generated hardware. The Verilog is hierarchical in the sense that each *function* definition in the SAFL source has a corresponding *module* definition in the generated Verilog. This is useful for two reasons:

– it makes the Verilog more readable, since the high-level circuit structure is explicit; and

– a hierarchical design is useful when performing place-and-route/logic synthesis, since typical RTL compilers operate more efficiently when provided with modular designs. Furthermore, typical logic synthesisers often allow users to specify optimisation priorities (e.g. area vs. time) on a per-module basis.

5.2.4 Compiling External Functions

The SAFL language allows functions to be declared as `extern`. Such functions are called *external functions* since their bodies are written in some other language (e.g. Verilog). At the SAFL level only signatures are provided for `extern` functions. However, although the function bodies are omitted the FLaSH compiler still generates logic which deals with the call/return mechanism.

For an external function, all the hardware shown in Figure 5.10 is generated by the compiler except for the function body itself. The details are best illustrated with an example. Assume that a SAFL specification contains the following declaration:

```
extern memory(address:16, value:32, write:1):32
```

As with normal (i.e. non-`extern`) functions the FLaSH compiler generates a Verilog module definition, $H_{\texttt{memory}}$, corresponding to the SAFL function `memory`. As well as all the logic to deal with the function call mechanism, arbitration etc. $H_{\texttt{memory}}$ contains a Verilog `module` instantiation of the following form:

```
extern_memory memory_body(clock, d_out, c_out, c_in,
                          address, value, write);
```

Compiling the generated Verilog directly using a standard RTL compiler will lead to an error, since module `extern_memory` is not defined. It is up to the designer to supply a module with this name and interface before compiling the FLaSH-generated code. The technical details of the interfacing mechanism required to implement the `extern_memory` module are given below:

1. Receiving a single cycle pulse on input `c_in` signifies that computation can begin.
2. The data-inputs `address`, `value` and `write` are used to transmit the argument values of the incoming call. From the moment `c_in` goes high, these data-inputs contain valid data.
3. Once computation is complete, a single-cycle pulse must be generated on output `c_out`.
4. The result of the function must be placed on data-output `d_out` and be valid from the point that `c_out` goes high.

Note that although the `extern_memory` module must be provided in Verilog form, this does not mean that one can only integrate Verilog with SAFL code. Indeed one is able to implement external functions in any synthesis system which can generate synthesisable Verilog (e.g. Lava, HandelC—see Chapter 2). When implementing SAFL on FPGAs, we commonly map `extern` functions onto vendor-supplied IP blocks such as dual-ported RAMs, ROMs etc. (see Chapter 10).

5.3 Translation to GALS Hardware

Globally Asynchronous Locally Synchronous (GALS) designs consist of a number of separately clocked synchronous subsystems connected via an asynchronous communication architecture. The GALS methodology is attractive as it offers a potential compromise between (*i*) the difficulty of distributing a fast clock in large synchronous systems; and (*ii*) the seeming area-time overhead of fully-asynchronous circuits. Another compelling argument in favour of GALS is that one is able to use existing industrial hardware design tools to synthesise and simulate the synchronous subsystems. It is only for the asynchronous interconnect that special techniques and tools are required.

We extend the SAFL syntax with a new primitive `ClockDomain` which allows `fun` declarations to be grouped together in the following manner:

```
ClockDomain my_domain1 {
  fun f1() ...
  fun f2() ...
}

ClockDomain my_domain2 {
  fun f3() ...
  fun f4() ...
}
```

The FLaSH compiler generates a separate clock domain for each `ClockDomain` block. The `ClockDomain` primitive is followed by the *name* of the defined clock domain. For each block, `ClockDomain` ⟨*name*⟩, a new clock signal is generated with name `clk_`⟨*name*⟩. In the example above two clock signals (`clk_my_domain1` and `clk_my_domain2`) would appear in the Verilog output. It is up to the designer to connect these signals to appropriate clock generation circuitry at the RTL-level.

Functions in separate `ClockDomain`s are free to call each other. Hardware to interface separate clock-domains is automatically inserted at domain boundaries. Note that `ClockDomain`s cannot be nested.

5.3.1 A Brief Discussion of Metastability

Before launching into the technical details of the inter-clock-domain interfacing circuitry it is first necessary to consider the problem of *metastability* [143].

A *synchronisation failure* may occur if the data and clock inputs of a flip-flop do not satisfy the necessary setup- and hold-time constraints. More specifically, a synchronisation failure occurs when, at the point of latching the data, the voltage on the latch's input is close to the threshold voltage of the inverters in the latching circuit. When this happens, the latch may enter a *metastable* state: an unstable equilibrium between making a decision to resolve to a logic-1 or a logic-0. Although in practice slight imbalances will eventually force the

latch to resolve one way or the other, the time for the metastability to resolve is unbounded.

Analysis of the metastability problem [56] tells us that the probability, P, of a latch remaining in a metastable state (i.e. not resolving to a valid logic-1 or logic-0) for time t can be approximated by:

$$P = ke^{-t/\tau}$$

where constants k and τ are determined by the physical properties of a given target technology (e.g. CMOS or TTL). Thus we can reduce the probability of metastability to an arbitrarily small level[3] by waiting for a predetermined time before reading the value of a potentially metastable register. If one is transferring data between two clock domains which operate asynchronously with respect to each other then one must consider metastability issues carefully. The problem is that the data in one clock domain may be changing just as the second clock domain is trying to latch it.

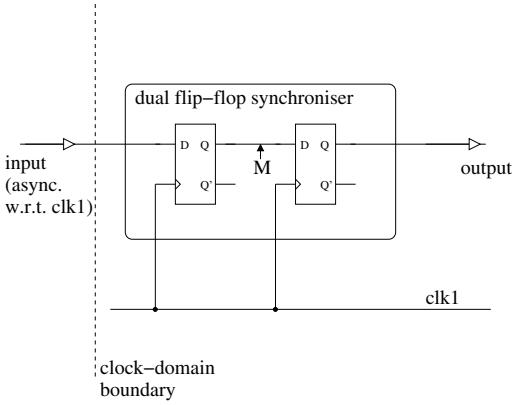


Fig. 5.14. A dual flip-flop synchroniser. Potential metastability occurs at the point marked "M". However, the probability of the synchroniser's output being in a metastable state is significantly reduced since any metastability is given a whole clock cycle to resolve

A number of researchers have investigated techniques for dealing with metastability. Common techniques include:

- designing circuitry to delay the clock if a latch input violates the setup time [102]; and
- latching asynchronous signals multiple times (sequentially), thus allowing more time for metastability to resolve before the signal is read [143].

[3] The precise definition of a "satisfactory level" depends entirely on the application domain in which the circuit is employed. For example, in applications where failure results in a machine having to be rebooted then a Mean Time Between Failure (MTBF) of once a month may be tolerable; however, if failure results in widespread death and destruction than a longer MTBF may be more appropriate.

The pros and cons of each approach and their technical details is not the subject of our research. Indeed when compiling SAFL to GALS circuits we are free to choose any of these implementation techniques. For expository purposes this chapter adopts the latter method, employing the simple technique of double latching asynchronous signals. The circuit we use is known as a *dual flip-flop synchroniser* and is shown in Figure 5.14. The first flip-flop latches the data and potentially enters a metastable state. However, a whole clock-cycle goes by before the data is read (by the second latch). Thus, by the time the data enters the second latch sufficient time has elapsed to make the probability of metastability sufficiently small for our purposes. (If required, more registers may be cascaded to decrease the probability of metastability—and hence the Mean Time Between Failure—at a cost of increased latency through the synchroniser.)

5.3.2 Interfacing between Different Clock Domains

There are two separate elements involved in dealing with inter-clock-domain function calls. Firstly there is the design of the control circuitry to interface the two clock domains (described below); secondly we must modify our arbiter design slightly (see Section 5.3.3).

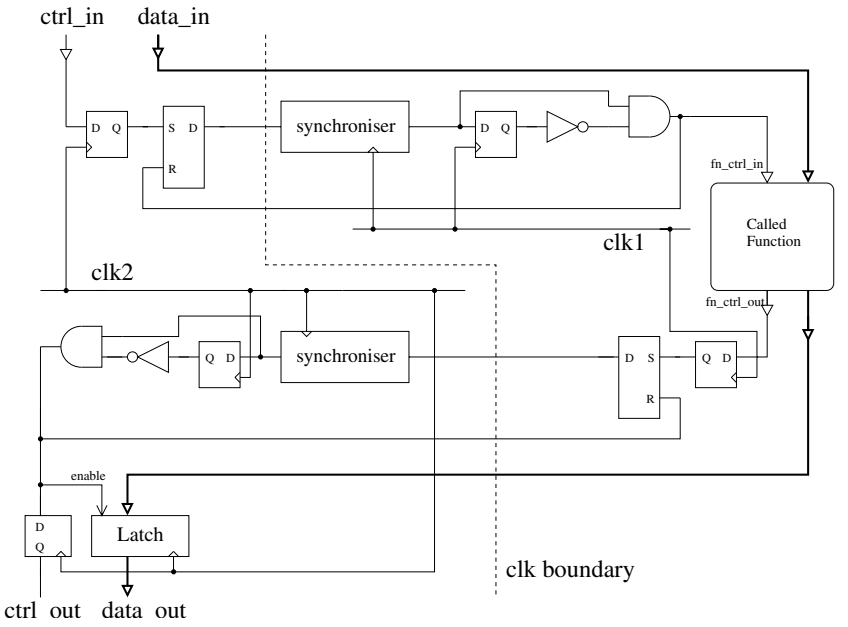


Fig. 5.15. An inter-clock-domain function call

Figure 5.15 shows the control circuitry that we use to deal with calls between separate clock domains. The diagram shows an inter-clock-domain call where the callee is clocked by `clk1` and the caller is clocked by `clk2`.

As with the synchronous translation described in Section 5.2 a one-cycle pulse arrives at the call's control input, `ctrl_in`. The first thing we do is to feed this control pulse into a D-Type flip-flop to delay the pulse by one cycle (in the caller's clock domain). The reason for this one cycle delay is to allow the function call's argument values on the corresponding data-path to stabilise.

The control pulse is then fed into the set-input of an *asynchronous* RS-latch. A small delay after the positive edge of the control pulse arrives on the set-input the latch's D-output goes high (and remains high until the R input is asserted). Thus the one-cycle pulse has now been converted into a signal which will remain high until the callee has sampled it. This signal is fed into a synchroniser which samples it in the callee's clock domain. Since the signal remains high indefinitely we know that it will eventually be sampled regardless of the relative speeds of the two domains. We use a dual flip-flop synchroniser to latch the signal into a new clock domain (see Figure 5.14)[4].

On leaving the synchroniser the signal is fed into a small piece of circuitry (consisting of a flip-flop, an inverter and an AND-gate) which simply converts the constant logic-1 value back into a single-cycle pulse (in the `clk1`-domain)[5]. We also reset the RS-latch since, by this stage, we know that its output has been successfully sampled. The resulting one-cycle pulse in the `clk1`-domain can now be fed directly into the called function's arbitration/ECCU circuitry in the usual way (see Figures 5.11 and 5.12). Note that the data-inputs to the function can be read directly from the `clk2`-domain since we know for sure that their values are stable and will remain so.

On completing execution a one-cycle pulse (in the `clk1`-domain) is emitted on the function's corresponding control output, `fn_control_out`. As before we delay the pulse by a single cycle to ensure that the function's data output has stabilised. This pulse is fed into a circuit (symmetrical to the one just described) which converts it into a one-cycle pulse in the `clk2`-domain.

The circuit of Figure 5.15 makes a number of assumptions about the relative speeds of the two clock domains:

- the output of the synchroniser must remain high for at least 1 cycle (in its own clock domain) after the RS-latch (in the other clock domain) is reset. Note that this is indeed the case for the dual flip-flop synchroniser shown in Figure 5.14;
- the synchronisers must have time to reset to their initial state (i.e. output = logic-0) before their inputs are reasserted.

One of the benefits of the synchronisation scheme just described is that, since it uses standard components, it can be implemented easily on an FPGA. We can

[4] We assume that the clock speeds and target technology are such that a single cycle delay is sufficient for the probability of meta-stability to be negligible. However, should this not be the case for a particular design more flip-flops can be added to the synchroniser in the usual way to allow the meta-stability longer to resolve.

[5] Note that the circuit to convert the constant logic-1 back into a single-cycle pulse does *not* add an extra cycle latency.

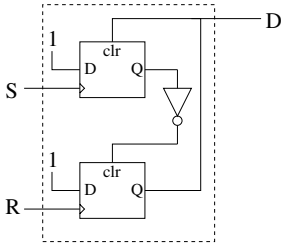


Fig. 5.16. Building an asynchronous RS latch out of two D-Type flip-flops with asynchronous resets (`clr`)

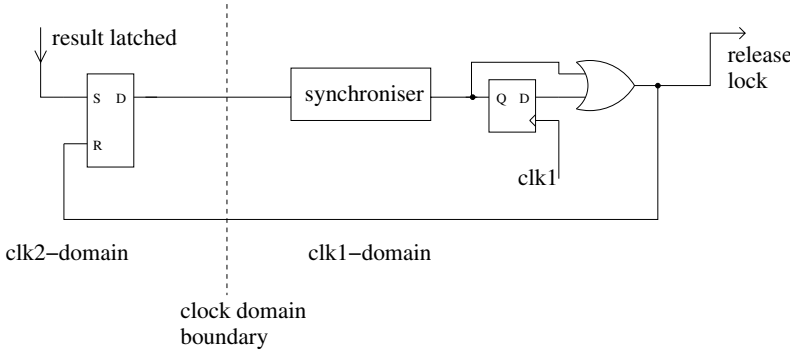


Fig. 5.17. Extending the inter-clock-domain call circuitry with an explicit arbiter release signal

implement asynchronous RS latches on FPGAs using 2 D-Type flip-flops with asynchronous resets as shown in Figure 5.16.

5.3.3 Modifying the Arbitration Circuitry

The synchronous arbiter shown in Figure 5.12 (and in context in Figure 5.10) uses the function's control output to release the lock. This is fine when both caller and callee are in the same clock domain as we know that the result will have been latched in the single cycle it takes to reset the lock flip-flop.

However, more care must be taken when dealing with separate clock domains: we need to make absolutely sure that the caller will have had time to read the result before the lock on the callee is released. To ensure that this is the case we need to modify the design of a functional-unit (Figure 5.10) so that the arbiter's release signal is only asserted once the caller has safely latched the function's shared data output.

For asynchronous calls (those which emanate from a different clock domain to the callee) we make the generation of the release signal the duty of the caller. The caller's control output signal (see Figure 5.15) is fed into the "result latched" signal of the circuit shown in Figure 5.17. The circuit operates on the same principle as the design of Figure 5.15 to convert a one-cycle pulse in the `clk2`

domain back into a one-cycle pulse in the `clk1` domain. Since the caller's control output is only asserted once the function's result has been latched we ensure that the function resource remains locked until its result has been read by the caller.

5.4 Summary

In this Chapter we have described the technical details involved in translating SAFL to hardware. Two back-ends have been described: one which targets synchronous designs; another which targets GALS circuits. One of SAFL's strengths is that it can be compiled to different design styles. Recall that this is not the case for behavioural HDLs which rely on structural blocks as an abstraction mechanism (e.g. Behavioural Verilog) since architecture-specific details necessarily become ingrained in the specification of inter-block communication mechanisms (see Section 3.1).

A topic of future work is to investigate synthesising SAFL to other design styles. In particular we are interested in various flavours of asynchronous design (e.g. fully delay-insensitive, matched delay etc. [40]).

6

Analysis and Optimisation of Intermediate Code

In this chapter we give examples of the kinds of analyses and optimisations which can be applied to the FLaSH compiler's intermediate graphs. The analyses and optimisations presented here have been implemented as part of the FLaSH compiler.

6.1 Architecture-Neutral verses Architecture-Specific

Before handing control over to one of the compiler's back-ends *architecture-neutral* analysis and optimisation is performed. We say that an optimisation is *architecture-neutral* if it does not rely on low-level assumptions about the chosen target technology. The key point is that architecture-neutral optimisation is able to improve the efficiency of generated hardware regardless of the specific design-style being targetted (e.g. it would be equally applicable to a synchronous design or a fully asynchronous design). Such analyses and optimisations are only made possible due to the high-level nature of SAFL. In languages such as Verilog and VHDL, even at the behavioural level, low-level assumptions about the underlying technology (e.g. timing assumptions) become too ingrained in the specification to support any concept of architecture-neutrality.

We have already seen an example of architecture-neutral analysis and optimisation in Chapter 4: Parallel Conflict Analysis (PCA), which operates at the abstract-syntax level, does not make any low-level assumptions about the target design-style but instead relies only on the temporal ordering of events enforced by the SAFL semantics (e.g. in computing `f(g(x), h(y))` the innermost calls `g(x)` and `h(y)` must have completed before the outermost call to `f` can take place).

In contrast, an *architecture-specific* analysis/optimisation is one which is only applicable to a particular design style (e.g. synchronous hardware). Architecture-specific optimisation helps the FLaSH compiler to generate more efficient designs: the more details known about the target technology, the greater the potential for analysis and optimisation.

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 87–111, 2004.

In this chapter we describe one analysis of each form which, in contrast to PCA, operate at the intermediate-code level:

- *Register Placement Analysis* (Section 6.3) is an architecture-neutral optimisation which allows us to reduce the number of temporary registers required to latch values of SAFL-expressions. (Recall that in the naïve compilation strategy outlined in the previous chapter, each call to a shared user-defined function, f, requires a separate temporary register to latch the value of f's shared output.)
- *Synchronous Timing Analysis* (Section 6.5) is an architecture-specific analysis which allows us to statically infer some timing information (in terms of cycle counts) about the resulting circuit. A number of optimisations are presented which make use of such timing information (Section 6.5.2).

6.2 Definitions and Terminology

Recall that a FLaSH intermediate graph is a triple $(\mathcal{N}, E_c, E_d)$ where:

$\mathcal{N}$ is a set of nodes
$E_c \subseteq \mathcal{N} \times \mathcal{N}$ is a set of *control edges*
i.e. $(n, n') \in E_c \Leftrightarrow$ control flows out of n into n'
$E_d \subseteq \mathcal{N} \times \mathcal{N}$ is a set of *data edges*
i.e. $(n, n') \in E_d \Leftrightarrow$ data flows out of n into n'

Also recall that given a node n, we define the formula $(n : \text{CALL } f)$ to hold iff n is a call node (and similarly for other node forms). We say that a node n is a *data-producer* if it has a data output-port. If n is a data-producer then we define $n.DataOut$ to refer to n's (single) data output-port.

We define functions to compute successors/predecessors as follows:
$Succ_c(n) = \{n' \mid (n, n') \in E_c\}$
$Succ_d(n) = \{n' \mid (n, n') \in E_d\}$
$Pred_c(n) = \{n' \mid (n', n) \in E_c\}$
$Pred_d(n) = \{n' \mid (n', n) \in E_d\}$

We define R^+ to be the transitive closure of relation R. For example, $Succ_c^+(n)$ is the set of nodes which occur after n on the control path. Similarly $R^\star$ is the reflexive-transitive closure of relation R.

6.3 Register Placement Analysis and Optimisation

The SAFL code fragment shown below was first presented in Section 5.2.2:

```
let var x = f(4)
 in let var y = f(5)
     in x + y
    end
end
```

According to the naïve compilation scheme described in Chapter 5 two temporary registers are synthesised: one for each call to `f`. The purpose of these temporary registers is to latch the value of `f`'s shared output (see Section 5.2.2). Recall that these temporary registers are called *permanising registers* since they make the result of computing an expression permanent, decoupling the caller from the callee.

It turns out that we can do significantly better than simply adding a permanising register on the output of every call node. In some circumstances we can infer that an expression will remain valid for as long as is necessary without requiring a temporary latch. In such cases permanisors are not required.

In this section we describe a new compilation scheme. To start with we assume an intermediate graph which has no permanisors. We then apply analyses to determine which expressions will become *invalid* prematurely if permanisors are not added. (Note that the only reason an expression can become invalid is if it *depends* on the output of a shared resource which is subsequently invoked by another caller—see example above). Temporary latches are then added to permanise precisely this set of expressions.

6.3.1 Sharing Conflicts

If, at run-time, an expression, e, is invalidated (i.e. another call is made to a shared resource on which e depends) we say that e has been subjected to a *sharing conflict*. It turns out that it is useful to further categorise sharing conflicts into two classes: *parallel sharing conflicts* and *sequential sharing conflicts*[1].

Sequential Sharing Conflicts

Assume that a SAFL program contains two calls to a shared function f. Following the notation of Chapter 4 we refer to these two distinct calls as f^α and f^β. Let us further assume that the first call to complete, f^α, is subjected to a sharing conflict as the second call, f^β, may change the shared output of hardware resource H_f.

We say that the sharing conflict is *sequential* if f^α will *always* be executed strictly before f^β (or vice-versa) regardless of the target technology in which the design is finally implemented. We have already seen an example of a sequential sharing conflict in Section 6.3. We can determine (due to the semantics of `let`) that the call `f(5)` only takes place after the call `f(4)` has terminated.

Parallel Sharing Conflicts

In contrast if f^α and f^β are involved in a *parallel sharing conflict* then (given no extra details about the underlying implementation) the order in which f^α

[1] The reason why this classification is helpful is that we already have an analysis that we can use to detect potential parallel sharing conflicts: namely, PCA (see Chapter 4). Hence we only need to develop a new analysis to detect *sequential sharing conflicts*.

and f^{β} are executed in is non-deterministic. This is the case when the two calls occur in separate parallel threads (compare and contrast the two intermediate graphs shown in Figure 6.1). Although Soft Scheduling ensures that an arbiter will be generated to dynamically serialise the concurrent accesses to the shared resource (see Chapter 4) we do not know which of the calls will be computed first[2]. Hence we have to place permanising registers after both calls—either may be corrupted by the other.

Representing Permanising Registers in Intermediate Code

An example of both a sequential conflict and a parallel conflict is shown in Figure 6.1. The horizontal dotted lines show the points where data may become invalid. It is at these points that permanising registers must be placed. In order to represent permanising registers at the intermediate level we introduce a new node which models a latch:

Node Type	*No. Control*		*No. Data*	
	Inputs	*Output*	*Inputs*	*Outputs*
PERMANISOR	1	1	1	1

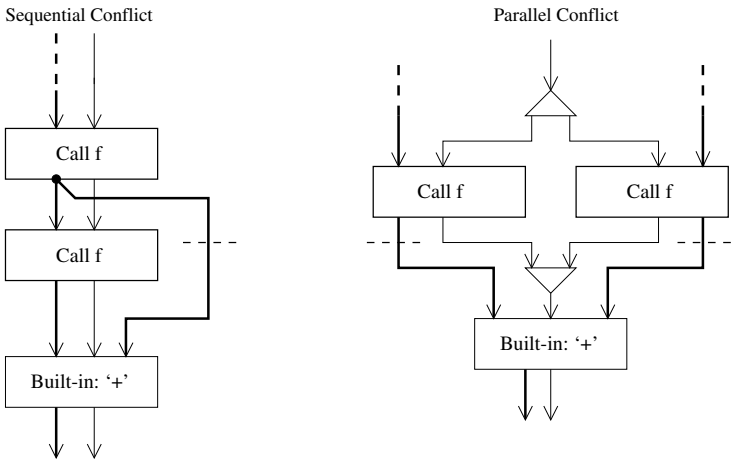


Fig. 6.1. A sequential conflict (left) and a parallel conflict (right). The horizontal dotted lines show the points where data may become invalid. These are the points where permanising registers are required

[2] Of course, if we had specific details about the target technology we could determine this information statically. However, by not making any assumptions about arbitration policies etc. we ensure that Register Placement Analysis is architecture-neutral—even after register placement has been performed a back-end is still free to choose different arbitration schemes for different functional units.

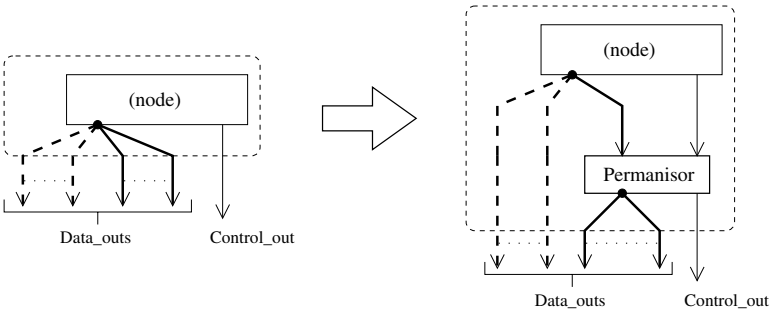


Fig. 6.2. We insert permanisors on data-edges using this transformation. The dashed data-edges represent those which do not require permanisors; the solid data-edges represent those which do require permanisors

On receiving control, this node latches its data input, propagating control once the data output (read directly from the latch) is valid. Once we have determined which data-edges require permanising we can insert PERMANISOR nodes using the transformation shown in Figure 6.2. The transformation is based on the observation that multiple data-edges originating at a single node can all share a permanisor if necessary.

6.3.2 Technical Details

Our register placement optimisation works by determining the places where an expression may be invalidated unless its value is latched. Rather than adopting the naïve approach of placing a PERMANISOR node on the output of a call to a shared function, we insert PERMANISORs only if the value of an expression is required after it may have become invalid.

The first stage is to insert permanising registers on the output of each call which is subject to a parallel sharing conflict (according to the results of Parallel Conflict Analysis—see Chapter 4). In this section we assume that this has already been done and continue by presenting a data-flow style analysis over intermediate graphs which allows us to infer which data-edges require permanising due to *sequential conflicts*. We also make the assumptions that hardware-level functional-units respect the following invariants:

Invariant 6.3.1 *After a call to a functional-unit, H_f, the data on H_f's (shared) output remains valid until another call to H_f occurs.*

Invariant 6.3.2 *Functions latch their arguments when called.*

Invariant 6.3.2 (analogous to *callee-save* in software compilers) means that the caller does not have to worry about keeping arguments valid *throughout* the duration of a call; arguments need only be valid at the point of call. The reader may wish to verify that our hardware implementation of functions does indeed satisfy these invariants (see Figure 5.10). In Section 6.4 we show how to modify the analysis to deal with a hybrid of both *callee-save* and *caller-save* policies.

The register placement analyses for detecting sequential conflicts are performed in three stages:

Resource Dependency Analysis tags each data-producing node, n, with the set of functional-units that $n.DataOut$ *depends* upon. (We say that a node, n, *depends* on a functional-unit, H_f, iff changing the value on H_f's (shared) output may *invalidate* (i.e. change) the value of $n.DataOut$.)

Validity Analysis tags each node, n, with the set of nodes whose data output is guaranteed to be *valid* when control reaches n. (We say that $n.DataOut$ is *valid* if its value has not been corrupted by a subsequent call which changes a shared function output on which n depends.)

Sequential Conflict Register Placement uses validity information to decide which data-edges require permanising registers to resolve sequential conflicts.

We require the following definition:

Definition 6.3.1. *CG is the call-graph relation of the program being translated. Thus, $CG(f)$, is the set of functions that f may call directly and $CG^\star(f)$ is the set of all functions which may be invoked as a result of invoking f.*

Dataflow equations for the register placement analysis are summarised in Figure 6.5. The following sections clarify some of the terminology and describe the intuition behind the equations:

6.3.3 Resource Dependency Analysis

Recall that a data-producing node, n, *depends* on a functional-unit, H_f, iff changing the value on H_f's (shared) output may invalidate the value of $n.DataOut$. The resource dependency equations map a node, n, onto the set of functional-units on which n depends.

Definition 6.3.2. *Given a node, n, $\mathcal{D}_{out}(n)$ is the set of (names of) functional-units that n depends on.*

$$\mathcal{D}_{out} : \mathit{Node} \to \mathcal{P}(\mathit{Functional\text{-}unit\ name})$$

$$\mathcal{D}_{out}(n) = \begin{cases} \emptyset & \text{if } (n : \textsc{jump}) \vee (n : \textsc{permanisor}) \\ CG^\star(f) & \text{if } (n : \textsc{call}\ f) \\ \bigcup_{p \in Pred_d(n)} \mathcal{D}_{out}(p) & \text{otherwise} \end{cases}$$

The resource dependency equations reflect the following observations:

1. If n is a JUMP or a PERMANISOR then its output is not dependent on any functional-units.

2. If n is a CALL node (n : CALL f) then $n.DataOut$ is dependent on f and the functional-units that f may access. We know that $n.DataOut$ is not dependent on any of its data-predecessors since H_f latches these values at the beginning of the call decoupling n from all $n' \in Pred_d(n)$.
3. Otherwise n is dependent on the same functional-units as its data-predecessors since changes to the data-outputs of any $n' \in Pred_d(n)$ may be propagated through to $n.DataOut$. This clause also handles the case of n being an ENTRY_NODE since in this case $Pred_d(n) = \emptyset$.

6.3.4 Data Validity Analysis

The data validity equations form the core of the register placement analysis. Defined mutually recursively, $\mathcal{V}_{in}$ and $\mathcal{V}_{out}$ map a node, n, onto the set of nodes whose data-output is *guaranteed* to be valid when control respectively reaches and leaves n. However, before launching into the data validity equations we must first introduce some auxiliary definitions. Definitions 6.3.3 and 6.3.4 formalise the notion of a *thread* allowing us to reason precisely about parallelism (they are shown diagrammatically in Figure 6.3). Definition 6.3.5 defines *kill* which maps a node, n, onto the set of nodes whose data outputs are invalidated as a result of control passing through n.

Definition 6.3.3. *Given a* CONTROL_SPLIT *node, n, $Join(n)$ is the corresponding* CONTROL_JOIN *node*[3].

Definition 6.3.4. *Given a* CONTROL_SPLIT *node, n, such that $Succ_c(n) = \{s_1, \ldots, s_k\}$, let $thread_i = Succ_c^\star(s_i) \cap Pred_c^+[Join(n)]$ (for $1 \leq i \leq k$). Then, $\pi(n, s_i)$ is the set of nodes in each of n's threads except the thread containing node s_i:*

$$\pi(n, s_i) = \bigcup_{j \neq i} thread_j$$

Definition 6.3.5. *Given a node n, $kill(n)$ is the set of nodes whose data outputs are invalidated as a result of control passing through n:*

$$kill : Node \rightarrow \mathcal{P}(Node)$$

$$kill(n) = \begin{cases} \{n' \neq n \mid \mathcal{D}_{out}(n') \cap CG^\star(f) \neq \emptyset\} & \text{if } (n : \text{CALL } f) \\ \emptyset & \text{otherwise} \end{cases}$$

[3] Due to the properties of the translation to intermediate code each CONTROL_SPLIT node has a corresponding CONTROL_JOIN node (cf. *bras* and *kets* in a well-bracketed string).

The equations for *kill* reflect the following observations:

- The only way a node's data output can be invalidated by executing n is if n invokes some shared resource. Thus if n is not a CALL node then nothing can be invalidated.
- If n is a call node (n:CALL f) then every node which is dependent on something which n may invoke (either directly or indirectly) is invalidated.

The data validity equations are now presented:

Definition 6.3.6. *Given a node, n, $\mathcal{V}_{in}(n)$ is the set of nodes whose data-output is guaranteed to be valid when control* reaches *n.*

Definition 6.3.7. *Given nodes n and s, $\mathcal{V}^s_{out}(n)$ is the set of nodes which are guaranteed to be valid when control* leaves *n along edge $(n, s) \in E_c$*

For the sake of clarity, in cases where we know that n has only one control successor (i.e. $Succ_c(n) = \{n'\}$), we write $\mathcal{V}^\diamond_{out}(n)$ to mean $\mathcal{V}^{n'}_{out}(n)$.

$$\begin{aligned}\mathcal{V}_{in} &: Node \to \mathcal{P}(Node)\\ \mathcal{V}^s_{out} &: Node \to \mathcal{P}(Node)\end{aligned}$$

$$\mathcal{V}_{in}(n) = \begin{cases} \bigcap\limits_{p \in Pred_c(n)} \mathcal{V}^\diamond_{out}(p) \text{ if } (n : \text{CONDITIONAL\_JOIN}) \\ \bigcup\limits_{p \in Pred_c(n)} \mathcal{V}^n_{out}(p) \text{ otherwise} \end{cases}$$

$$\mathcal{V}^s_{out}(n) = \mathcal{V}_{in}(n) \cup \{n\} \setminus \begin{cases} \bigcup\limits_{n' \in \pi(n,s)} kill(n') \text{ if } (n : \text{CONTROL\_SPLIT}) \\ kill(n) \qquad\qquad \text{otherwise} \end{cases}$$

The intuition behind $\mathcal{V}_{in}(n)$ is as follows:

1. If n is a CONDITIONAL_JOIN node then at run-time control will arrive at n from *either* its true-branch *or* its false-branch. Thus the nodes guaranteed to be valid when control reaches n are those that are guaranteed to be valid at *both* the end of the true-branch *and* the end of the false-branch.
2. If n is not a CONDITIONAL_JOIN node then the nodes that are guaranteed to be valid when control reaches n are those that were guaranteed to be valid just after n's control-predecessors have been executed.

$\mathcal{V}^s_{out}(n)$ reflects the following intuition:

- If n is not a CONTROL_SPLIT node then the nodes guaranteed to be valid when control leaves n are precisely:
  - those nodes which were valid when control arrived at n;
  - plus n itself;
  - minus the nodes that were invalidated as a result of executing n. i.e. those nodes in $kill(n)$

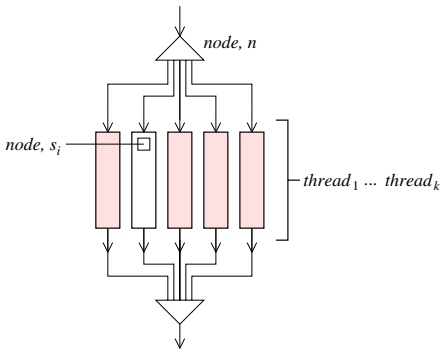


Fig. 6.3. The nodes contained in the highlighted threads are those returned by $\pi(n, s_i)$

– If n is a CONTROL_SPLIT node then things are a little more complicated since we have to cope with the parallelism that n introduces. As we do not know statically which interleaving of parallel operations will occur at run-time we are forced to make a *safe* approximation: when analysing a particular thread (e.g. the one containing s_i in Figure 6.3) we assume that every other parallel thread has already been executed (i.e. we assume that any nodes whose data *may* be invalidated *have* been invalidated).

6.3.5 Sequential Conflict Register Placement

Sequential conflict register placement is the process where we decide which data-edges $(n, n') \in E_d$ require registers to resolve sequential conflicts. We define a predicate $Perm(n, n')$ which holds iff data-edge (n, n') requires a permanisor. As a first approximation, we simply observe that if a node n is not guaranteed to be valid at n' then we must place a permanising register on data-edge (n, n'):

$$\forall (n, n') \in E_d.\ Perm(n, n') \Leftrightarrow n \notin \mathcal{V}_{in}(n')$$

Although this works, it has a tendency to insert unnecessary permanisors where conditionals are concerned. To see the problem consider the following code fragment

```
if t(x) then f(x) else g(x)
```

and visualise the corresponding intermediate graph. The problem is that, according to our data validity analysis, neither `f(x)` nor `g(x)` are valid on entry to the CONDITIONAL_JOIN node (since the equations for validity analysis state that the only nodes which are guaranteed to be valid when control reaches a CONDITIONAL_JOIN are those which are valid at the end of *both* branches of the conditional). Thus two permanisors will be inserted to latch the results of the calls to `f(x)` and `g(x)`. It is clear, however, that these permanisors are unnecessary: if we take the true-branch of the conditional then `f(x)` will be valid and we do not care about `g(x)`; conversely if we take the false-branch `g(x)` will be valid and we do not care about `f(x)`.

We can improve the accuracy of our model (i.e. make it insert considerably fewer permanisors) by giving CONDITIONAL_JOIN nodes a special treatment.

Definition 6.3.8. *Given a* CONDITIONAL_JOIN *node, n', and a node, n, which occurs before n' on the control path*[4]*, $\mathcal{RV}_{in}(n, n')$ 'Relative-$\mathcal{V}_{in}$' is the set of nodes guaranteed to be valid on entry to n' given the extra information that control passes through node n.*

$$\mathcal{RV}_{in}(n, n') = \bigcap_{n'' \in (Succ_c^\star(n) \cap Pred_c(n'))} \mathcal{V}^\diamond_{out}(n'')$$

This is based on the observation that if we know which way a CONDITIONAL_SPLIT has branched, we can do much better at predicting the nodes that are going to be valid at the corresponding CONDITIONAL_JOIN: the nodes which are valid at n' are those which are valid at the end of the conditional branch in which n occurs. We use the equation $Succ_c^\star(n) \cap Pred_c(n')$ to calculate the final node in the conditional branch containing n (shown graphically in Figure 6.4). Note that if n is not in either of the conditional branches joining at n' then (since n occurs before n' on the control path) $Succ_c^\star(n) \cap Pred_c(n') = Pred_c(n')$. Thus, in this case $\mathcal{RV}_{in}(n, n')$ reduces to $\mathcal{V}_{in}(n')$ (intuitively this is what we expect: if n is not in either of the conditional branches joining at n' then we have not gained any extra information by stating that control passes through n.)

The vertically hatched nodes are those in $Succ_c^\star(n)$; the horizontally hatched nodes correspond to $Pred_c(n')$. Only control flow is shown.

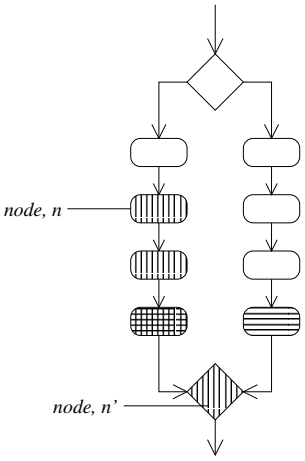


Fig. 6.4. Diagrammatic explanation of $Succ_c^\star(n) \cap Pred_c(n')$

Now we can use $\mathcal{RV}_{in}$ to define a more accurate version of *Perm* as follows:

[4] I.e. $n' \in Succ_c^+(n)$.

Definition 6.3.9. *$Perm(n, n')$ holds iff data-edge (n, n') requires* permanising*:*

$$\forall (n, n') \in E_d.\ Perm(n, n') \Leftrightarrow \begin{cases} n \notin \mathcal{RV}_{in}(n, n') \\ \qquad\qquad \text{if } (n' : \textsc{conditional\_join}) \\ n \notin \mathcal{V}_{in}(n') \quad \text{otherwise} \end{cases}$$

6.4 Extending the Model: Calling Conventions

The equations presented in Figure 6.5 assume a *callee-save* model. In a hardware implementation this corresponds to every functional-unit latching its arguments on a call. Sometimes these latches are unnecessary and savings can be made by adopting a *caller-save* model, where functional-units do not latch their arguments but make the assumption that the caller will keep the arguments valid throughout the duration of the call.

For the sake of completeness we show in this section how the data-flow analyses can be modified to cope with a combination of both callee-save and caller-save conventions. Let predicate $CalleeSave(f)$ hold for a function, f, iff f adopts a callee-save model. In this way we can specify on a per-resource basis which functional-units latch their arguments and which functional-units require their arguments to remain valid throughout a call. (We use a suitable pragma to convey this information to the compiler; however, one could imagine a global analysis which attempts to choose the best strategy). Thus, for caller-save functional-units, our invariants (see Section 6.1) are replaced with the following:

Invariant 6.4.1 *After a call to a (caller-save) functional-unit, H_f, the data on H_f's (shared) output remains valid until either (i) another call to H_f occurs or (ii) the values on H_f's inputs change.*

Invariant 6.4.2 *The caller keeps the arguments valid throughout the duration of the call.*

It turns out that we only have to modify the resource dependency analysis and the permanisation analysis; validity analysis remains unchanged.

6.4.1 Caller-Save Resource Dependency Analysis

We add an extra clause to the definition of $\mathcal{D}_{out}$ (see Section 6.3.3) reflecting the observation that for a node n : CALL f, where H_f does not latch its arguments, n is dependent upon:

- the functional-units that n's data predecessors are dependent upon; and
- all the functional-units that H_f may invoke.

Resource Dependency Analysis

$$\mathcal{D}_{out} : Node \to \mathcal{P}(Function\ name)$$

$$\mathcal{D}_{out}(n) = \begin{cases} \emptyset & \text{if } (n : \textsc{jump}) \vee [(n : \textsc{call}\ f) \wedge HasParConflict(n)] \\ CG^\star(f) & \text{if } (n : \textsc{call}\ f) \wedge \neg HasParConflict(n) \\ \bigcup_{p \in Pred_d(n)} \mathcal{D}_{out}(p) & \text{otherwise} \end{cases}$$

Data Validity Analysis

$$\mathcal{V}_{in} : Node \to \mathcal{P}(Node)$$
$$\mathcal{V}^s_{out} : Node \to \mathcal{P}(Node)$$

$$\mathcal{V}_{in}(n) = \begin{cases} \bigcap_{p \in Pred_c(n)} \mathcal{V}^\diamond_{out}(p) & \text{if } (n : \textsc{conditional\_join}) \\ \bigcup_{p \in Pred_c(n)} \mathcal{V}^n_{out}(p) & \text{otherwise} \end{cases}$$

$$\mathcal{V}^s_{out}(n) = \mathcal{V}_{in}(n) \cup \{n\} \setminus \begin{cases} \bigcup_{n' \in \pi(n,s)} kill(n') & \text{if } (n : \textsc{control\_split}) \\ kill(n) & \text{otherwise} \end{cases}$$

where $\pi(n, s)$ (see Definition 6.3.4 and Figure 6.3) is the set of nodes in each of CONTROL_SPLIT node n's threads except the thread containing node s.

and $kill(n)$ is the set of nodes whose data outputs are invalidated as a result of control passing through n:

$$kill : Node \to \mathcal{P}(Node)$$

$$kill(n) = \begin{cases} \{n' \neq n \mid \mathcal{D}_{out}(n') \cap CG^\star(f) \neq \emptyset\} & \text{if } (n : \textsc{call}\ f) \\ \emptyset & \text{otherwise} \end{cases}$$

$\mathcal{RV}_{in}(n, n')$ is the set of nodes valid on entry to the CONDITIONAL_JOIN node n' given the extra information that control passes through node n.

$$\mathcal{RV}_{in}(n, n') = \bigcap_{n'' \in (Succ^\star_c(n) \cap Pred_c(n'))} \mathcal{V}^\diamond_{out}(n'')$$

Sequential Conflict Register Placement

$$\forall (n, n') \in E_d.\ Perm(n, n') \Leftrightarrow \begin{cases} n \notin \mathcal{RV}_{in}(n, n') & \text{if } (n' : \textsc{conditional\_join}) \\ n \notin \mathcal{V}_{in}(n') & \text{otherwise} \end{cases}$$

Fig. 6.5. Summary: Register Placement for Sequential Conflicts

Hence we have:

$$\mathcal{D}_{out}(n) = \begin{cases} \emptyset & \text{if } (n : \textsc{jump}) \vee [(n : \textsc{call}\ f) \wedge \\ & \qquad HasParConflict(n)] \\ CG^{\star}(f) & \text{if } (n : \textsc{call}\ f) \wedge \\ & \qquad \neg HasParConflict(n) \wedge \\ & \qquad CalleeSave(f) \\ CG^{\star}(f) \cup \bigcup_{p \in Pred_d(n)} \mathcal{D}_{out}(p) & \text{if } (n : \textsc{call}\ f) \wedge \\ & \qquad \neg HasParConflict(n) \wedge \\ & \qquad \neg CalleeSave(f) \\ \bigcup_{p \in Pred_d(n)} \mathcal{D}_{out}(p) & \text{otherwise} \end{cases}$$

6.4.2 Caller-Save Permanisation Analysis

We update our definition of *Perm* to reflect the observation that if we are not dealing with a callee-save function, it is the duty of the caller to keep the function arguments valid until after the call.

$$\forall (n, n') \in E_d.\ Perm(n, n') \Leftrightarrow \begin{cases} n \notin \mathcal{RV}_{in}(n, n') & \\ \qquad \text{if } (n' : \textsc{conditional\_join}) & \\ n \notin \mathcal{V}^{\circ}_{out}(n') & \\ \qquad \text{if } (n' : \textsc{call}\ f) \wedge \neg CalleeSave(f) & \\ n \notin \mathcal{V}_{in}(n') & \text{otherwise} \end{cases}$$

6.5 Synchronous Timing Analysis

This section turns to the problem of architecture-specific analysis. We have developed several architecture-specific optimisations which can improve the efficiency of generated *synchronous* hardware. Each of these optimisations is based on an analysis which statically infers the number of cycles that operations will take to execute.

Although, in general, it is an undeciable problem to determine the length of a function's execution, in practice there are many cases where it is feasible. For example, consider the following SAFL program:

```
fun g(x) = x+4
fun f(x,y) = let val z = g(x)
              in z*y
             end
```

Here, assuming that `g` and `f` are both callee-save functions (which latch their arguments), we can deduce that a call to `f` will take 2 cycles: one cycle to latch its arguments, and a second cycle for `g` to latch its argument. Note that our analysis has to take permanising registers into account since these add an extra cycle latency. Hence we apply synchronous timing analysis at the intermediate-code level after register placement has been performed since it is only at this point that we know where permanisors have been inserted.

6.5.1 Technical Details

First some more definitions. We define $\mathbb{N}_\infty$ to be the set of non-negative integers augmented with the symbol ∞:

$$\mathbb{N}_\infty \stackrel{\text{def}}{=} \mathbb{N} \cup \{\infty\}$$

Integer addition is extended to $\mathbb{N}_\infty$ by adopting the rule:

$$\forall x \in \mathbb{N}_\infty.(x + \infty = \infty) \wedge (\infty + x = \infty)$$

The integers in $\mathbb{N}_\infty$ are subject to the usual total ordering; ∞ is larger than every integer element of $\mathbb{N}_\infty$. We use the notation, $[x_1, x_2]$, to represent closed intervals in $\mathbb{N}_\infty$ (where $x_1 \leq x_2$). The following operations are defined on intervals:

$$\begin{aligned}
\text{Addition: } [x_1, y_1] + [x_2, y_2] &\stackrel{\text{def}}{=} [x_1 + x_2, y_1 + y_2] \\
\text{Widening: } [x_1, y_1] \cup [x_2, y_2] &\stackrel{\text{def}}{=} [min(x_1, x_2), max(y_1, y_2)] \\
\text{Join: } [x_1, y_1] \odot [x_2, y_2] &\stackrel{\text{def}}{=} [max(x_1, x_2), max(y_1, y_2)]
\end{aligned}$$

Predicate $isEntry(n)$ holds iff node n is marked as a function's entry node (see Section 5.1.2). Predicate $isRecursive(f)$ holds iff the SAFL function with name f contains at least one (tail) recursive call. The function

$$exitNode(f) : Function\ Name \rightarrow Node$$

maps a function name, f, onto the exit node of f's intermediate graph. Given a CALL node, n, predicate $HasParConflict(n)$ holds iff n is a call node, and the call is subject to a parallel sharing conflict (as detected by PCA—see Chapter 4).

The Synchronous Timing Analysis equations are defined in Figure 6.6. The functions $\mathcal{T}_{in}(n)$ and $\mathcal{T}_{out}(n)$ map nodes in an intermediate graph to intervals in $\mathbb{N}_\infty$. If for a node, n, contained in the intermediate graph of a function, f, $\mathcal{T}_{out}(n) = [x, y]$ then we know that control will *leave* n a *minimum* of x cycles after f is called and a *maximum* of y cycles after f is called. Similarly if $\mathcal{T}_{in}(n) = [x, y]$ then we know that control will *arrive* at n a minimum of x cycles after f is called and a maximum of y cycles after f is called. If y is ∞ then this signifies that our analysis cannot statically infer the maximum time (respectively) that control will leave n [5]. This section continues by explaining the informal intuition behind the equations of Figure 6.6.

[5] Due to the structure of the equations—see Figure 6.6—x can never be ∞.

Definition 6.5.1. *Given a node, n, $\mathcal{T}_{in}(n)$ is the interval during which control is guaranteed to arrive at n.*

- If n is an entry-node then it will be executed 1 cycle after its enclosing function is invoked. (The 1 cycle latency is due to the time taken to latch its arguments—see Section 5.2.2. Note that, for the purposes of this presentation, we assume that functions are *caller-save*; a *callee-save* function does not incur this 1 cycle latency.)
- If n is a CONTROL_JOIN node then we wait for control to arrive from each $n' \in Pred_c(n)$ before proceeding. (This behaviour is reflected by the join operator, $\odot$.)
- Otherwise, we determine the interval at which control arrives at n by widening the intervals of n's control predecessors.

Definition 6.5.2. *Given a node, n, $\mathcal{T}_{out}(n)$ is an interval during which control is guaranteed leave n (and hence the interval during which n.DataOut is guaranteed to become valid).*

- If (n : PERMANISOR) then we know that 1 cycle is required for n to latch its data input.
- If (n : CALL f) then the time control leaves n is determined by the time that control enters n and also the time taken for function f to be executed. If the call has a parallel sharing conflict (as detected by PCA—see Chapter 4) then we do not know how long f will take to execute since the execution time depends on whether H_f is busy/free. However, we do know that f will take a minimum of 2 cycles to execute since the arbiter presented in Section 5.2.2 adds an extra cycle latency to the call.
- If n is not any of the nodes listed above then control passes straight through without taking any extra cycles.

Definition 6.5.3. *Given a SAFL function name f, $FT(f)$ is the interval which safely bounds the execution time of the function.*

- If f contains (tail) recursive calls then we do not try and analyse its maximum execution time; instead we safely approximate it as ∞. However, even if f is recursive we know that the minimum time possible for f's execution is 1 cycle since this is the time required for H_f to latch its incoming arguments.
- If f is not recursive then the time taken for f to execute is determined by the time that control leaves f's exit node.

6.5.2 Associated Optimisations

The optimisations associated with synchronous timing analysis are outlined in this section. We start by extending our functions *min* and *max* to operate on intervals in the obvious way:

$$min[x, y] = x$$
$$max[x, y] = y$$

$$\mathcal{T}_{in} : \mathit{Node} \to [\mathbb{N}_\infty, \mathbb{N}_\infty]$$
$$\mathcal{T}_{out} : \mathit{Node} \to [\mathbb{N}_\infty, \mathbb{N}_\infty]$$

$$\mathcal{T}_{in}(n) = \begin{cases} [1,1] & \text{if } \mathit{isEntry}(n) \\ \bigodot_{p \in Pred_c(n)} \mathcal{T}_{out}(n) & \text{if } n : \textsc{control\_join} \\ \bigcup_{p \in Pred_c(n)} \mathcal{T}_{out}(n) & \text{otherwise} \end{cases}$$

$$\mathcal{T}_{out}(n) = \begin{cases} \mathcal{T}_{in}(n) + [1,1] & \text{if } n : \textsc{permanisor} \\ \mathcal{T}_{in}(n) + FT(f) & \text{if } n : \textsc{call } f \wedge \neg \mathrm{HasParConflict}(n) \\ \mathcal{T}_{in}(n) + [2,\infty] & \text{if } n : \textsc{call } f \wedge \mathrm{HasParConflict}(n) \\ \mathcal{T}_{in}(n) & \text{otherwise} \end{cases}$$

$$FT : \mathit{Function\ Name} \to [\mathbb{N}_\infty, \mathbb{N}_\infty]$$

$$FT(f) = \begin{cases} [1,\infty] & \text{if } \mathit{isRecursive}(f) \\ \mathcal{T}_{out}(\mathit{exitNode}(f)) & \text{otherwise} \end{cases}$$

Fig. 6.6. Synchronous Timing Analysis

Removing Unnecessary Arbitration

Consider translating the following SAFL program into synchronous hardware:

```
fun f(x) = g(h(x+1), h(k(x+2)))
```

Note that we can remove the arbiter for `h` if we can infer that the execution of `k` always requires more cycles than the execution of `h`. More generally, if n_1 and n_2 occur in parallel threads of an intermediate graph and are both of type (CALL f) then if:

$$min(\mathcal{T}_{in}(n_1)) > max(\mathcal{T}_{out}(n_2))$$

we know that the call at n_2 will have fully completed before the call at n_1 starts (since the minimum time at which control can enter n_1 is greater than the maximum time at which control leaves n_2). Conversely, if:

$$min(\mathcal{T}_{in}(n_2)) > max(\mathcal{T}_{out}(n_1))$$

then we know that the call at n_1 will have fully completed before the call at n_2 starts.

If either of these two conditions hold then we know that the two calls to f will occur in non-overlapping time intervals, thus the need for dynamic arbitration is eliminated. This optimisation essentially allows us to improve the accuracy of PCA given more information about the target technology (in this case synchronous hardware). Obviously this optimisation is not applicable to asynchronous designs as it stands. However, it may be possible to use detailed feedback from model simulations incorporating layout delays to enable a similar type of optimisation in the asynchronous case. Investigating this claim is a topic of future work.

Eliminating Control_Joins and Function Call Control Signals

In many cases we can use cycle counting to optimise CONTROL_JOINs away. If we can infer that control will arrive at each of the control inputs simultaneously then we no longer need a CONTROL_JOIN node. Instead, we can pick any one of the control inputs (arbitrarily) and connect it to the control output. As well as removing the need for a CONTROL_JOIN circuit (see Section 5.2.1) this optimisation offers another potential saving: since some control signals are no longer connected to anything, we can remove the circuitry that generates them (see Figure 6.7).

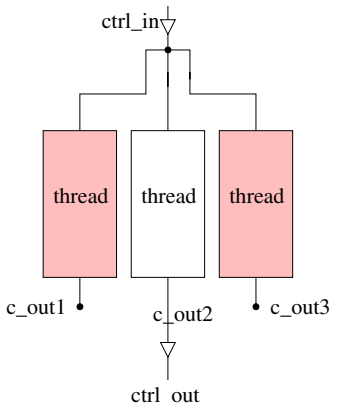


Fig. 6.7. A block diagram of a circuit-level implementation of 3 parallel threads. Suppose that our analysis has detected that the "done" control outputs of the 3 threads will be asserted simultaneously. Thus we have no need for a CONTROL_JOIN NODE. Since signals "c_out1" and "c_out3" are no longer connected to anything we can optimise away the control circuitry of the shaded blocks

Thus we know that a CONTROL_JOIN node n can be eliminated if:

$$min(\mathcal{T}_{in}(n)) = max(\mathcal{T}_{in}(n))$$

In fact we can do better than this. If we know which of a CONTROL_JOIN's inputs will be asserted *last* then we can simply connect this input to the control output

(and leave other control inputs unconnected). Let n be a CONTROL_JOIN node. The set of n's control predecessors whose control outputs finish last is given by:

$$\{x \in Pred_c(n) \mid \forall y \in (Pred_c(n) \backslash \{x\}).min(\mathcal{T}_{out}(x)) \geq max(\mathcal{T}_{out}(y))\}$$

Intuitively these are the nodes whose *earliest* possible completion times are greater than the *latest* possible completion times of the rest of n's control precessors. Instead of implementing an explicit CONDITIONAL_JOIN node we can simply choose a node from this set and propogate its control output (see Figure 6.7)[6]. Of course if the set is empty it means that our analysis has not been able to infer any useful information. In this case a the CONDITIONAL_JOIN must be implemented in hardware in the usual way.

Recall that, for each function call, control circuitry is required to implement the call/return mechanism (see Figure 4.4). However, if we can statically determine the number of cycles that a function call will take then there is scope to eliminate some of this control circuit. Instead of using a control signal from the callee to detect the termination of the call, the caller can instead use a counter to simply wait for the appropriate number of cycles. In this way the need for a control return wire from the callee is eliminated. As a result the callee's ECCU (see Figure 5.11) can also be simplified since the callee no longer has to remember the caller's identity.

Let (n:CALL f) be a node which represents a function call. We can statically determine the exact time of the call if:

$$\neg HasParConflict(n) \wedge min(FT(f)) = max(FT(f))$$

6.6 Results and Discussion

In this section we present experimental results (area/time figures) which demonstrate the effects of both Register Placement Analysis and the synchronous timing optimisations.

6.6.1 Register Placement Analysis: Results

We start by comparing designs generated using the register placement techniques (described in Section 6.3.2 with the naïve compilation strategy outlined in Section 5.2.2). As expected, when a design contains no shared resources, both compilation strategies produce identical results. As the amount of resource sharing increases the differences between the two approaches becomes more apparent.

To measure the effects of register placement analysis we consider compiling the expression $u - 3 * x * u * dx - 3 * y * dx$ (taken from the differential equation

[6] Although we are free to choose any node from this set it is advantageous to choose the one with the least amount of control circuitry in its thread. Since we remove much of the control circuitry from the other threads this strategy allows us to save more area.

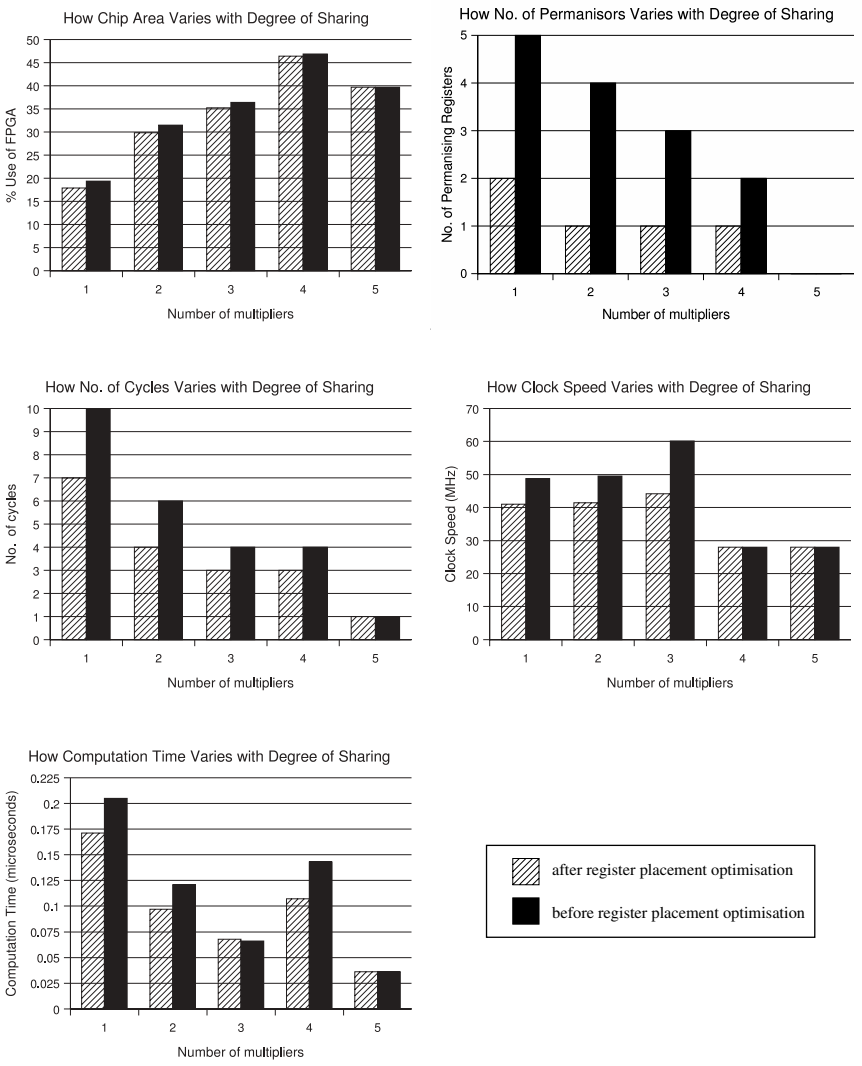


Fig. 6.8. How various paramaters (area, number of permanisors, number of cycles, clock speeds and computation time) vary as the degree of resource sharing changes

solver example of Section 1.1) using different numbers of 32-bit multipliers. For any given number of multipliers there are a variety of possible SAFL programs which can compute this expression (reflecting different scheduling and binding decisions). Thus it is important that we specify the exact code fragments we used in this experiment: Figure 6.9 presents the 5 SAFL programs which were chosen to measure the effects of Register Placement Analysis. Note that we use the `let` constructs to specify our chosen static schedules explicitly in each case. Thus, no dynamic arbitration is required to schedule access to the shared multipliers.

No. of ∗'s	SAFL Program

1	`fun mult1(x:32,y:32):32 = x*y`
	`fun main(x:32,u:32,dx:32,y:32):32 =`
	`  let val t1 = mult1(3, x)`
	`  ---`
	`  val t2 = mult1(u,dx)`
	`  ---`
	`  val t4 = mult1(t1,t2)`
	`  ---`
	`  val t3 = mult1(y,dx)`
	`  ---`
	`  val t5 = mult1(3, t3)`
	`  in u - t4 - t5`
	`  end`

2	`fun mult1(x:32,y:32):32 = x*y`
	`fun mult2(x:32,y:32):32 = x*y`
	`fun main(x:32,u:32,dx:32,y:32):32 =`
	`  let val t1 = mult1(3:32, x)  val t2 = mult2(u,dx)`
	`  ---`
	`  val t3 = mult1(y,dx)`
	`  ---`
	`  val t4 = mult2(t1,t2)  val t5 = mult1(3, t3)`
	`  in u - t4 - t5`
	`  end`

3	`fun mult1(x:32,y:32):32 = x*y`
	`fun mult2(x:32,y:32):32 = x*y`
	`fun main(x:32,u:32,dx:32,y:32):32 =`
	`  let val t1 = mult1(3:32, x)  val t2 = mult2(u,dx)`
	`  val t3 = y*dx`
	`  ---`
	`  val t4 = mult2(t1,t2)  val t5 = mult1(3, t3)`
	`  in u - t4 - t5`
	`  end`

4	`fun mult1(x:32,y:32):32 = x*y`
	`fun main(x:32,u:32,dx:32,y:32):32 =`
	`  let val t1 = mult1(3:32, x)  val t2 = u*dx`
	`  val t3 = mult1(y,dx)`
	`  ---`
	`  val t4 = t1*t2  val t5 = mult1(3, t3)`
	`  in u - t4 - t5`
	`  end`

5	`fun main(x:32,u:32,dx:32,y:32):32 =` `  u - 3*x*u*dx - 3*y*dx`

Fig. 6.9. SAFL programs with different degrees of resource sharing

We deliberately choose an arithmetic example since this is where the register placement analysis has the most impact. Although our examples are small, we argue that the savings are still meaningful in the context of larger designs. For example, consider a larger design which repeatedly computes the value of our arithmetic expression, $u - 3 * x * u * dx - 3 * y * dx$, as part of an inner-loop. In this case reducing the number of permanisors in the arithmetic expression may significantly improve the overall performance of the system (recall that each permanising register adds another cycle of latency).

We compiled each of the SAFL programs of Figure 6.9 to synchronous hardware using both the naïve compilation strategy (Section 5.2.2) and the more complex register placement strategy (Section 6.3.2). The RTL output of the FLaSH compiler was mapped to an Altera Apex-II FPGA (EP20K200EFC484-2X; 200×10^3 gate equivalent) using the industrial synthesis tools *Leonardo* (Altera edition) and *Quartus-II*. The results, which show how the degree of resource sharing affects different parameters (chip area, number of permanisors, number of cycles taken to compute, clock speeds and actual computation time in seconds), are presented graphically in Figure 6.8. We give the results in tabular form and discuss their significance in the following sections.

Chip Area

Figure 6.10 demonstrates that the Register Placement strategy does indeed insert fewer permanising registers than the naïve compilation strategy. Figure 6.11 shows how these area savings are reflected at the chip-level. As expected, the %-saving (both in terms of number of permanisors and chip area) shows a downwards trend as the number of multipliers in the design are increased. Finally, when the design contains 5 multipliers then, since there is no longer any resource sharing, there is no difference between the two compilation strategies.

No. of Multipliers	No. of Permanisors		% Saving
	With Reg Placement	Naive compilation	
1	2	5	60
2	1	4	75
3	1	3	67
4	1	2	50
5	0	0	0

Fig. 6.10. Number of Permanising Registers

Computation Time

Figure 6.12 shows how the number of clock cycles required for the computation decreases as the number of multipliers in the circuit increases. The reason for this decrease is simply because the increased number of multipliers allows more of

No. of Multipliers	Chip Area (%-use of FPGA)		% Saving
	With Reg Placement	Naive compilation	
1	17.85	19.4	7.99
2	29.86	31.44	5.03
3	35.20	36.42	3.35
4	46.43	46.86	0.92
5	39.71	39.71	0.00

Fig. 6.11. Chip area (as %-use of FPGA)

No. of Multipliers	No. of Cycles		% Saving
	With Reg Placement	Naive compilation	
1	7	10	30
2	4	6	33
3	3	4	25
4	3	4	25
5	1	1	0

Fig. 6.12. Number of clock cycles required for computation

No. of Multipliers	Clock Speed (MHz)		% Saving
	With Reg Placement	Naive compilation	
1	41.0	48.8	−16.0
2	41.4	49.5	−16.4
3	44.2	60.1	−26.5
4	28	28	0
5	28	28	0

Fig. 6.13. Clock Speeds of Final Design

the computation to be performed in parallel. Note that the Register Placement compilation strategy offers less savings in terms of cycle time as the number of multipliers in the circuit increases. This is exactly as we would expect: the savings in cycle time are entirely due to the reduction in permanising registers (since it takes an extra cycle to latch data into a permanising register).

Figure 6.13 shows how the clock speeds of the generated designs are affected by the degree of sharing. The interesting point to note is that the Register Placement strategy actually leads to *slower* clock speeds than the naïve compliation scheme. The reason for this is that as permanising registers are removed the critical path becomes longer. These results also show just how difficult it is to predict how a design will behave when it is finally mapped to hardware. The huge drop in clock speed to 28MHz is incurred as the designs become larger because the place-and-route tool starts spreading single multipliers right across the FPGA rather than positioning them in contiguous regions of space. This vastly increases the amount of wiring on the chip increasing the length of the critical path significantly.

However, even though the *clock speeds* of the design are decreased by the register placement scheme note that the actual *computation time* is generally faster as a result of register placement (see Figure 6.14). This is because the drop in clock-speed is compensated for by a corresponding drop in the number of cycles required for the computation.

An interesting observation is that although the Register Placement Analysis is architecture-neutral (e.g. applicable to both synchronous and asynchronous implementations) the effects that it would have in the asynchronous case are quite different. In the synchronous world Register Placement Analysis is guaranteed to save chip area and cycles (since we are removing registers). However, in turn this can increase the length of the critical path which may (as we have seen in this Figure 6.13) decrease the clock frequency. Although we hope that the loss of clock frequency may be compensated for by the decrease in the number of cycles leading to faster overall computation times this may not always be the case. For asynchronous implementations the result is more predictable: since one does not have to worry about critical paths a decrease in both time (since we remove the time-overhead incurred in latching data) and area (since we remove the circuitry required to perform the latching) is attained.

No. of Multipliers	Computation Time (μs)		% Saving
	With Reg Placement	Naive compilation	
1	0.171	0.205	16.6
2	0.097	0.121	19.3
3	0.068	0.066	−3.03
4	0.107	0.143	25.2
5	0.036	0.036	0

Fig. 6.14. Time taken for design to perform computation

6.6.2 Synchronous Timing Optimisations: Results

We tried compiling the SAFL code fragments of Figure 6.9 with and without the *control-join elimination* and *function call control signal* optimisations (see Section 6.5.2). For these programs we found that the optimisations only saved us a small percentage in chip area: between 0.8% and 1.0% of the FPGA. The savings are small because for each of these programs the control circuitry is only a small fraction of the circuit as a whole—the vast majority of area is taken up by combinatorial multipliers which are not affected by the optimisations.

In contrast, applying the same optimisations to the SAFL DES encrypter/decrypter of Section 10.2 results in more impressive savings. For the DES circuit, 156 control joins were optimised away leading to a 7.94% reduction in chip area. Here the saving is much larger because the control circuitry forms a greater percentage of the design.

Unlike the register placement optimisation, the optimisations based on synchronous timing analysis (see Section 6.5.2) cannot increase the length of the critical path[7]. Therefore the savings in area gained from these optimisations need not be traded against a possible decrease in clock speed.

Applying the Optimisations

Recall that the Register Placement optimisation is architecture-neutral. This means that it can always be applied to a whole SAFL program, regardless of the design style (e.g. synchronous or asynchronous) being targetted. In contrast the optimisations based on Synchronous Timing Analysis are architecture specific: they can only be applied if we are targetting synchronous hardware.

An interesting scenario is to consider what happens when we target GALS (Globally Asynchronous Locally Synchronous) hardware—see Section 5.3. In this case, as usual, we are able to apply our Register Placement optimisation. However, we are also able to optimise hardware in each clock domain separately by applying our Synchronous Timing optimisations on a *per clock-domain* basis.

6.7 Summary

In this chapter we presented two analyses:

- register Placement Analysis (Section 6.3),
- synchronous Timing Analysis (Section 6.5)

and three corresponding optimisations:

- removing unnecessary temporary registers,
- removing redundant arbitration (on top of arbiters already removed by Parallel Conflict Analysis—see Chapter 4),
- eliminating unnecessary control circuitry.

Obviously this is not an exhaustive survey. There are many more analyses and optimisations which could be employed to improve the efficiency of hardware generated from SAFL code. However, for reasons of time and space, we do not consider any further optimisations in this monograph.

Although we have presented the technical details and experimental results arising from a number of different analyses and optimisations, the primary motivation of this chapter is to justify our claim (first stated in Section 1.4) that:

> SAFL is designed specifically to support ... [global] analysis and optimisation

[7] It is possible that the optimisations may *decrease* the length of the critical path since control-join elimination removes OR gates from the control path. However we have found that this seldom happens in practice as the control circuitry is rarely on the critical path.

Global analyses such as these are not feasible in existing behavioural HDLs which employ structural blocks as their primary abstraction mechanism (see Section 3.3.4). In contrast SAFL is able to support global static analysis.

We feel that the concept of architecture-neutrality is an important one. Existing behavioural HDLs and their corresponding synthesis tools typically only focus on a single design style. If we are to move towards tools which can target different design styles[8] then architecture-neutral analyses allow one to migrate complex optimisation code into compiler phases which are executed *before* control is passed to one of the many back-ends. This simplifies the design and coding of each of the *multiple* backends at the cost of increasing the complexity of the *single* front-end.

[8] As it becomes increasingly difficult to distribute a single clock across large chips we believe that being able to target different design styles from a single high-level specification will become important (see Section 1.3 and the Semiconductor Industry Association (SIA) Roadmap [1]).

7

Dealing with I/O

Whilst SAFL is an excellent vehicle for high-level synthesis research we recognise that it is not expressive enough for industrial hardware description. In particular the facility for I/O is lacking and, in some circumstances, the "call and wait for result" interface provided by the function model is too restrictive. To address these issues we have developed a language, SAFL+, which extends SAFL with process calculus features including synchronous channels and channel-passing in the style of the π-calculus [100].

This chapter is structured as follows:

- We extend SAFL with synchronous channels and assignment and argue that the resulting combination of functional, concurrent and imperative styles is a powerful framework in which to describe a wide range of hardware designs (Section 7.1).
- Channel passing in the style of the π-calculus [100] is introduced. By parameterising functions over both data and channels the SAFL+ `fun` declaration becomes a powerful abstraction mechanism unifying a range of structuring techniques treated separately by existing HDLs (Section 7.1.3).
- We show how SAFL+ is implemented at the circuit-level (Section 7.2) and define the language formally by means of an operational semantics (Section 7.3).

7.1 SAFL+ Language Description

In this section we present the syntax of SAFL+ and informally describe its semantics. The language semantics are defined formally in Section 7.3.

SAFL+ is a concurrent, first-order, call-by-value language which, in the style of ML [101], supports a combination of functional and imperative programming. Function call arguments and `let`-definitions are evaluated in parallel as in SAFL; synchronous channels allow parallel threads to communicate with each other.

Function declarations take the form:

$$\texttt{fun}\ f(x_1, \ldots, x_k)\ [c_1, \ldots, c_j] = e \qquad (\text{where } k, j \geq 0)$$

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 113–127, 2004.

We make a syntactic distinction between arguments used to pass data, $x_1, \ldots, x_k$, and arguments used to pass channels $c_1, \ldots, c_j$. Iteration is provided in the form of self-tail-recursive calls. As with SAFL, general recursion is forbidden to permit static allocation of storage (see Section 3.3.2). Programs have a distinguished function, `main`, which represents an external world interface—at the hardware level it accepts values on an input port and may later produce a value on an output port.

We use slightly different notation in this chapter in order to be consistent with our published work on SAFL+ [131, 104]. In the following a ranges over primitive functions (such as `+`, `*` etc.), f ranges over user-defined functions and l ranges over record field labels. We use r for array variables, c for channel variables, x for other variables, and i for integer constants. The abstract syntax of SAFL+ programs, p, is presented in Figure 7.1.

$$
\begin{array}{lcll}
e & \leftarrow & x \mid i \mid () & \text{(Variable, Integer constant, Unit constant)} \\
 & \mid & \{l_1 = e_1, \ldots l_k = e_k\} \mid e.l & \text{(Record creation/selection)} \\
 & \mid & r[e] \mid r[e] := e & \text{(Array read/write)} \\
 & \mid & c? \mid c\,!\,e & \text{(Channel read/write)} \\
 & \mid & a(e_1, \ldots, e_k) & \text{(Call to primitive function)} \\
 & \mid & f(e_1, \ldots, e_k)[c_1, \ldots, c_j] & \text{(Call to user-defined function)} \\
 & \mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \text{(Conditional)} \\
 & \mid & \texttt{let}\ \vec{x} = \vec{e}\ \texttt{in}\ e_0 & \text{(Parallel let)} \\
 & \mid & \texttt{static}\ p\ \texttt{in}\ e & \text{(Local declarations)} \\
 & \mid & e \parallel e \mid e; e & \text{(Parallel/sequential composition)} \\
 & & & \\
d & \leftarrow & \texttt{fun}\ f(x_1, \ldots, x_n)[c_1, \ldots, c_n] = e & \text{(Function declaration)} \\
 & \mid & \texttt{channel}\ c & \text{(Channel declaration)} \\
 & \mid & \texttt{channel external}\ c & \text{(I/O Channel declaration)} \\
 & \mid & \texttt{array}\ [i]\ r & \text{(Array declaration)} \\
 & & & \\
p & \leftarrow & d \mid d\ p &
\end{array}
$$

Fig. 7.1. The abstract syntax of SAFL+ programs, p

Our existing compiler provides a number of simple syntactic sugarings on top of those already discussed in Section 3.2.4:

- The declaration `array [1] r` can be written `reg r`. When accessing arrays of unit size one writes `r` instead of `r[0]`.
- Functions without channel parameters can omit their square brackets completely (both in definition and calls).
- A `case`-statement is translated into nested conditionals in the usual way.

The `static` construct, which is used to introduce local definitions, is provided purely for syntactic convenience. It has no dynamic significance (and hence must

not be confused with the kind of *dynamic* channel-creation present in the π-calculus.) We note the similarity between SAFL+'s `static` construct and the C language's static *storage-class*.

7.1.1 Resource Awareness

We extend the concept of resource awareness first introduced in Section 3.3.2 to our extended language. As with SAFL, our approach is to model hardware as a fixed set of communicating and (possibly) shared resources. As can be seen from Figure 7.1, a program consists of a series of resource declarations. However, whereas SAFL only supports function resources, SAFL+ facilitates the declaration of three different types of resource, each of which addresses a key element of hardware design:

Resource type	Purpose	H/w Representation
Function	Computation	General Purpose Logic
Channel	Communication	Buses, Wires and Control Logic
Array	Storage	Memories or Registers

We say that SAFL+ is *resource-aware* since each declaration, d, (be it a function, channel or array declaration) corresponds to a *single* hardware block, H_d. Multiple references to d at the source-level (e.g. multiple calls to a function or multiple assignments to an array) correspond to the sharing of H_d at the circuit-level.

A call, $f(\vec{x})[\vec{c}]$, corresponds to: (i) acquiring mutually exclusive access to resource, H_f; (ii) passing data $\vec{x}$ and channel-parameters $\vec{c}$ into H_f; (iii) waiting for H_f to terminate; and (iv) latching[1] the result from H_f's shared output.

For a concrete example see the SAFL+ code fragment in Figure 7.3 which describes a lock shared between functions `f1` and `f2`. Synthesising this example leads to three resources: $H_{\texttt{f1}}$, $H_{\texttt{f2}}$ and $H_{\texttt{lock}}$. Note that $H_{\texttt{lock}}$ is shared between resources $H_{\texttt{f1}}$ and $H_{\texttt{f2}}$.

Resource-awareness means that, although a SAFL+ compiler is free to optimise the internals of `fun` definitions, it must respect the circuit structure specified by the programmer (i.e. one declaration = one hardware-level resource).

7.1.2 Channels and Channel Passing

SAFL+ provides synchronous channels to allow parallel threads to synchronise and transfer information. Channels can be used to transfer data locally within a function, or globally, between concurrently executing functions.

Our channels generalise Occam [76] and Handel-C [2] channels in a number of ways: SAFL+ channels can have any number of readers and writers, are bidirectional and can connect any number of parallel processes. As in the π-calculus, if there are multiple readers and multiple writers all wanting to communicate

[1] Register Placement Analysis (see Section 6.3) can be used to optimise these latches away under certain circumstances.

on the same channel then a single reader and a single writer are chosen non-deterministically.

At the hardware level a channel is implemented as a many-to-many communications bus supporting the atomic transfer of single values between readers and writers (see Section 7.2). No language-support is provided for *bus-transactions* (e.g. lock the bus for 20 cycles and write the following sequence of data values onto it). In Figure 7.3 a SAFL+ code fragment is presented which shows how such transactions can be implemented by using explicit locking.

Channels declared as `external` are used for I/O: writing to an external channel corresponds to an output action; reading an external channel corresponds to reading an input. There is no synchronisation on external channels although writes are guaranteed to occur under mutual exclusion. For example, for an external channel `c`, the only two possible output sequences occurring as a result of evaluating expression `(c!2 || c!3)` are $\langle 2, 3\rangle$ or $\langle 3, 2\rangle$. (See Section 7.3).

```
fun Accumulate(state) [c] =
  let val read_value = c?
   in if read_value=0 then state
                      else Accumulate(state+read_value)
  end

fun GenNumbers(state) [c] =
  c!state; if c=0 then () else GenNumbers(state-1)

fun sum(x) =
  static channel connect
  in GenNumbers(x) [connect] || Accumulate(0)[connect]
  end
```

Fig. 7.2. Illustrating Channel Passing in SAFL+

The code fragment of Figure 7.2 illustrates how channel passing is supported by SAFL+. Two resources parameterised over channel parameters are defined:

- `Accumulate` reads integers from a channel, returning their total when a 0 is read;
- `GenNumbers` writes a decreasing stream of integers to a channel, terminating when 0 is reached.

The function, `sum(x)` calculates the sum of the first `x` integers by composing the two resources in parallel and linking them with a common channel, `connect`. (Note that the parallel composition operator, `||`, waits for both its components to terminate before returning the value of the rightmost one.)

Channel parameters are not passed on recursive calls. Once a function resource, f, has been acquired by means of an *external* (i.e. non-recursive) call, $f(\vec{x})[\vec{c}]$, f's channel parameters remain bound to $\vec{c}$ until f terminates. See the operational semantics presented in Section 7.3 for a more precise description.

7.1.3 The Motivation for Channel Passing

By parameterising functions over both data and channel parameters, the SAFL+ `fun` definition becomes a powerful abstraction mechanism, encapsulating a wide range of structuring primitives treated separately in existing HDLs:

- Pure functions can be expressed by omitting channel parameters:
  `fun f(x,y) = ...`
- Structural-level blocks (cf. Verilog's `module` construct) can be expressed as non-terminating `fun` declarations parameterised over channels:
  `fun module() [in1,in2,out] = ...; module()`
- HardwareC-style `process` declarations can be expressed as non-terminating `fun` definitions (possibly without channel or data parameters):
  `fun process() = ...; process()`
- HardwareC-style `procedure`s can be expressed as `fun` declarations that return a unit result:
  `fun procedure(x,y) = ...; ()`

```
fun lock()[acquired, release] = acquired!(); release?

fun f1() = static channel go
                  channel done
           in (lock()[go,done] ||
               (go?;    (* code for f1's critical region *)
                done!()) )
           end

fun f2() = static channel go
                  channel done
           in (lock()[go,done] ||
                 (go?;   (* code for f2's critical region *)
                  done!()) )
           end
```

Fig. 7.3. Using SAFL+ to describe a lock explicitly

As well as unifying a number of common abstraction primitives, SAFL+ also supports a style of programming not exploited by existing HDLs. Recall the definition of `Accumulate` in Section 7.1.2. The `Accumulate` function can be seen as a hybrid between a structural-level block (since it is parameterised over a port, `c`) and a function (since it terminates, returning a result). More generally, by passing in locally defined channels, a caller, f, is able to synchronise and communicate with its callee, g, during g's execution. For example, consider the SAFL+ code in Figure 7.3 which declares a lock shared between functions `f1` and `f2`. The lock is used to enforce mutual exclusion between critical regions contained within the function bodies: The `lock` function is parameterised over

two channels: `acquired` is signalled as soon as `lock` starts executing, indicating to the caller that the lock has been acquired; `release` is used by the caller to signal that it has finished with the lock (at which point `lock` terminates). Recall that resource-awareness means that `lock` represents a single resource shared by functions `f1` and `f2`: the compiler ensures that only one caller can acquire it at a time. By passing in locally defined channels, functions `f1` and `f2` are able to communicate with `lock` during its execution. (Note that, since SAFL+ channels are bidirectional, we could use a single control channel to signal both acquisition and release requests; we use two channels merely for expository purposes.)

7.2 Translating SAFL+ to Hardware

In Chapter 5 we have already described how we translate the functional subset of SAFL+ into hardware. The basic principle involves translating each function definition into a single hardware block consisting of logic to serialise concurrent accesses and registers to latch arguments. Tail-recursive calls are translated into feedback loops at the circuit level.

Here we extend this by showing how the non-functional features (i.e. channels and arrays) can be integrated into our existing framework and compiled to synchronous hardware. As before, we adopt the graphical convention that thick lines represent data-wires and thin lines represent control signals.

A channel is translated into a shared bus surrounded with the necessary control logic to arbitrate between waiting readers and writers. Figure 7.4 shows channel control circuitry in a case where there are two readers and three writers. Since we are primarily targeting FPGAs we choose to multiplex data onto the bus rather than using tri-state buffers. To perform a read operation the reader signals its read-request and blocks until the corresponding read-acknowledge is signalled. We adopt the convention that the read-acknowledge line remains high for one cycle during which time the reader samples the data from the channel. To perform a write operation the writer places the data to be written onto a channel's data-input and signals the corresponding write-request line; the writer blocks until the corresponding write-acknowledge is signalled. Our current compiler synthesises static fixed-priority arbiters (see Section 5.13) to resolve multiple simultaneous read requests or multiple simultaneous write requests. However, since the SAFL+ semantics do not specify an arbitration policy, future compilers are free to exploit other selection mechanisms.

Our SAFL+ compiler performs a static flow-analysis to determine which *actual channels* (those bound directly by the `channel` construct) a given formal-channel-parameter may range over. This information enables the compiler to statically connect each channel operation (read or write) to every possible actual channel that it may need to access dynamically. At the circuit level channel values are represented as (small) integers which are passed as additional parameters on a function call.

The FLaSH intermediate code (Chapter 5) is augmented with READ and WRITE nodes representing channel operations. In cases where our flow-analysis

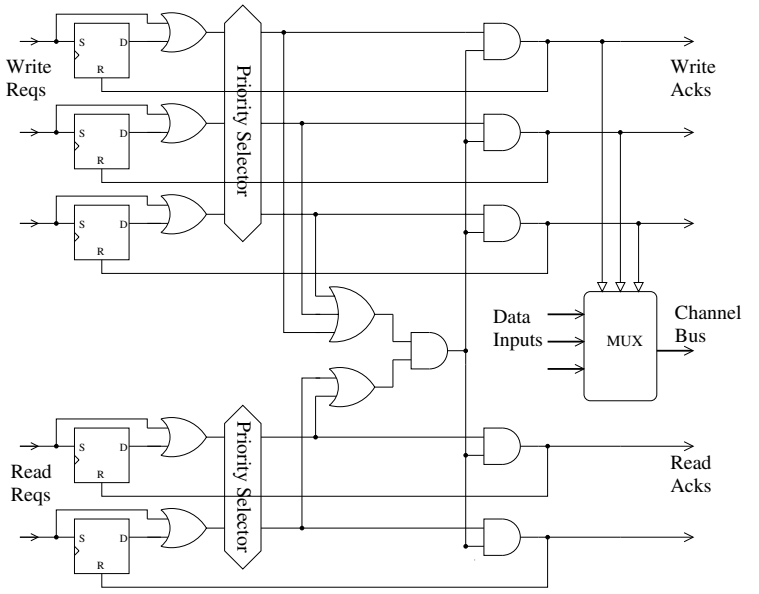


Fig. 7.4. A Channel Controller. The synchronous RS flip-flops (R-dominant) are used to latch pending requests (represented as 1-cycle pulses). Static fixed priority selectors are used to arbitrate between multiple requests. The three data-inputs are used by the three writers to put data onto the bus

detects that a channel operation may refer to a number of possible actual channels, multiplexers and demultiplexers are used to dynamically route to the appropriate channel. READ nodes have a control-input (used to signal the start of the operation), a control-output (used to signal the completion of the operation), a channel-select-input (used to select which actual channel to read from) and a data-output (the result of the read operation). Similarly WRITE nodes have a control-input, a control-output, a channel-select-input and a data-output. As in Chapter 5, our current compiler represents control events as 1-cycle pulses. Figure 7.5 shows (*i*) a READ node connected to three channels and (*ii*) a WRITE node connected to two channels. Each of the boxes labelled 'Chan' is a channel (as in Figure 7.4). Although each such channel may well have other readers/writers these are not shown in the figure. The data-wires labelled 'CB' are the channel busses, those labelled 'DI' are channels' data-inputs (multiplexed onto the channel busses—see Fig. 7.4). 'ChSel' is the channel-select-input. Note that (although not shown in this figure) channel busses may be shared among many readers. The dotted line represents the boundary between the resource performing the channel operation and the channel circuits themselves.

We extend the translation of `fun` declarations to include extra registers to latch channel-parameters. At the circuit-level channel-parameters are fed into the select lines of the multiplexers and demultiplexers seen in Figure 7.5. In this

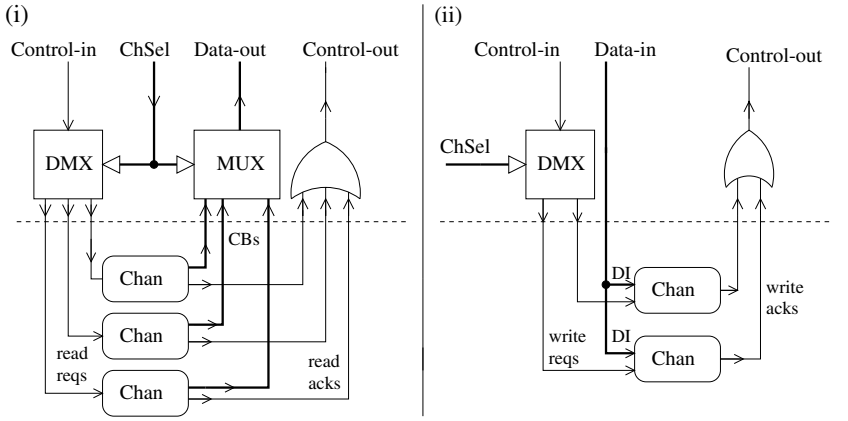


Fig. 7.5. (*i*) A READ node connected to three channels; (*ii*) A WRITE node connected to two channels. The component marked DMX is a demultiplexer which routes the control signal to one of the three channels depending on the value of its select input (ChSel)

example 'ChSel' would be read directly from the registers storing the enclosing function's channel-parameters.

Arrays are represented as RAMs wrapped up in the necessary logic to arbitrate between multiple concurrent accesses. Our compiler translates array declarations:

```
array [i] r
```

into SAFL+ function definitions with signature:

```
fun r (addr:int, data:int, wr_select:bit) : int
```

Calling `r` always returns the value stored at memory location `addr`. If `wr_select` is 1 then location `addr` is updated to contain `data`. Hence array assignments,

```
r[e1] := e2
```

are translated into function calls of the form

```
r(1,e2,e1)
```

and (similarly) array accesses, `r[e]`, are translated into calls of the form `r(0,0,e)`. Treating arrays as SAFL+ functions in this way allows us to use the compiler's existing machinery to synthesise the necessary logic to serialise concurrent accesses to the array and latch address lines. The compiler automatically generates the body of `r`, which consists solely of RAM.

7.2.1 Extending Analyses from SAFL to SAFL+

We have just seen how `array`s, `reg`s and assignment are treated as function calls. We have also already shown (in Section 3.2.4) how SAFL+'s sequential "`;`" and

parallel "||" composition operators can be translated into plain SAFL. Thus, when extending our analyses of Chapters 4 and 6.1, we only need to consider how to deal with channel reads and writes.

The current version of our Register Placement Optimisation treats channels in a naïve manner, inserting a permanisor after every read operation. In fact, given the circuit diagram of a channel shown in Figure 7.4 there is often little alternative, since the output of the channel is only guaranteed to be valid for a single cycle. As an architecture-specific analysis we can optimise away permanisors on channel reads when we can infer that the result is only required for a single cycle (e.g. because it is immediately fed back into the enclosing function's argument registers as part of a tail-recursive call).

$$\mathcal{C}[\![c!e]\!] = \mathcal{C}[\![e]\!]$$
$$\mathcal{C}[\![e?]\!] = \mathcal{C}[\![e]\!]$$

$$\mathcal{A}[\![c!e]\!] = \mathcal{A}[\![e]\!]$$
$$\mathcal{A}[\![e?]\!] = \mathcal{A}[\![e]\!]$$

Fig. 7.6. Extending PCA to deal with channel reads and writes

Parallel Conflict Analysis, which detects which function calls to a shared functional-unit may occur simultaneously, can be trivially extended to deal with channel read/write constructs by augmenting it with the equations shown in Figure 7.6.

7.3 Operational Semantics for SAFL+

In this section we define the meaning of the SAFL+ language formally through an operational semantics. Although, at first sight, the semantics may seem theoretical and far-removed from hardware-implementation we argue that this is not the case. It is worth pointing out that many of the symbols in Figure 7.10 have a direct correspondence to circuit-level components. For example, *channel resources*, $\langle v \rangle_c$ (see below), represent channel controller circuits (as shown in Figure 7.4) and the (*Call*) rule (see Figure 7.10) corresponds directly to transferring data into the callee's argument registers (circuits corresponding to this can be seen in Section 5.10).

A SAFL+ program consists of a series of function definitions of the form:

$$\texttt{fun}\ f\ (x_1, \ldots, x_k)\ [c_1, \ldots, c_j] = b_f$$

We write b_f for the body of function, f, $x_1, \ldots, x_k$ for formal parameters and $c_1, \ldots, c_j$ for channel parameters. Again, we write $\vec{x}$ to mean $x_1, \ldots, x_k$ and, similarly, $\vec{c}$ to mean $c_1, \ldots, c_j$.

Due to the static nature of SAFL+, we can simplify matters by assuming that: (*i*) SAFL+ programs have been α-converted to make all variable names distinct; and (*ii*) scope-flattening has been performed, bringing local declarations to the top level and eliminating `static` statements. (Note that bringing a locally defined function to the top level may require extra arguments to be added to the function in order to pass in values for its free variables.)

We give the semantics by describing how one *program state*, P, evolves into another, say Q, by means of a *transition*: $P \xrightarrow{\alpha} Q$, where α represents an optional I/O action taking one of the following forms:

$\bar{\mathbf{c}}\langle v\rangle$	Output v on external channel $\mathbf{c}$
$\mathbf{c}(v)$	Read a value v from external channel $\mathbf{c}$
$go(\vec{v})$	Pass parameters $\vec{v}$ into the `main` function
$done(v)$	Read result v from the `main` function

Note that we use a bold-face $\mathbf{c}$ to range over external channels (in contrast to c, which ranges over non-external channels).

A program state consists of a parallel composition of *function resources*, *channel resources* and *array resources* (see Figure 7.7). Our presentation borrows notation and ideas from Marlow *et al* [94].

$$
\begin{array}{rcll}
P & \leftarrow & (\!|e|\!)_f & \text{(busy function)} \\
 & | & \mathbb{O}_f & \text{(available function)} \\
 & | & \langle\rangle_c & \text{(empty channel)} \\
 & | & \langle v\rangle_c & \text{(full channel, holding value } v\text{)} \\
 & | & \langle \text{Ack}\rangle_c & \text{(channel in acknowledge state)} \\
 & | & P \mid P & \text{(parallel composition)} \\
 & & & \\
e & \leftarrow & v & \text{(value)} \\
 & | & \mathcal{W}_g & \text{(awaiting result from } g\text{)} \\
 & | & \{l_1 = e, \ldots, l_k = e\} \quad | \quad e.l & \text{(as in Fig. 7.1)} \\
 & | & x \quad | \quad f(e,\ldots,e)[c_1,\ldots,c_n] \quad | \quad a(e,\ldots,e) & \ldots \\
 & | & \texttt{let } (x_1,\ldots,x_k) = (e,\ldots,e) \texttt{ in } e & \ldots \\
 & | & \texttt{if } e \texttt{ then } e \texttt{ else } e \quad | \quad c\,!\,e \quad | \quad c? \quad | \quad e \parallel e & \ldots \\
 & | & e;\, e \quad | \quad r[e] := e \quad | \quad r[e] & \text{(as in Fig. 7.1)} \\
 & & & \\
v & \leftarrow & i & \text{(integer, } i \in \mathbb{N}\text{)} \\
 & | & () & \text{(unit)}
\end{array}
$$

Fig. 7.7. The Syntax of Program States, P, Evaluation States, e, and values, v

Each non-external channel declaration, `channel` c, corresponds to a channel resource. When an empty channel resource (written $\langle\rangle_c$) reacts with a waiting writer a value, v, is transferred and c becomes full (written $\langle v\rangle_c$). On reacting with a waiting reader, the value is consumed and the c enters an acknowledge state (written $\langle \text{Ack}\rangle_c$). The Ack interacts with the writer, notifying it that communication has taken place and returning c to the empty state, $\langle\rangle_c$. The explicit

use of Ack models the synchronous nature of SAFL+ channels ensuring that a writer is blocked until its data has been consumed by a reader.

Array resources, $[\mathcal{S}_i]_r$, correspond to array declarations, `array [i] r`. The contents of the array, $\mathcal{S}_i$, is a function mapping indexes $0 \ldots (i-1)$ onto values. We write $\mathcal{S}_i\{j \mapsto v\}$ to denote the function which is as $\mathcal{S}_i$ but maps index j onto value v. Accessing elements outside the bounds of an array leads to undefined behaviour. To reflect this we define $\mathcal{S}_i(j)$ to be an undefined value if $j \geq i$. Furthermore if $j \geq i$ then $\mathcal{S}_i\{j \mapsto v\}$ represents an undefined state mapping indexes $0 \ldots (i-1)$ onto undefined values.

Each SAFL+ function declaration, `fun` f $(\vec{x})$ $[\vec{c}] = b_f$, is represented by a function resource. At any given time a function resource may be *busy* (performing a computation) or *available* (waiting to perform a computation). An available function resource, f, is written $\mathbb{0}_f$, signifying that f is not in use; a busy function resource takes the form $(\!|e|\!)_f$ signifying that f is currently in *evaluation state* e. The syntax of evaluation states (see Figure 7.7) is essentially the same as the syntax of SAFL+ expressions augmented with the $\mathcal{W}_g$ construct which represents waiting for a result from function resource g.

As with the Chemical Abstract Machine [22] program states can be viewed as a "solution" of reacting resources. We formalise this notion in the standard way by defining structural congruence, $\equiv$, to be the least congruence which satisfies the (*Comm*) and (*Assoc*) equations of Figure 7.8. Rule (*Par*) allows transitions within parallel compositions and (*Equiv*) makes it possible to use the structural congruence relation to bring different parts of the program state together.

SAFL+ is an implicitly parallel language—an expression may contain a number of sub-expressions which can be evaluated concurrently. To formalise this notion we use a *context*, $\mathbb{E}$, to highlight the parts of an evaluation state which can be evaluated concurrently (see Figure 7.9). Intuitively a context, $\mathbb{E}[\cdot]$, is an evaluation state with a hole $[\cdot]$ into which we can insert an evaluation state, e, to derive a new evaluation state $\mathbb{E}[e]$.

$$\begin{array}{rcll} P \mid Q & \equiv & Q \mid P & (\mathit{Comm}) \\ P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\mathit{Assoc}) \end{array}$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \quad (\mathit{Par}) \qquad \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \quad (\mathit{Equiv})$$

Fig. 7.8. Structural congruence and structural transitions

A useful mental model is to consider a frontier of evaluation which is defined by $\mathbb{E}$ and advanced by applying the transition rules (see Section 7.3.1 and Figure 7.10).

$$
\begin{array}{l}
\mathbb{E} \leftarrow [\cdot] \\
\quad \mid f(\mathbb{E}, e_2, \ldots, e_k) \mid \ldots \mid f(e_1, \ldots, e_{k-1}, \mathbb{E}) \\
\quad \mid a(\mathbb{E}, e_2, \ldots, e_k) \mid \ldots \mid a(e_1, \ldots, e_{k-1}, \mathbb{E}) \\
\quad \mid \{n_1 = \mathbb{E},\ n_2 = e_2,\ \ldots,\ n_k = e_k\} \\
\qquad\qquad \ldots \\
\quad \mid \{n_1 = e_1,\ \ldots,\ n_{k-1} = e_{k-1},\ n_k = \mathbb{E}\} \\
\quad \mid \texttt{let}\ (x_1, \ldots, x_k) = (\mathbb{E}, e_2, \ldots, e_k)\ \texttt{in}\ e \\
\qquad\qquad \ldots \\
\quad \mid \texttt{let}\ (x_1, \ldots, x_k) = (e_1, \ldots, e_{k-1}, \mathbb{E})\ \texttt{in}\ e \\
\quad \mid r[\,\mathbb{E}\,] \mid r[\,\mathbb{E}\,] := e \mid r[\,e\,] := \mathbb{E} \mid \mathbb{E}.l \\
\quad \mid \texttt{if}\ \mathbb{E}\ \texttt{then}\ e\ \texttt{else}\ e \mid \mathbb{E};\ e \mid \mathbb{E} \parallel e \mid e \parallel \mathbb{E} \mid c\,!\,\mathbb{E}
\end{array}
$$

Fig. 7.9. A context, $\mathbb{E}$, defining which sub-expressions may be evaluated in parallel

7.3.1 Transition Rules

For clarity, we present the transition rules for SAFL+ in two parts: Figure 7.10(a) gives the rules for SAFL+ without channel passing. Figure 7.10(b) and Section 7.3.2 explain how the rules can be modified to handle channel passing.

Substitution of values, $v_1 \ldots v_n$, for variables, $x_1 \ldots x_n$ in an evaluation state, e, is written, $\{v_1/x_1, \ldots, v_n/x_n\}e$, and for convenience abbreviated to $\{\vec{v}/\vec{x}\}e$.

The rules in Figure 7.10 are divided into six categories:

- (*Call*) and (*Return*) deal with interaction between separate functional resources.
- (*Ch-Write*), (*Ch-Read*) and (*Ch-Ack*) model communication over synchronous channels.
- (*Input*) and (*Output*) deal with I/O through reading and writing external channels.
- (*Ar-Write*) and (*Ar-Read*) handle writing and reading of array resources respectively.
- (*Start*) and (*End*) correspond to externally invoking and receiving the result from the `main` function.
- The remainder of the rules deal with local computation within a function resource.

Note that the left hand side of the (*Tail-Rec*) rule is not enclosed in a context. This reflects the fact that tail recursive calls cannot occur in parallel with any other expressions; hence a context is unnecessary.

7.3.2 Semantics for Channel Passing

To deal with channel passing, function resources need to store the channel parameters passed from an external call. We use the notation $(\!|\ \cdot\ |\!)^{\vec{c}}_f$ to represent

(a) Rules for SAFL+ without channel passing:

$$(\!|\mathbb{E}[g(v_1,\ldots,v_k)]|\!)_f \mid \mathbb{O}_g \longrightarrow (\!|\mathbb{E}[\mathcal{W}_g]|\!)_f \mid (\!|\{\vec{v}/\vec{x}\}b_g|\!)_g \quad (\mathit{Call})$$
$$\text{providing } f \neq g$$

$$(\!|v|\!)_f \mid (\!|\mathbb{E}[\mathcal{W}_f]|\!)_g \longrightarrow \mathbb{O}_f \mid (\!|\mathbb{E}[v]|\!)_g \quad (\mathit{Return})$$
$$(\!|\mathbb{E}[c\ !\ v]|\!)_f \mid \langle\rangle_c \longrightarrow (\!|\mathbb{E}[\mathcal{W}_c]|\!)_f \mid \langle v\rangle_c \quad (\mathit{Ch\text{-}Write})$$
$$(\!|\mathbb{E}[c?]|\!)_f \mid \langle v\rangle_c \longrightarrow (\!|\mathbb{E}[v]|\!)_f \mid \langle \mathrm{Ack}\rangle_c \quad (\mathit{Ch\text{-}Read})$$
$$(\!|\mathbb{E}[\mathcal{W}_c]|\!)_f \mid \langle \mathrm{Ack}\rangle_c \longrightarrow (\!|\mathbb{E}[()]|\!)_f \mid \langle\rangle_c \quad (\mathit{Ch\text{-}Ack})$$

$$(\!|\mathbb{E}[\mathbf{c}\ !\ v]|\!)_f \xrightarrow{\bar{\mathbf{c}}\langle v\rangle} (\!|\mathbb{E}[()]|\!)_f \quad (\mathit{Output})$$
$$(\!|\mathbb{E}[\mathbf{c}?]|\!)_f \xrightarrow{\mathbf{c}(v)} (\!|\mathbb{E}[v]|\!)_f \quad (\mathit{Input})$$

$$(\!|\mathbb{E}[r[v_1]\ :=\ v_2]|\!)_f \mid [\mathcal{S}_i]_r \longrightarrow (\!|\mathbb{E}[()]|\!)_f \mid [\mathcal{S}_i\{v_1 \mapsto v_2\}]_r \quad (\mathit{Ar\text{-}Write})$$
$$(\!|\mathbb{E}[r[v]]|\!)_f \mid [\mathcal{S}_i]_r \longrightarrow (\!|\mathbb{E}[\mathcal{S}_i(v)]|\!)_f \mid [\mathcal{S}_i]_r \quad (\mathit{Ar\text{-}Read})$$

$$\mathbb{O}_{\mathrm{main}} \xrightarrow{go(\vec{v})} (\!|\{\vec{v}/\vec{x}\}b_{\mathrm{main}}|\!)_{\mathrm{main}} \quad (\mathit{Start})$$
$$(\!|v|\!)_{\mathrm{main}} \xrightarrow{done(v)} \mathbb{O}_{\mathrm{main}} \quad (\mathit{End})$$

$$(\!|\mathbb{E}[a(v_1,ldots,v_k)]|\!)_f \longrightarrow (\!|\mathbb{E}[v]|\!)_f \quad (\mathit{PrimOp})$$
$$\text{where } v = a(v_1,\ldots,v_k)$$

$$(\!|\mathbb{E}[\{\ldots,l=v,\ldots\}.l]|\!)_f \longrightarrow (\!|\mathbb{E}[v]|\!)_f \quad (\mathit{RecSelect})$$
$$(\!|\mathbb{E}[\texttt{if } 0 \texttt{ then } e_1 \texttt{ else } e_2]|\!)_f \longrightarrow (\!|\mathbb{E}[e_2]|\!)_f \quad (\mathit{CFalse})$$
$$(\!|\mathbb{E}[\texttt{if } n \texttt{ then } e_1 \texttt{ else } e_2]|\!)_f \longrightarrow (\!|\mathbb{E}[e_1]|\!)_f \quad (\mathit{CTrue})$$
$$\text{providing } n \neq 0$$

$$(\!|\mathbb{E}[\texttt{let } \vec{x} = \vec{v} \texttt{ in } e]|\!)_f \longrightarrow (\!|\mathbb{E}[\{\vec{v}/\vec{x}\}e]|\!)_f \quad (\mathit{Let})$$
$$(\!|\mathbb{E}[v; e]|\!)_f \longrightarrow (\!|\mathbb{E}[e]|\!)_f \quad (\mathit{Seq})$$
$$(\!|\mathbb{E}[v_1 \parallel v_2]|\!)_f \longrightarrow (\!|\mathbb{E}[v_2]|\!)_f \quad (\mathit{Par})$$
$$(\!|f(v_1,\ldots,v_k)|\!)_f \longrightarrow (\!|\{\vec{v}/\vec{x}\}b_f|\!)_f \quad (\mathit{Tail\text{-}Rec})$$

(b) Modifications for Channel Passing:

$$(\!|\mathbb{E}[g(v_1,\ldots,v_k)[d_1,\ldots,d_j]]|\!)^{\vec{c'}}_f \mid \mathbb{O}_g \longrightarrow (\!|\mathbb{E}[\mathcal{W}_g]|\!)^{\vec{c'}}_f \mid (\!|\{\vec{d}/\vec{c},\vec{v}/\vec{x}\}b_g|\!)^{\vec{d}}_g \quad (\mathit{Call})$$
$$(\!|v|\!)^{\vec{c}}_f \mid (\!|\mathbb{E}[\mathcal{W}_f]|\!)^{\vec{d}}_g \longrightarrow \mathbb{O}_f \mid (\!|\mathbb{E}[v]|\!)^{\vec{d}}_g \quad (\mathit{Ret})$$
$$(\!|f(v_1,\ldots,v_k)|\!)^{\vec{c'}}_f \longrightarrow (\!|\{\vec{c'}/\vec{c},\ \vec{v}/\vec{x}\}b_f|\!)^{\vec{c'}}_f \quad (\mathit{Tail\text{-}Rec})$$

Fig. 7.10. Transition Rules for SAFL+

a function resource which has been called with actual channel parameters, $\vec{c}$. For convenience we allow ourselves to omit the channel parameters from a rule, defining $(\!|e_1|\!)_f \rightarrow (\!|e_2|\!)_g$ to mean $(\!|e_1|\!)_f^{\vec{c}} \rightarrow (\!|e_2|\!)_g^{\vec{c}}$. Under this convention the only modifications required for channel passing are to the (*Call*), (*Ret*) and (*Tail-Rec*) rules—see Figure 7.10(b).

7.3.3 Non-determinism

There are three sources of non-determinism in SAFL+ specifications. Firstly, when expressions are composed in parallel, no order of evaluation is specified. Thus if two parallel expressions have conflicting side-effects then non-determinism is introduced. For example, `(x := 3 || x := 4)` may terminate in a state in which `x` is *either* 3 *or* 4. Secondly, as in the π-calculus, channels with multiple readers and writers select one reader and one writer non-deterministically. For example `(c!2 || c!5 || (c? - c?))` could evaluate to *either* -3 *or* 3. Finally, reading or writing elements outside the bounds of an array leads to undefined behaviour. Recall that $S_i(j)$ returns a random value if $j \geq i$ and, similarly, assigning to an out-of-bounds element corrupts the entire array.

Although we could make SAFL+ completely deterministic we choose not to for the following reasons:

- An unspecified evaluation order for function calls and `let`-definitions allows more freedom for the compiler to exploit parallelism, leading to the generation of more efficient hardware.
- Imposing Occam-style restrictions on SAFL+ channels (i.e. unidirectional, connecting exactly two processes) would reduce the expressivity of SAFL+. To see an example of this consider the problem of merging data from two separate channels onto a single channel. Since SAFL+ does not provide an explicit non-deterministic choice operator (cf. Occam's `ALT` construct), the only way to represent such a system is to exploit a multiple-writer, single-reader channel.
- Array bounds checking incurs a serious penalty since every array access requires a comparison. Not only would this reduce the performance of the generated hardware, but the extra comparators required may significantly increase the area (gate-count) of the circuit. In general we feel that this is unacceptable.

7.4 Summary

In this chapter we have described how the SAFL language, and its associated compiler, can be extended with state, synchronous channels and π-calculus-style channel passing, thus extending the IO capabilities of the language significantly.

By allowing functions to be parameterised over channels we believe that the resulting abstraction mechanism offers an elegant compromise between:

1. the high-level properties of functions; and
2. the flexibility of structural blocks (cf. Verilog `modules`).

We justify this claim by observing that, as with structural blocks, a programmer is now able to parameterise blocks of hardware over input and output ports. However, unlike structural blocks, SAFL+ functions still abstract data-flow and control-flow. Thus the global analyses that we advocate in Chapters 4 and 6 are still applicable.

8

Combining Behaviour and Structure

A contributing factor to the success of Verilog and VHDL is their support for *both* behavioural *and* structural-level design. The ability to combine behavioural and structural primitives in a single specification offers engineers a powerful framework: when the precise low-level details of a component are not critical, behavioural constructs can be used; for components where finer-grained control is required, structural constructs can be used[1]. However, the flip-side is that by supporting multiple levels of abstraction both Verilog and VHDL are very large languages which are difficult to analyse, transform and reason about.

Resource awareness allows SAFL to describe the *system-level* structure of a design by mapping `fun` declarations to circuit-level functional units. In contrast, systems such as μFP and Lava offer much finer-grained control over circuit structure, taking logic-gates (rather than function definitions) as their structural primitives. In this chapter we present a single, pure-functional framework which integrates Lava-style structural expansion with SAFL. We illustrate this technique with a realistic example in Chapter 10 where it is used in the SAFL specification of a fully functional DES encryption/decryption circuit.

8.1 Motivation and Related Work

In Chapter 2 we observed that there is a large body of work on using functional languages to describe hardware at the structural level. Notable systems in this area include μFP [132], Hydra [109], Hawk [96] and Lava [25]. Recall that the central idea behind each of these systems is to use the powerful features found in existing functional languages (e.g. higher-order functions, polymorphism and lazy evaluation) to build netlists from simple primitives. These primitives can be given different semantic interpretations allowing, for example, the same specification to be either simulated or translated into a netlist. However, whilst this technique is obviously appealing, there are problems involved in generating netlists for circuits which contain feedback loops. The difficulty

[1] Note the analogy with embedding assembly code in a higher-level software language.

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 129–139, 2004.

is that, in a pure functional language, a cyclic circuit (expressed as a series of mutually recursive equations) naturally evaluates to an infinite tree preventing the netlist translation phase from terminating.

A number of solutions to this problem have been proposed: O'Donnell advocates the explicit tagging of components at the source-level [110]. In this system the programmer is responsible for labelling distinct components of a circuit with unique values. Whilst this allows a pure functional graph traversal algorithm to detect cycles trivially (by maintaining a list of tags which have already been seen) it imposes an extra burden on the programmer and significantly increases potential for manual error (since it is the programmer's job to ensure that distinct components have unique tags). Lava [25] also uses tagging to identify cycles, but employs a *state monad* [141] to generate fresh tags automatically. Although this neatly abstracts the low-level tagging details from the designer, Claessen and Sands argue that the resulting style of programming is "unnatural" and "inconvenient" [37]. In the same paper, Claessen and Sands propose another solution which involves augmenting Haskell (the functional language in which Lava is embedded) with immutable references which support a test for equality. This extension makes graph sharing observable at the source-level but, although it is shown that many useful laws still hold, full equational reasoning is no longer possible—for example, β-reduction no longer preserves equality.

In this chapter we present an alternative approach: a *single*, pure-functional framework in which we can describe hardware at both structural and behavioural levels of abstraction. At the structural level (where recursion is realised as static expansion) circuits are restricted to being acyclic; these acyclic circuit fragments are then composed at the SAFL-level (where tail-recursion corresponds to feedback loops in the generated hardware). As well as gaining the engineering benefits of a system capable of structural and behavioural design (see above) we also eliminate the observable sharing problem. Since cycles are not permitted at the structural level we do not have to worry about infinite loops being statically expanded. Conversely, since feedback loops are represented as tail-recursive calls at the SAFL-level there is no need to introduce impure language features.

8.2 Embedding Structural Expansion in SAFL

We have developed our own system for structural hardware description which we refer to as Magma[2]. Section 8.2.1 describes the details of the Magma system. In Section 8.2.2 we show how Magma is integrated with SAFL.

8.2.1 Building Combinatorial Hardware in Magma

An argument in favour of Lava, Hydra and other similar systems, is that since they are embedded in existing functional languages they are able to leverage existing tools and compilers. Furthermore, use of non-standard interpretation

[2] As it is a restricted form of Lava.

of basis functions means that the *same* compiler can be used to perform both hardware simulation and synthesis. These compelling benefits lead us to adopt a similar approach. However, in contrast to Lava, which is embedded in Haskell [3], we choose to embed Magma in ML [101]. The choice of ML is fitting for two main reasons: firstly, since we only wish to describe acyclic circuits, ML's strict evaluation is appropriate for both simulation and synthesis interpretations; secondly, since SAFL also borrows much of its syntax and semantics from ML, both Magma and SAFL share similar conventions (an important consideration when we are dealing with specifications containing a mixture of both Magma and SAFL).

An ML Module System Primer

In order to understand the workings of Magma some familiarity with the ML module system is required. Whilst we do not describe the full details of the ML-module system here, this section is sufficient to allow readers unfamiliar the module system to understand the remainder of this chapter. For more information the reader is referred to an in-depth survey [120].

The basic element of ML's module system is the `structure`. The structure provides a way of packaging both type and value (including function) definitions into a single entity. An important feature of structures is that they provide a hierarchical name-space. For example, if a function, f, is defined in a structure $\mathcal{S}$ we refer to it as $\mathcal{S}.f$.

An ML `signature` provides a mechanism to specify interfaces. A signature contains a set of name and type-declarations. One can use a signature to constrain a structure using the "`:`" operator. Only values whose types are explicitly declared in the constraining signature are visible outside the constrained structure.

Finally, the ML module system provides `functor`s. A functor is essentially a parameterised structure, dependent on another structure which is later passed into it. For example, consider the following (contrived) code fragment which defines a functor, `FTR`, parameterised over a structure, `S` (where `S` is constrained by signature, `SSIG`):

```
functor FTR(S:SSIG) =
  struct
     val a = S.f(3)
  end
```

Passing a structure, `T`, into `FTR` yields a new structure containing a single item, `a`, which has the value `T.f(3)`. Magma makes use of `functor`s to parameterise hardware specifications over interpretations of their basis functions. This provides a convenient way of using the same code for both simulation and synthesis (see below).

Specifying Hardware in Magma

The Magma system essentially consists of a library of ML code. A signature called `BASIS` is provided which declares the types of supported basis functions (see Figure 8.1). Values `b0` and `b1` correspond to logic-0 (false) and logic-1 (true) respectively. Functions `orb`, `andb`, `notb` and `xorb` correspond to logic functions *or*, *and*, *not* and *xor*. Following the ideas first presented in Section 2.3.2 two structures which implement `BASIS` are provided:

- `SimulateBasis` provides a simulation interpretation. We implement `bit`s as boolean values; functions `orb`, `andb` etc. have their usual boolean interpretations.
- `SynthesisBasis` provides a synthesis interpretation. We implement `bit`s as strings representing names of wires in a net-list. Functions `orb`, `andb` etc. take input wires as arguments and return a (fresh) output wire. Calling one of the basis functions results in its netlist declaration being written to the selected output stream as a side-effect. For example, if the result of calling `andb` with string arguments "`in_wire1`" and "`in_wire2`" is the string "`out_wire`" then the following is output to `StdOut`:

```
and(out_wire,in_wire1,in_wire2);
```

```
signature BASIS =
    sig
        type bit
        val b0   : bit
        val b1   : bit
        val orb  : bit * bit -> bit
        val andb : bit * bit -> bit
        val notb : bit * bit
        val xorb : bit * bit -> bit
    end
```

Fig. 8.1. The definition of the `BASIS` signature (from the Magma library)

Figure 8.2 shows a Magma specification of a ripple-adder. As with all Magma programs, the main body of code is contained within an ML `functor`. This provides a convenient abstraction, allowing us to parameterise a design over its basis functions. By passing in the structure `SimulateBasis` (see above) we are able to instantiate a copy of the design for simulation purposes; similarly, by passing in `SynthesisBasis` we instantiate a version of the design which, when executed, outputs its netlist. The signature `RP_ADD` is used to specify the type of the `ripple_add` function. Using this signature to constrain the `RippleAdder` functor also means that *only* the `ripple_add` function is externally visible; the functions `carry_chain` and `adder` can only be accessed from within the functor. The use of signatures to specify interfaces in this way is not compulsory but, for the usual software-engineering reasons, it is recommended.

```
signature RP_ADD =
  sig
      type bit
      val ripple_add : (bit list * bit list) -> bit list
  end

functor RippleAdder (B:BASIS):RP_ADD =
      struct

         type bit=B.bit
         fun adder (x,y,c_in) =
             (B.xorb(c_in, B.xorb(x,y)),
              B.orb( B.orb( B.andb (x,y), B.andb(x,c_in)),
                            B.andb(y,c_in)))

         fun carry_chain f _ ([],[]) = []
           | carry_chain f c_in (x::xs,y::ys) =
                let val (res_bit, c_out) = f (x,y,c_in)
                 in res_bit::(carry_chain f c_out (xs,ys))
                end

         val ripple_add = carry_chain adder B.b0
      end
```

Fig. 8.2. A simple ripple-adder described in Magma

Let us imagine that a designer has just written the ripple-adder specification shown in Figure 8.2 and now wants to test it. This can be done by instantiating a simulation version of the design in an interactive ML session:

```
- structure SimulateAdder = RippleAdder (SimulationBasis);
```

The adder can now be tested by passing in arguments (a tuple of `bit` lists) and examining the result. For example:

```
- SimulateAdder.ripple_add ([b1,b0,b0,b1,b1,b1],
                            [b0,b1,b1,b0,b1,b1])
val it = [b1,b1,b1,b1,b0,b1]  : SimulateAdder.bit list
```

Let us now imagine that the net-list corresponding to the rippler-adder is required. We start by instantiating a synthesis version of the design:

```
- structure SynthesiseAdder = RippleAdder (SynthesisBasis);
```

If we pass in lists of input wires as arguments, the `ripple_add` function prints its netlist to the screen and returns a list of output wires:

```
- SynthesiseAdder.ripple_add (Magma.new_bus 5,
                              Magma.new_bus 5)
xor(w_1,w_45,w_46);
xor(w_2,w_1,w_44);
```

```
...
and(w_149,w_55,w_103);
val it = ["w_149","w_150","w_151","w_152","w_153"]
```

The function `new_bus`, part of the Magma library, is used to generate a bus of given width (represented as a list of wires).

8.2.2 Integrating SAFL and Magma

Our approach to integrating Magma and SAFL involves using delimiters to embed Magma code fragments inside SAFL programs. At compile time the embedded Magma is synthesised and the resulting netlist is incorporated into the generated circuit (see Figure 8.3). This technique was partly inspired by web-scripting frameworks such as ASP and PHP [4] which can be embedded in HTML documents. (When a dynamic web-page is fetched the ASP or PHP code is executed generating HTML which is returned to the client.) To highlight this analogy we use ASP-style delimiters "`<%`" and "`%>`" to mark the start and end points of Magma code fragments. Our compiler performs type checking across the SAFL-Magma boundary, ensuring the validity of the final design.

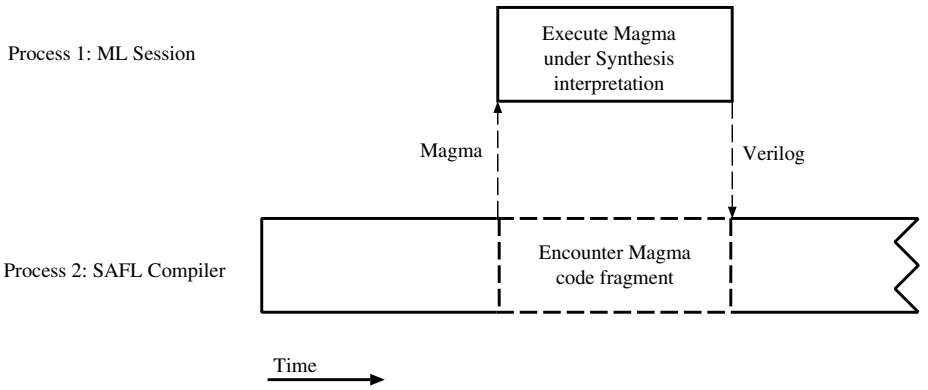


Fig. 8.3. A diagrammatic view of the steps involved in compiling a SAFL/Magma specification

The SAFL parser is extended to allow a special type of Magma code fragment at the beginning of a specification. This initial Magma fragment, which is referred to as the *library block*, contains an ML functor called `Magma_Code`. Functions within `Magma_Code` can be called from other Magma fragments in the remainder of the specification. Figure 8.4 illustrates these points with a simple example in which the Magma ripple adder (initially defined in Figure 8.2) is invoked from a SAFL specification. Although the programmer could have just written "`+`" in this instance, this would leave the compiler free to implement the adder with a design of its choosing. The key point is that Magma gives the designer greater control

```
(* Magma library block containing Magma_Code functor: *)

<%
  signature RP_ADD =
     ... (* as in Figure 2 *)

  functor Magma_Code (B:BASIS):RP_ADD =
     ... (* as RippleAdder functor in Figure 2 *)
%>

fun mult(x, y, acc) =
  if (x=0 | y=0) then acc
    else mult(x<<1, y>>1,
              if y[0:0] then <% ripple_add %>(acc,x) else acc)
```

Fig. 8.4. A simple example of integrating Magma and SAFL into a single specification

over the design: the generated adder will have the precise structural configuration specified in Figure 8.2. The details of the SAFL-Magma integration are discussed later in this section; for now it suffices to observe that Magma fragments are treated as functions at the SAFL-level and applied to SAFL expressions.

The treatment of Magma fragments is similar to that of primitive functions (such as `+`, `-`, `*` etc.). In particular, Magma code fragments are expanded in-place. For example, if a specification contains two Magma fragments of the form, `<% ripple_add %>`, then the generated hardware contains two separate ripple adders. Note that if we require a shared `ripple_adder` then we can encapsulate the Magma fragment in a SAFL function definition and rely on SAFL's resource-awareness properties. For example, the specification:

```
fun add(x, y) = <% ripple_add %> (x,y)
fun mult_3(x) = add(x, add(x,x))
```

contains a single ripple adder shared between the two invocations within the definition of the `mult_3(x)` function. Since embedded Magma code fragments represent pure functions (i.e. do not cause side effects) they do not inhibit SAFL-level program transformation. Thus our existing SAFL-level transformations (corresponding to resource duplication/sharing [105], hardware/software co-design [106]—see Chapter 9 etc.) remain valid. A larger example of SAFL/Magma integration is presented in the DES example described in Chapter 10 and Appendix Appendix A. Various source-level transformations are applied to the DES specification in Appendix Appendix B.

Implementation and Technical Details

Consider the general case of a Magma fragment, m, embedded in SAFL:

$$\texttt{<\%}\ m\ \texttt{\%>}(e_1, \ldots, e_k)$$

where $e_1, \ldots, e_k$ are SAFL expressions. On encountering the embedded Magma code fragment, `<%` m `%>`, our compiler performs the following operations:

1. An ML program, $\mathcal{M}$, (represented as a string) is constructed by concatenating the *library block* together with commands to instantiate the `Magma_Code` functor in its synthesis interpretation (see above).
2. The bit-widths of SAFL expressions, $e_1, \ldots, e_k$, are determined (bit-widths of variables are known to the SAFL compiler) and ML code is added to $\mathcal{M}$ to construct corresponding busses, $B_1, \ldots, B_k$, of the appropriate widths (using the `Magma.new_bus` library call).
3. $\mathcal{M}$ is further augmented with code to:
   a) execute ML expression, $m(B_1, \ldots, B_k)$, which, since the library block has been instantiated in its synthesis interpretation, results in the generation of a netlist; and
   b) wrap up the resulting netlist in a Verilog `module` declaration (adding Verilog `wire` declarations as appropriate).
4. A new ML session is spawned as a separate process and program $\mathcal{M}$ is executed within it.
5. The output of $\mathcal{M}$, a Verilog module declaration representing the compiled Magma code fragment, is returned to the SAFL compiler where it is added to the object code. Our SAFL compiler also generates code to instantiate the module, connecting it to the wires corresponding to the output ports of SAFL expressions $e_1, \ldots, e_k$. The module's output is connected to the wires which give the result of the SAFL expression.

In order that the ML-expression $m(B_1, \ldots, B_k)$ type checks, m must evaluate to a function, $\mathcal{F}$, with a type of the form:

```
(bit list * bit list * ... * bit list)  ->  bit list
```

with the arity of $\mathcal{F}$'s argument tuple equal to k. If m does not have the right type then a type-error is generated in the ML-session spawned to execute $\mathcal{M}$. Our SAFL compiler traps this ML type-error and generates a meaningful error of its own, indicating the offending line-number of the SAFL/Magma specification. In this way we ensure that the bit-widths and number of arguments applied to `<%` m `%>` at the SAFL-level match those expected at the Magma-level.

Another property we wish to ensure at compile time is that the output port of a Magma-generated circuit is of the right width. We achieve this by incorporating width information corresponding to the output port of Magma-generated hardware into our SAFL compiler's type-checking phase. Determining the width of a Magma specification's output port is trivial—it is simply the length of the `bit` list returned when $m(B_1, \ldots, B_k)$ is executed.

8.3 Aside: Embedding Magma in VHDL/Verilog

A common practice in the hardware design industry is to generate repetitive combinatorial logic by writing scripts (in a language such as Perl) which, when

executed, generates the necessary VHDL or Verilog code. The output of the script is then cut and pasted into the VHDL/Verilog design and the glue-code required to integrate the two written manually. Clearly there are a number of ways in which this design methodology can be improved. In particular it would be beneficial if (*i*) type checking could be performed across the Verilog/VHDL–scripting language boundary and (*ii*) the necessary glue-code generated automatically at compile time. The question that naturally arises is whether it is possible to use the SAFL-Magma integration techniques we have already described to integrate, say, Verilog and Magma.

Although the complex syntax of the Verilog language makes integration with Magma more difficult the basic principles outlined earlier in the paper are still applicable. Since the widths of Verilog variables are statically known to the Verilog compiler we can use the same width-checking techniques across the Verilog-Magma boundary that we employed across the SAFL-Magma divide in Section 8.2.2. We have devised three different forms of integration mechanism which we believe would be of use to Verilog programmers. These are mentioned briefly below:

Expressions

In the context of a Verilog expression (e.g. the right-hand-side of an `assign` statement), integration can be performed using the function-call mechanism already described in the context of SAFL. For example, a Verilog design may contain code such as:

```
assign after_perm = <% perm p_initial %>(before_perm);
```

Here, the Magma expression is statically expanded and treated in a similar way to one of Verilog's primitive operators. The Magma code for `perm` and `p_initial` can be seen in Appendix Appendix A.

Explicit Module Definitions

In some cases an engineer may wish to treat a Magma function as a named Verilog module which can subsequently be instantiated in the usual Verilog fashion. To handle this type of integration we introduce the following form:

```
module ModName(out, in_1, in_2) -->
      <% carry_chain adder B.b0 %>
```

We use the symbol `-->` to indicate that the module's body is specified by the given Magma expression. Note that an explicit output port, `out`, is required to read the result of the function. This form of integration can be seen as syntactic sugar. In general, it can be translated into the expression-integration form as follows:

```
module ModName(out, in_1, ..., in_N);
  output out;
  input in_1, ..., in_N;
  assign out =
          <% ... Magma expression ... %>(in_1, ..., in_N);
endmodule
```

Implicit Module Definitions

It is often convenient to avoid the explicit definition of a named module (e.g. if it is instantiated only once). For this reason we propose a third form of integration as follows:

```
<% perm p_initial %> my_perm(out_w, in_w);
```

In this case the augmented Verilog compiler automatically generates a fresh module definition (with a name of its choosing), instantiates it (with instance name `my_perm`) and connects it to existing wires `out_w` and `in_w`. Again, notice that in the Verilog domain it is necessary to explicitly mention the output of the function. In contrast, in the Magma domain, function composition can still be used to connect hardware blocks together without the overhead of explicitly declaring connecting wires. For this reason, designers may wish to move as much of the combinatorial logic specification as possible into the Magma portion of the design.

8.4 Summary

In this chapter we have motivated and described a technique for combining both behavioural and structural-level hardware specification in a stratified pure functional language. We believe that the major advantages of our approach are as follows:

- As in Verilog and VHDL, we are able to describe systems consisting of both behavioural and structural components.
- SAFL-level program transformation remains a powerful technique for architectural exploration. The functional nature of the SAFL/Magma integration means that our library of SAFL transformations are still applicable.
- By only dealing with combinatorial circuits at the structural-level we eliminate the problems associated with graph-sharing in a pure functional language (see Section 8.1). We do not sacrifice expressivity: cyclic (sequential) circuits can still be formed by composing combinatorial fragments at the SAFL-level in a more controlled way.

We also showed how similar techniques can be used to embed languages such as Magma into industrial HDLs such as Verilog or VHDL. We believe that this approach offers a great deal over the ad-hoc "Perl-script" technique so commonly

employed in practice. In particular: (*i*) type-checking across the Verilog-scripting language boundary catches a class of common errors; (*ii*) time-consuming glue-code required for the integration is generated automatically; and (*iii*) as is often argued, the features of a functional language such as polymorphism, static type-checking and higher-order functions, encourage code-reuse and aid correctness. Another compelling benefit for integrating a functional-language (such as Lava or Magma) into Verilog/VHDL is that the techniques of Claessen *et al.* for concisely encapsulating place-and-route information [38] can potentially be employed to generate efficient layouts for repetitive combinatorial logic.

Whilst we accept that the majority of working hardware engineers are not familiar with functional programming (and hence not likely to embrace the technique) we also observe that there are an increasing number of Computer Science graduates (as opposed to Electronic Engineering graduates) seeking employment in the hardware design sector[3]. With this in mind, it is conceivable that an easily implementable integration mechanism between languages such as Magma/Lava and industrial HDLs such as Verilog/VHDL (see Section 8.3) may help to make the tried-and-tested technique of structural hardware specification using functional languages more attractive to the hardware design industry.

[3] We do not wish to imply that EE graduates are inferior to their CS counterparts! We are simply commenting that they often have different areas of expertise.

9

Transformation of SAFL Specifications

In their survey paper [98], McFarland *et al* highlight source-level transformation of input specifications as an important technique for the future of HLS. The idea is that high-level transformation of behavioural specifications will be used to express a number of architectural trade-offs. Such transformations may be applied fully automatically, fully manually or (ideally) within a unified framework facilitating a combination of the two approaches.

Some researchers have investigated high-level transformations: Walker and Thomas formulated behavioural transformations [142] within the framework of the System Architect's Workbench [135] and Source-level transformations of behavioural VHDL where proposed by Nijhar and Brown [107]. Despite this, however, program transformation has had very little impact on the hardware design industry. We believe that the two main reasons why this is the case are:

1. Many features commonly found in behavioural HDLs make it difficult to apply program-transformation techniques (e.g. an imperative programming style with low-level circuit structuring primitives such as Verilog's `module` construct).
2. It is difficult for a designer to know what the impact of a behavioural-level transformation will have on a generated design.

In contrast, the SAFL language is designed to facilitate source-to-source transformation. Whereas traditional "black-box" synthesis systems synthesise hardware according to user-supplied constraints, our approach is to select a particular implementation by applying transformation rules to the SAFL source as a pre-compilation phase. The two points above are addressed specifically in the design of SAFL:

- The functional properties of the language allow equational reasoning and hence make a wide range of transformations applicable (as we do not have to worry about side effects).
- The resource-aware properties of SAFL give many transformations precise meaning at the design-level (e.g. we know that duplicating a function definition in the source is guaranteed to duplicate the corresponding resource in the generated circuit).

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 141–154, 2004.

Recall that we have already seen several examples of source-to-source program transformation of SAFL specifications: in Chapter 4 we saw how local transformation of SAFL expressions can be used to represent static scheduling policies; Chapter 3 demonstrated that simple applications of Burstall and Darlington's *fold/unfold* transformations [29] can be used to represent structural tradeoffs (e.g. resource duplication vs. sharing). In this chapter we:

1. give an example of a much more complicated transformation which allows a designer to investigate hardware/software partitioning by transforming SAFL specifications (Section 9.1);
2. demonstrate a pipelining transformation which makes use of the extra capabilities offered by the SAFL+ language (Section 9.3); and
3. present our thoughts on how these transformations could be integrated into a semi-automatic transformation tool.

9.1 Hardware Software CoDesign

The purpose of this section is to demonstrate how hardware/software partitioning can be seen as a source-to-source transformation at the SAFL level thus providing a formal framework in which to investigate hardware/software co-design. In fact we go one step further than traditional co-design since as well as partitioning a specification into hardware and software parts our transformation procedure can also synthesise an architecture tailored specifically for executing the software part. This architecture consists of any number of interconnected heterogeneous processors. There are a number of advantages to our approach:

- Synthesising an architecture specifically to execute a known piece of software can offer significant advantages over a fixed architecture [113].
- The ability to synthesise multiple processors allows a wide range of area-time tradeoffs to be explored. Not only does hardware/software partitioning affect the area-time position of the final design, but the number of processors synthesised to execute the software part is also significant: increasing the number of processors pushes the area up whilst potentially reducing execution time (as the processors can operate in parallel).
- Resource-awareness allows a SAFL specification to represent shared resources. This increases the power of our partitioning transformation since, for example, multiple processors can access a shared resource (see Figure 9.1 for an example).

9.1.1 Comparison with Other Work

Hardware/software co-design is well-studied and many tools have been built to aid the partitioning process [17, 35]. Although these systems differ in their

approach to co-design they are similar in so far as partitioning is a "black-box" phase performed as part of the synthesis process. By making partitioning visible at the source-level we believe our approach to be more flexible—hardware/software co-design is just one of a library of source-to-source transformations which can be applied incrementally to explore a wide range of architectural trade-offs.

The idea of converting a program into a parameterised processor and corresponding instruction memory is not new; Page described a similar transformation [113] within the framework of Handel [112] (a subset of Occam for which a silicon compiler was written). However, the extra transformational power provided by our functional specification language allows us to generalise this work in a number of ways. Rather than synthesising a single parameterised processor our method allows one to generate a much more general architecture consisting of multiple communicating processors accessing a set of (potentially shared) hardware resources.

9.2 Technical Details

The first step in the partitioning transformation is to define a partitioning function, π, specifying which SAFL functions are to be implemented directly in hardware and which are to be mapped to a processor for software execution. Automated partitioning is not considered here; we assume that π is supplied by the user. For expository purposes we initially describe a transformation where all processors are variants of a stack machine: Section 9.2.1 describes the operation of the stack machine and Section 9.2.2 shows how it can be encoded as a SAFL function; a compiler from SAFL to stack code is presented in Section 9.2.3. In Section 9.2.6 we generalise our partitioning transformation to a network of heterogeneous processors.

Let $\mathcal{M}$ be the set of processor instances used in the final design. We assume a (partial) partitioning function

$$\pi : \mathit{SAFL\ function\ name} \rightharpoonup \mathcal{M}$$

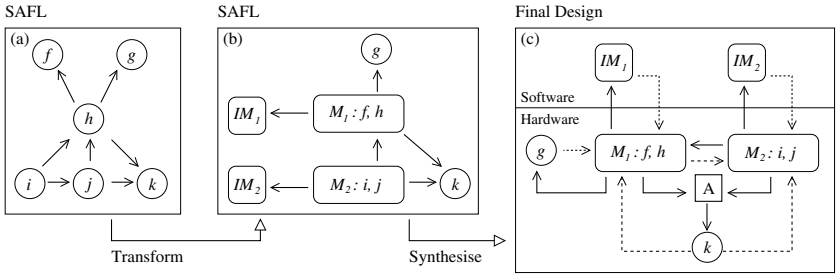
mapping the function definitions in our SAFL specification onto processors in $\mathcal{M}$. $\pi(f)$ is the processor on which function f is to be implemented. If $f \notin \mathit{Dom}(\pi)$ then we realise f in hardware, otherwise we say that f is *located* on machine $\pi(f)$. Note that multiple functions can be mapped to the same processor.

We extend π to a transformation function

$$\hat{\pi} : \mathit{SAFL\ Program} \rightarrow \mathit{SAFL\ Program}$$

such that given a SAFL program, P, $\hat{\pi}(P)$ is another SAFL program which respects the partitioning function π. Figure 9.1 shows the effect of a partitioning transformation, $\hat{\pi}$, where

$$\begin{aligned} \mathcal{M} &= \{M_1, M_2\}; \text{ and} \\ \pi &= \{(f, M_1),\ (h, M_1),\ (i, M_2),\ (j, M_2)\} \end{aligned}$$



Partitioning: (a) shows the call-graph of a SAFL specification, P; (b) shows the call-graph of $\hat{\pi}(P)$, where $\pi = \{(f, M_1), (h, M_1), (i, M_2), (j, M_2)\}$. IM_1 and IM_2 are instruction memory functions (see Section 9.2.2); (c) shows the structure of the final circuit after compilation. The box marked 'A' represents an arbiter (inserted automatically by the SAFL compiler) protecting shared resource k; the bold arrows represent calls, the dotted arrows represent return values.

Fig. 9.1. A diagrammatic view of the partitioning transformation

In this example we see that g and k are implemented in hardware since $g, k \notin Dom(\pi)$. $\hat{\pi}(P)$ contains function definitions: $\texttt{M}_1$, $\texttt{M}_2$, $\texttt{IM}_1$, $\texttt{IM}_2$, $\texttt{g}$ and $\texttt{k}$ where $\texttt{M}_1$ and $\texttt{M}_2$ are processor instances and $\texttt{IM}_1$ and $\texttt{IM}_2$ are instruction memories (see Section 9.2.2).

9.2.1 The Stack Machine Template

Our stack machine can be seen as a cut-down version of both Landin's SECD machine [90] and Cardelli's Functional Abstract Machine [30]. Each instruction has an op-code field and an operand field n. Figure 9.2 defines the instructions supported by the stack machine.

The stack machine *template*, *SMT*, is an abstract model of the stack machine parameterised on the code it will have to execute. Given a stack machine program, s, (i.e. a list of stack machine instructions as outlined above) $SMT\langle s \rangle$ is a stack machine *instance*: a SAFL function encoding a stack machine specialised for executing s. Our notion of a template is similar to a VHDL *generic*.

9.2.2 Stack Machine Instances

A stack machine instance, $\texttt{SM}_i \in \mathcal{M}$, is a SAFL function of the form:

$$\texttt{fun SM}_i\texttt{(a}_1\texttt{, ..., a}_{n_i}\texttt{, PC, SP) = ...}$$
$$\text{where } n_i = max(\{arity(f) \mid \pi(f) = \texttt{SM}_i\})$$

Arguments `PC` and `SP` are used to store the program counter and stack pointer respectively; $\texttt{a}_1, \ldots, \texttt{a}_{n_i}$ are used to receive arguments of functions located on $\texttt{SM}_i$. Each stack machine instance is associated with an instruction memory function, $\texttt{IM}_i$ of the form:

```
fun IM_i(address) =
   case address of 0 => instruction_0
                 | 1 => instruction_1
                    ... etc.
```

$\texttt{SM}_i$ calls $\texttt{IM}_i(\texttt{PC})$ to load instructions for execution.

PushC(n)	push constant n onto the stack
PushV(n)	push variable (from offset n into the current stack)
PushA(n)	push the value of the stack machine's argument $\texttt{a}_n$ (see Section 9.2.2) to the stack
Squeeze(n)	pop top value; pop next n values; re-push top value
Return(n)	pop result; pop link; pop n arguments; re-push result; branch to link
Call_Int(n)	push address of next instruction onto stack and branch to address n
Jz(n)	pop a value; if it is zero branch to address n
Jmp(n)	jump to address n
Alu2(n)	pop two values; do 2-operand builtin operation n on them and push the result
Halt	terminate the stack machine returning the value on top of the stack

We define a family of instructions to allow the stack machine to call external functions:

$\texttt{Call\_Ext}_f$	pops each of f's arguments from the stack; invokes the external function f and pushes the result to the top of the stack.

Fig. 9.2. The instructions provided by our stack machine

For example, consider a stack machine instance, $\texttt{SM}_{f,h}$, where we choose to locate functions f (of arity 2) and h (of arity 3). Then $n_{f,h} = 3$ yielding signature: $\texttt{SM}_{f,h}(\texttt{a}_1, \texttt{a}_2, \texttt{a}_3, \texttt{PC}, \texttt{SP})$. $\texttt{IM}_{f,h}$ is an instruction memory containing compiled code for f and h. To compute the value of $h(x, y, z)$ we invoke $\texttt{SM}_{f,h}$ with arguments $\texttt{a}_1 = x$, $\texttt{a}_2 = y$, $\texttt{a}_3 = z$, $\texttt{PC} = ext\_h_{entry}$ (h's external entry point—see Section 9.2.3) and $\texttt{SP} = 0$. Similarly to compute the value of $f(x, y)$ we invoke $\texttt{SM}_{f,h}$ with arguments $\texttt{a}_1 = x$, $\texttt{a}_2 = y$, $\texttt{a}_3 = 0$, $\texttt{PC} = ext\_f_{entry}$ and $\texttt{SP} = 0$. Note how we pad the a-arguments with 0's since $arity(f) < 3$.

The co-design of hardware and software means that instructions and ALU operations are only added to $\texttt{SM}_i$ if they appear in $\texttt{IM}_i$. Parameterising the stack machine template in this way can considerably reduce the area of the final design since we remove redundant logic in each processor instance.

We can consider many other areas of parameterisation. For example we can adjust the op-code width and assign op-codes to minimise instruction-decoding delay [113]. Appendix Appendix C gives the SAFL code for a 16-bit stack ma-

chine instance[1], an `alu2` function and and an example stack machine program which computes triangular numbers.

9.2.3 Compilation to Stack Code

Figure 9.3 gives a compilation function from SAFL to stack-based code. Although the translation of many SAFL constructs is self-explanatory, the compilation rules for function definition and function call require further explanation:

Compiling Function Definitions

The code generated for function definition

$$\texttt{fun}\ f(x_1, \ldots, x_k) = e$$

requires explanation in that we create 2 distinct entry points for f: f_{entry} and ext_f_{entry}. The *internal* entry point, f_{entry}, is used when f is invoked internally (i.e. with a `Call_Int` instruction). The *external* entry point, ext_f_{entry}, is used when f is invoked externally (i.e. via a call to $\pi(f)$, the machine on which f is implemented). In this latter case, we simply execute k `PushA` instructions to push f's arguments onto the stack before jumping to f's internal entry point, f_{entry}.

Compiling Function Calls

Suppose function g is in software ($g \in Dom(\pi)$) and calls function f. The code generated for the call depends on the location of f relative to g. There are three possibilities:

1. If f and g are both implemented in software on the same machine ($f \in Dom(\pi) \wedge \pi(f) = \pi(g)$) then we simply push each of f's arguments to the stack and branch to f's internal entry point with a `Call_Int` instruction. The `Call_Int` instruction pushes the return address and jumps to f_{entry}; the compiled code for f is responsible for popping the arguments and link leaving the return value on the top of the stack.
2. If f is implemented in hardware ($f \notin Dom(\pi)$) then we push each of f's arguments to the stack and invoke the hardware resource corresponding to f by means of a $\texttt{Call\_Ext}_f$ instruction. The $\texttt{Call\_Ext}_f$ instruction pops each of f's arguments, invokes resource f and pushes f's return value to the stack.
3. If f and g are both implemented in software but on different machines ($f, g \in Dom(\pi) \wedge \pi(f) \neq \pi(g)$) then g needs to invoke $\pi(f)$ (the machine on which f is located). We push $\pi(f)$'s arguments to the stack: the arguments for f possibly padded by 0s (see Section 9.2.2) followed by the program counter `PC` initialised to ext_f_{entry} and the stack pointer `SP` initialised to 0. We then invoke $\pi(f)$ using a $\texttt{Call\_Ext}_{\pi(f)}$ instruction.

[1] Approximately 2000 2-input equivalent gates when implemented in hardware. For simplicity we consider a simple stack machine with no `Call_Ext` instructions.

Let σ, be an environment mapping variable names to stack offsets (offset 0 signifies the top of the stack). Let g be the name of the function we are compiling. Then $[\![\cdot]\!]^g\sigma$ gives an instruction list corresponding to g. (We omit g for readability in the following—it is only used to identify whether a called function is located on the same machine).
We use the notation $\sigma\{x \mapsto n\}$ to represent environment σ extended with x mapping to n. σ^{+n} represents an environment constructed by incrementing all stack offsets in σ by n—i.e. $\sigma^{+n}(x) = \sigma(x) + n$. $\emptyset$ is the empty environment. The infix operator @ appends instruction lists. $Repeat(l, n)$ is l @ ... @ l (n times); (this is used to generate instruction sequences to pad argument lists with 0s).

$$[\![c]\!]\sigma \stackrel{\text{def}}{=} [\texttt{PushC}(c)]$$

$$[\![x]\!]\sigma \stackrel{\text{def}}{=} [\texttt{PushV}(\sigma(x))]$$

$$[\![f(e_1,\ldots,e_k)]\!]\sigma \stackrel{\text{def}}{=} \begin{cases} [\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; \ldots \;@\; [\![e_k]\!]\sigma^{+(k-1)} \\ \qquad @\; [\texttt{Call\_Ext}_f] \qquad \text{if } f \notin Dom(\pi) \\ [\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; \ldots \;@\; [\![e_k]\!]\sigma^{+(k-1)} \\ \quad @\; [\texttt{Call\_Int}(f_{entry})] \\ \qquad\qquad \text{if } f \in Dom(\pi) \wedge \pi(f) = \pi(g) \\ [\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; \ldots \;@\; [\![e_k]\!]\sigma^{+(k-1)} \\ \quad @\; Repeat([\texttt{PushC}(0)], \\ \qquad\qquad arity(\pi(f)) - 2 - k) \\ \quad @\; [\texttt{PushC}(ext\_f_{entry}), \texttt{PushC}(0), \\ \qquad\qquad \texttt{Call\_Ext}_{\pi(f)}] \\ \qquad\qquad \text{if } f \in Dom(\pi) \wedge \pi(f) \neq \pi(g) \end{cases}$$

$$[\![a(e_1,e_2)]\!]\sigma \stackrel{\text{def}}{=} [\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; [\texttt{Alu2}(a)]$$

$$[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!]\sigma \stackrel{\text{def}}{=} [\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1}\{x \mapsto 0\} \;@\; [\texttt{Squeeze}(1)]$$

$$[\![\texttt{ if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\sigma \stackrel{\text{def}}{=} \text{let } l \text{ and } l' \text{ be new labels in} \\ [\![e_1]\!]\sigma \;@\; [\texttt{Jz } (l)] \;@\; [\![e_2]\!]\sigma \;@\; [\texttt{Jmp } (l'), \textit{label: } l] \\ @\; [\![e_3]\!]\sigma \;@\; [\textit{label: } l']$$

$$[\![\texttt{fun } g(x_1,\ldots,x_k) = e]\!] \stackrel{\text{def}}{=} [\textit{label: } g_{entry}] \\ @\; [e]^g\emptyset\{x_k \mapsto 1, x_{k-1} \mapsto 2, \ldots, x_1 \mapsto k\} \\ @\; [\texttt{Return}(k)] \\ @\; [\textit{label: } ext\_g_{entry}, \texttt{PushA}(1), \ldots, \texttt{PushA}(k), \\ \texttt{Call\_Int}(g_{entry}), \texttt{Halt}]$$

Fig. 9.3. Compiling SAFL into Stack Code for Execution on a Stack Machine Instance

9.2.4 The Partitioning Transformation

Having introduced the stack machine (Section 9.2.1) and the associated compilation function (Section 9.2.3) the details of the partitioning transformation, $\hat{\pi}$, are as follows:

Let P be the SAFL program we wish to transform using π. Let f be a SAFL function in P with definition d_f of the form

$$\texttt{fun } f(x_1, \ldots, x_k) = e$$

We construct a partitioned program $\hat{\pi}(P)$ from P as follows:

1. For each function definition $d_f \in P$ to be mapped to hardware (i.e. $f \notin \mathit{Dom}(\pi)$) create a variant in $\hat{\pi}(P)$ which is as d_f but for each call, $g(e_1, \ldots, e_k)$:
   If $g \in \mathit{Dom}(\pi)$ then replace the call $g(\vec{e})$ with a call:
   $$m(e_1, \ldots, e_k, \underbrace{0, \ldots, 0,}_{\mathit{arity}(m)-2-k} \mathit{ext\_g}_{\mathit{entry}}, 0)$$
   where $m = \pi(g)$, the stack machine instance on which g is located.
2. For each $m \in \mathcal{M}$:
   a) Compile instruction sequences for functions located on m:
   $$\mathit{Code}_m = \{[\![d_f]\!] \mid \pi(f) = m\}$$
   b) Generate *machine code* for m, MCode_m, by resolving symbols in Code_m, assigning opcodes and converting into binary representation.
   c) Generate an *instruction memory* for m by adding a function definition, $\texttt{IM}_m$, to $\hat{\pi}(P)$ of the form:
   ```
   fun IM_m(address) =
      case address of 0 => instruction_0
                    | 1 => instruction_1
                       ... etc.
   ```
   where each `instruction_`i is taken from MCode_m.
   d) Generate a stack machine instance, $\mathit{SMT}\langle \mathit{Code}_m \rangle$ and append it to $\hat{\pi}(P)$.

For each $m \in \mathcal{M}$, $\hat{\pi}(P)$ contains a corresponding processor instance and instruction memory function. When $\hat{\pi}(P)$ is compiled to hardware resource-awareness ensures that each processor definition function becomes a single processor and each instruction memory function becomes a single instruction memory. The remaining functions in $\hat{\pi}(P)$ are mapped to hardware resources as required. Function calls are synthesised into optimised communication paths between the hardware resources—see Figure 9.1(c).

9.2.5 Validity of Partitioning Functions

This section concerns some fine technical details—it can be skipped on first reading. Recall the static-allocation restriction on mutually-recursive SAFL programs given in Section 3.4. Phrased informally the restriction can be written:

In order for a SAFL program to be valid, all recursive calls, including those calls which form part of mutually-recursive cycle, may only occur in tail-context. Non-recursive calls may appear freely.

Unfortunately, in general, a partitioning function, π, may transform a valid SAFL program, P, into an invalid SAFL program, $\hat{\pi}(P)$, which does not satisfy the recursion restrictions. For example consider the following program, P_{bad}:

```
fun f(x) = x+1;
fun g(x) = f(x)+2;
fun h(x) = g(x+3);
```

Partitioning P_{bad} with $\pi = \{(f, \texttt{SM}),\ (h, \texttt{SM})\}$ yields a new program, $\hat{\pi}(P_{bad})$, of the form:

```
fun IM(PC) = ...
fun SM(x,PC,SP) = ... let t = <top-of-stack>
                          in g(t) ...
fun g(x) = SM(x, <ext_f_entry>, 0) + 2;
```

$\hat{\pi}(P_{bad})$ has invalid recursion between `g` and `SM`. The problem is that the call to `SM` in the body of `g` is part of a mutually-recursive cycle and is not in tail-context.

We therefore require a restriction on partitions π to ensure that if P is a valid SAFL program then $\hat{\pi}(P)$ will also be a valid SAFL program. We give the following sufficient condition:

π is a valid partition with respect to SAFL program, P, iff all cycles occurring in the call graph of $\hat{\pi}(P)$ already exist in the call graph of P, with the exception of self-cycles generated by direct tail-recursion.

Thus, in particular, new functions in $\hat{\pi}(P)$—i.e. stack machines and their instructions memories—must not have mutual recursion with any other functions.

9.2.6 Extensions

Fine Grained Partitioning

We have presented a program transformation to map function definitions to hardware or software, but what if we want to map *part* of a function definition to hardware and the rest to software? This can be achieved by applying fold/unfold transformations before our partitioning transformation. For example, consider the function

```
fun f(x,y) = if x=0 then y
                    else f(x-1, x*y - 7 + 5*x)
```

If we choose to map `f` to software our design will contain a processor and associated machine code consisting of a sequence of instructions representing multiply `x` and `y`, subtract 7, add 5 times `x`. However, consider transforming `f` with a single application of the *fold*-rule [29]:

```
fun i(x,y) = x*y-7 + 5*x
fun f(x,y) = if x=0 then y else f(x-1, i(x,y))
```

Now mapping `f` to software and `i` to hardware leads to a software representation for `f` containing fewer instructions and a specialised processor with a `x*y-7 + 5*x` instruction.

Dealing with Heterogeneous Processors

So far we have only considered executing software on a network of stack machines. Although the stack machine is a familiar choice for expository purposes, in a real design one would often prefer to use different architectures. For example, specialised VLIW [67] architectures are a typical choice for data-dominated embedded systems since many operations can be performed in parallel without the overhead of dynamic instruction scheduling. In general, designs often consist of multiple communicating processors chosen to reflect various area and performance constraints. Our framework can be extended to handle a network of heterogeneous processors as follows:

Let *Templates* be a set of processor templates (cf. the stack machine template, *SMT*, in section 9.2.1).

Let *Compilers* be a set of compilers from SAFL to machine code for processor templates.

As part of the transformation process, the user now specifies two extra functions:

$$\delta : \mathcal{M} \rightarrow \mathit{Templates}$$
$$\tau : \mathcal{M} \rightarrow \mathit{Compilers}$$

δ maps each processor instance, $m \in \mathcal{M}$, onto a SAFL processor template and τ maps each $m \in \mathcal{M}$ onto an associated compiler. We then modify the transformation procedure described in Section 9.2.4 to generate a partitioned program, $\hat{\pi}_{\delta,\tau}(P)$ as follows: for each $m \in \mathcal{M}$ we generate machine code, MCode_m, using compiler $\tau(m)$; we then use processor template, $MT = \delta(m)$, to generate processor instance $MT\langle \mathit{MCode}_m \rangle$ and append this to $\hat{\pi}_{\delta,\tau}(P)$.

Extending the SAFL Language

Recall that the SAFL language specifies that all recursive calls must be in tail-context. Since only tail-recursive calls are permitted, our silicon compiler is able to statically allocate all the storage needed for a SAFL program.

As an example of these restrictions consider the following definitions of the factorial function:

```
rfact(x) = if x=0 then 1 else x*rfact(x-1)
ifact(x,a) = if x=0 then a else ifact(x-1,x*a)
```

`rfact` is not a valid SAFL program since the recursive call is not in a tail-context. However the equivalent tail-recursive factorial function, `ifact` which uses a second argument to accumulate partial results is a valid SAFL program.

Although one can sometimes transform a non-tail recursive program into an equivalent tail-recursive one [29], this is not always easy or natural. The transformation of factorial into its tail-recursive equivalent is only possible because multiplication is an associative operator. Thus, in general we require a way of extending SAFL to handle general unrestricted recursion. Our partitioning transformation provides us with one way to do this:

Consider a new language, SAFL_R constructed by removing the recursion restrictions from SAFL. We can use our partitioning transformation to transform SAFL_R to SAFL simply by ensuring that each function definition containing recursion other than in a tail-call context is mapped to software. Note that our compilation function (Figure 9.3) is already capable of dealing with general recursion without any modification.

9.3 Transformations from SAFL to SAFL+

Since SAFL is a pure functional language[2] program transformation can often be applied quite elegantly (as one does not have to worry about the complexity of side-effects). In contrast, the additional imperative features make program transformation awkward in the SAFL+ world. For this reason, we do not consider transformations which operate solely within the larger SAFL+ domain. However, we have found that we can usefully employ a class of transformations which convert pure functional SAFL code into SAFL+ code. We argue that SAFL to SAFL+ transformations give us the best of both worlds: since the transformation's source is pure SAFL, its application is not restricted by the presence of potential side-effects; furthermore, the code resulting from the transformation's application is able to take advantage of the extra IO capabilities (channels and references) provided by SAFL+. We use transformations from SAFL to SAFL+ to encode explicitly implementation-specific details which are left implicit at the SAFL-level. In the remainder of this section we outline one such transformation which converts a top-level SAFL function (in a particular form) into a pipelined stream processor.

Recall that a SAFL specification has one *top-level* function which is not called anywhere in the specification. The top-level function is often referred to as `main`; its formal parameters and result are realised as input/output ports (respectively).

Figure 9.4 shows a transformation which converts a top-level SAFL function (which provides a "fire-arguments-and-wait" interface) into a pipelined stream processor which reads its input data and writes its output data over synchronous channels. In order for the transformation to be applicable the top-level SAFL function must be in the specific form shown on the left hand side of Figure 9.4.

[2] As long as side-effecting `extern` functions are not used.

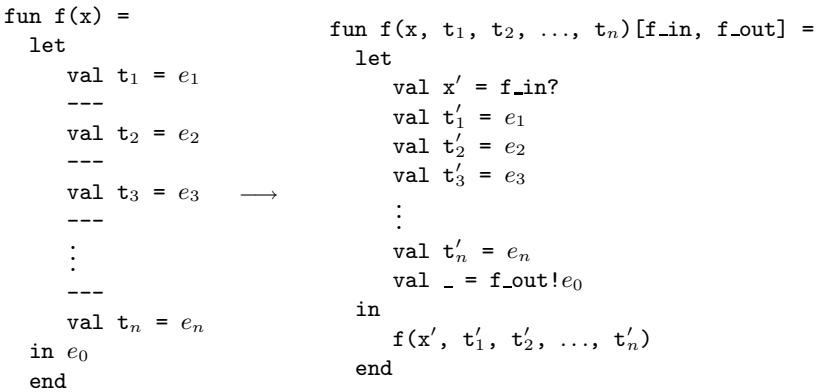


Fig. 9.4. Top-level pipelining transformation

We intend that functional transformations are used to manipulate the SAFL specification into this form if required.

Whereas the left hand side of Figure 9.4 computes each expression sequentially the pipelined version computes the expressions in parallel, using its argument registers to store intermediate values between pipeline stages. Two channels parameters are added: `f_in` is used to feed data into the pipeline, `f_out` is used to read data out of the pipeline. Each time the function is executed it reads its input, performs all pipeline stages and writes its output in parallel. (The underscore symbol used in the left hand side of a `val`-declaration is an ML-style anonymous binding.) On a recursive call the function's arguments are shifted right one place as data moves down the pipeline.

Let us illustrate the transformation with a simple example. Consider the function:

```
fun f(x) = f1(f2(f3(x)))
```

Two applications of the fold/unfold *abstraction* rule transforms this into a form ready for pipelining:

```
fun f(x) = let val v3 = f3(x)
               ---
               val v2 = f2(v3)
           in
               f1(v2)
           end
```

Applying the pipelining transformation yields:

```
fun f_pipe(x,v3,v2)[in,out] =
            let val new_x = in?
                val new_v3 = f3(x)
                val new_v2 = f2(v3)
```

```
            val _ = out!f1(v2)
         in
            f(new_x,new_v3,new_v2)
        end
```

The following code fragment gives an example of how the pipeline may be used:

```
fun demo_pipe() =
    static channel my_in
           channel external my_out
           fun producer(x) = my_in!x; producer(x+1)
    in
           producer(0) | f_pipe(0,0,0)[my_in, my_out]
    end
```

Function `f_pipe` is composed in parallel with a process which continuously writes data to the locally defined channel `my_in`. Function `f` continuously reads the data from `my_in`, computes each pipeline stage in parallel and writes data to `my_out`. Since `my_out` is defined as an external channel it is non-blocking (Our intention is that the data from `my_out` is read by some external device.) The first two values written to `my_out` are not valid outputs of `f`. However, from the third value onwards (once the pipeline has been filled with useful data), the value of `my_out` reflects `f` applied to the data written to the input channel.

The pipelining transformation is particularly useful in data-processing applications. Appendix Appendix B uses a combination of fold/unfold and the pipelining transformation to convert a non-pipelined SAFL DES specification into a 4-stage pipelined version. The example of Appendix Appendix B demonstrates how a concise (and hence hopefully correct!) SAFL specification can be mechanically transformed into a more efficient (but more complicated) design using a series of semantics-preserving transformations.

9.4 Summary

Source-level program transformation of a high level HDL is a powerful technique for exploring a wide range of architectural tradeoffs from an initial specification. We believe that transformation in the SAFL domain is particularly powerful since its functional properties make it easy to apply transformations and resource-awareness gives the transformations a precise meaning at the implementation-level.

A class of transformation that deserves further investigation is Partial Evaluation [81]. Although we have not studied the application of Partial Evaluation techniques to SAFL in depth, we believe that it could be used to transform a processor definition function and its corresponding instruction memory function into a single specialised unit with hardwired control.

We ultimately envisage the transformations described in this chapter (and also those described in chapters 3 and 4) being integrated into a *transformation tool* which helps designers apply source-to-source transformations to SAFL

specifications. Although we do not consider the proposed transformation assistant further here, we note that the development of such tools has been active research area for a long time. For example, in his 1979 PhD Thesis [46], Martin Feather designed a command driven transformation tool which assists a programmer apply fold/unfold transformations to a simple language of first-order recursion equations similar, in many respects, to SAFL. More recently Renault *et al* have developed a graphical tool which provides semi-automated support for applying fold/unfold transformations [126]. Having studied this system and other similar research [7, 118] we believe that much of the ground work for developing transformation assistants has already been investigated. It seems likely that a SAFL-specific transformation tool could be designed along similar lines.

10

Case Study

The purpose of this chapter is give an overview of the tools we use to translate SAFL(+) to silicon (see Section 10.1) with reference to a realistic example. A DES encryption/decryption circuit is specified in SAFL and implemented on an Altera Apex-II FPGA; area-time performance figures are given (Section 10.2).

To be applicable to real hardware design an HDL must be able to integrate with existing systems. We demonstrate that this is the case for SAFL by designing a SAFL interface which allows our DES design to write values on a monitor using a VGA driver that we implemented directly in RTL Verilog (Section 10.2.1).

10.1 The SAFL to Silicon Tool Chain

We start by demonstrating the tools which we use for mapping SAFL(+) onto FPGAs. Figure 10.1 shows a screenshot of the FLaSH compiler (see Chapter 5) in operation. FLaSH is implemented in the SML/NJ language [11], a dialect of Standard ML [101]. The leftmost window shows FLaSH running in an interactive SML/NJ session. In this window the user can enter commands to simulate SAFL(+) programs and compile SAFL(+) programs to RTL-Verilog. FLaSH also provides a number of visualisation tools which aid the designer. The middle window shows a graphical representation of the SAFL specification's corresponding intermediate graph; this gives the designer an intuition into how parts of a design will be mapped to hardware. The rightmost window shows a graphical representation of the SAFL specification's call graph. Edges on the call graph are annotated with the line/character position at which the corresponding call appears in the SAFL source. If an edge is highlighted in red then this indicates that the call is subject to a parallel sharing conflict. We find that this is a useful way for the designer to see at a glance which calls in a SAFL specification require arbitration. (The FLaSH compiler is integrated with AT&T's GraphViz [54] package to produce this graphical output.)

Once FLaSH has performed its task of high-level synthesis we feed the resulting RTL-Verilog file into the *Leonardo* tool which performs the task of *logic*

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 155–168, 2004.

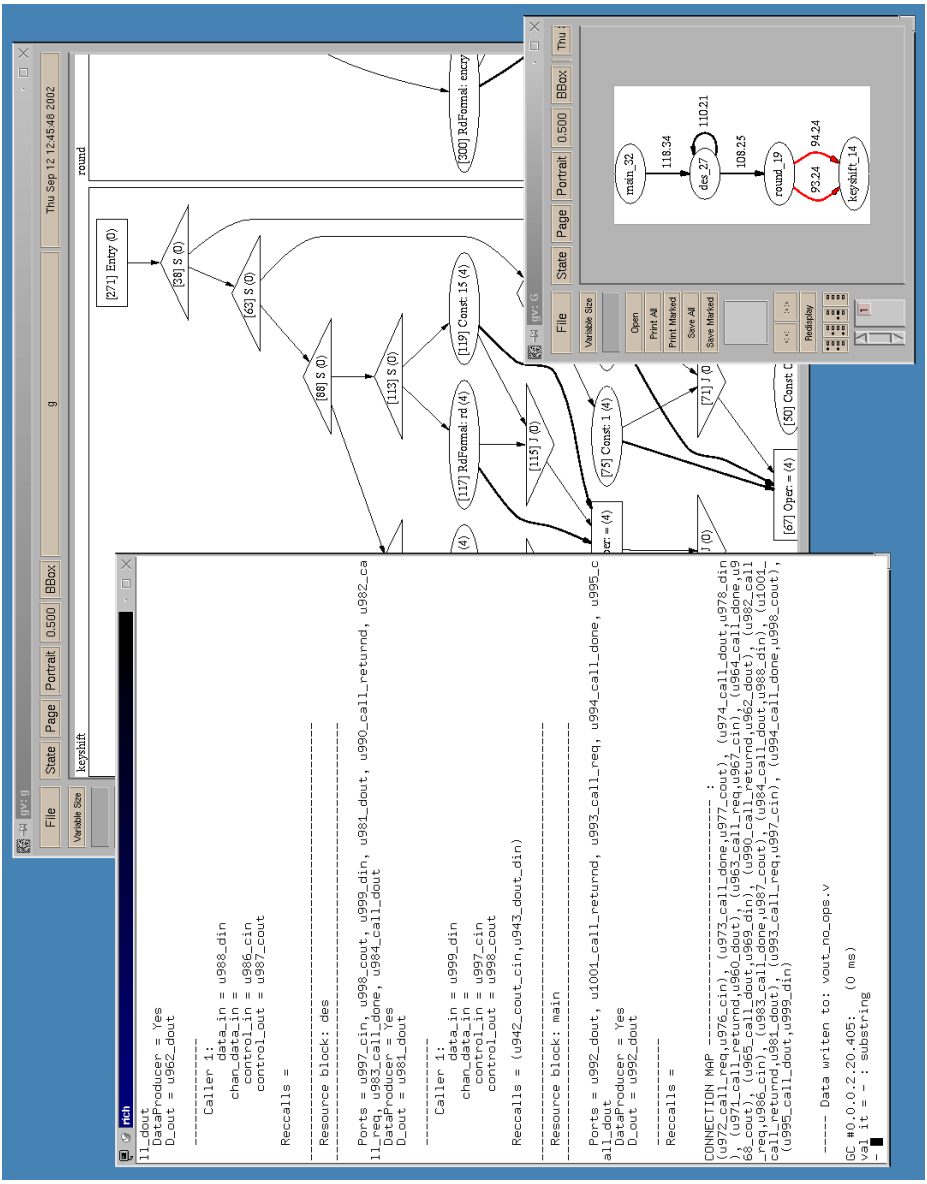


Fig. 10.1. Using the FLaSH compiler to compile a SAFL specification to RTL Verilog

synthesis, translating the RTL code into a netlist for implementation on a specified FPGA. (Note that there is no reason why we have to map SAFL to FPGAs; at this stage we could direct Leonardo to target a generic ASIC process.) Figure 10.2 shows Leonardo in operation. Note that the "Clock Frequency Report"

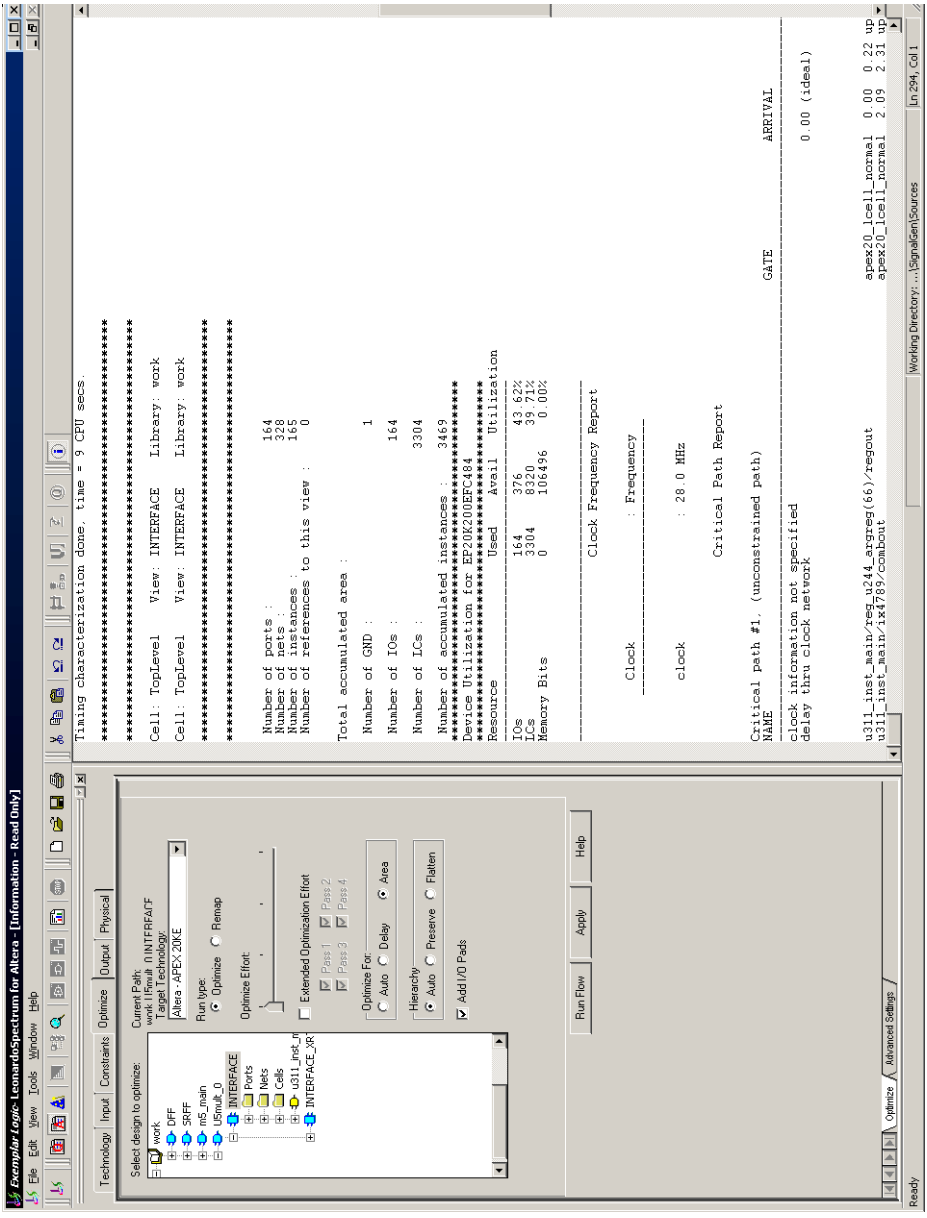


Fig. 10.2. Using the RTL-synthesis tool *Leonardo* to map the Verilog generated by the FLaSH compiler to a netlist

shown is only a rough estimate since we do not have any detailed place-and-route information at this stage of the design flow.

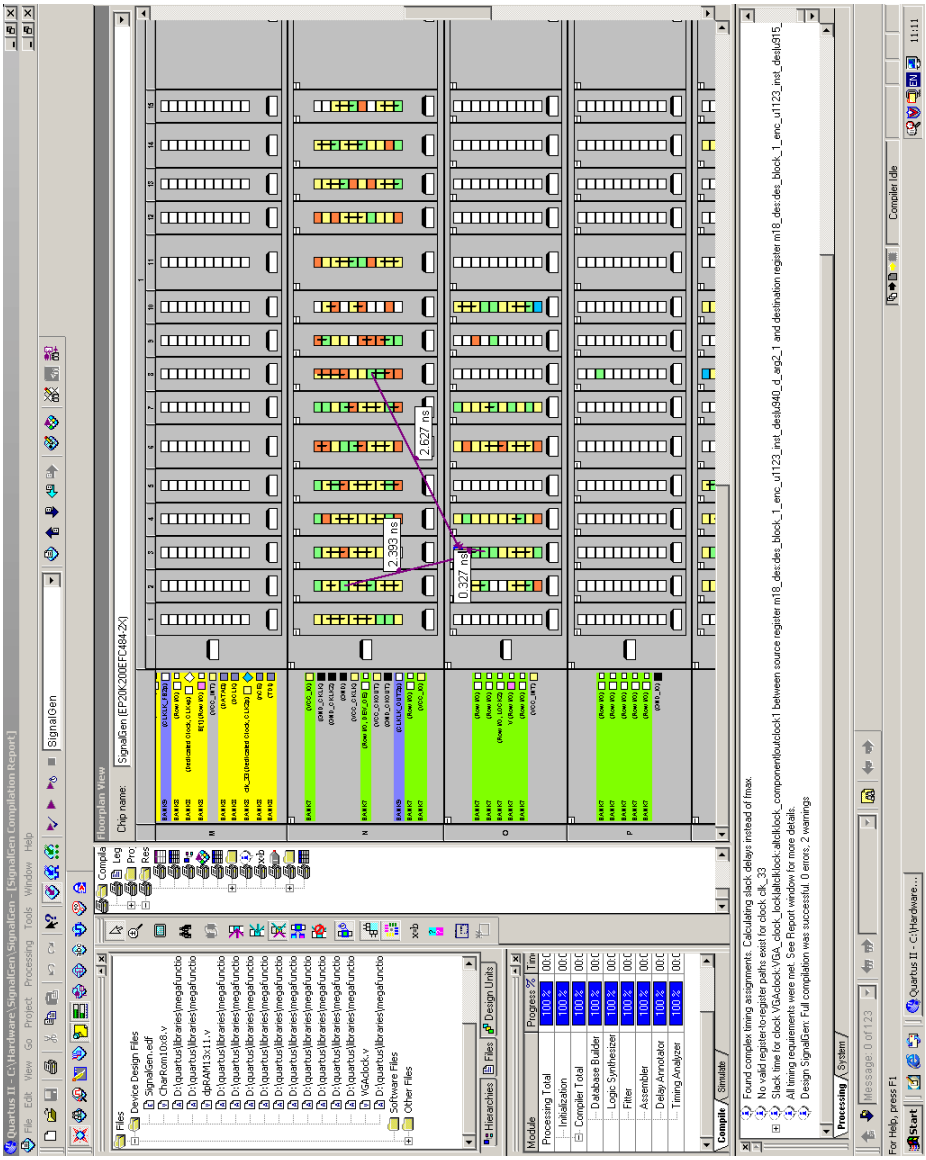


Fig. 10.3. Using the *Quartus II* package to map the netlist onto an Altera Apex-II FPGA

The Leonardo-generated netlist is loaded into *Quartus-II* which performs placement and routing analysis with respect to a specific FPGA. Figure 10.3 shows the floorplan of a DES circuit (which started as a SAFL specification) on an Altera APEX-II FPGA. The times in white boxes are displaying precise

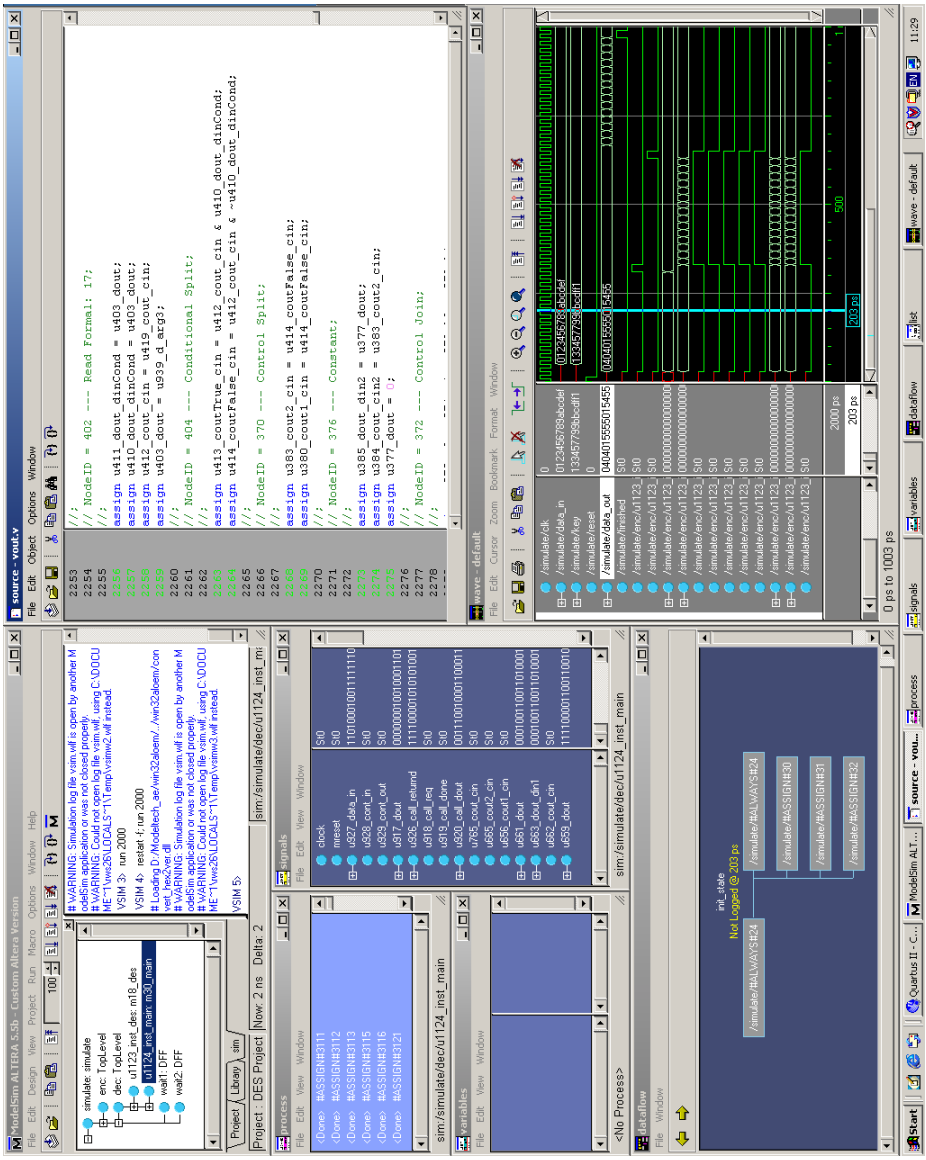


Fig. 10.4. Using the *ModelSim* package to simulate FLaSH-generated code at the RTL-level

routing delays. At this stage we know the exact value of the circuit's maximum clock frequency. Finally Quartus is used to load the design into an FPGA.

The FLaSH tool (Figure 10.1) provides a SAFL(+) interpreter which allows designers to check parts of their designs quickly. However, if lower level information is required (e.g. how many cycles will this take to compute?) or if the design

makes heavy use of `extern` functions, it is often useful to simulate the design at a lower-level (perhaps in conjunction with Verilog models representing the `extern` parts of the design). In this case we use FLaSH to compile the SAFL specification to RTL-Verilog and feed this into industrial hardware simulation software. Figure 10.4 shows a synchronous design generated by the FLaSH compiler being simulated in *ModelSim*.

10.2 DES Encrypter/Decrypter

Appendix Appendix A presents a SAFL specification of a Data Encryption Standard (DES) encryption/decryption circuit. Here we describe the code for the DES example; the details of the DES algorithms are not discussed in depth—we refer readers who are interested in knowing more about DES to Scheier's cryptography textbook [130].

The library block at the beginning of the DES specification defines three functions used later in the specification:

- `perm` is a curried function which takes a permutation pattern, p, (represented as a list of integers) and a list of bits, l. It returns l permuted according to pattern p.
- `ror` is a curried function which takes an integer, x, and a list of bits, l. It returns l rotated right by x.
- `rol` is as `ror` but rotates bits left (as opposed to right).

A set of permutation patterns required by the DES algorithm are also declared.

The DES algorithm requires 8 S-boxes, each of which is a substitution function which takes a 6-bit input and returns a 4-bit output. The S-boxes' definitions make use of one of SAFL's syntactic sugarings:

`lookup` e `with` $\{v_0$`, ...,` $v_k\}$

Semantically the `lookup` construct is equivalent to a `case` expression:

`case` e `of 0 =>` v_0 `| ...|` $(k-1)$ `=>` v_{k-1} `default` v_k.

To ensure that each input value to the `lookup` expression has a corresponding output value we enforce the constraint that $k = 2^w - 1$ where w is the width of expression e. Our compiler is often able to map `lookup` statements directly into ROM blocks, leading to a significantly more efficient implementation than a series of iterated tests.

Before applying its substitution each S-box permutes its input. We use our Magma permutation function to represent this permutation: `<% perm p_inSbox %>(x)`. Other examples of SAFL-Magma integration can be seen throughout the specification. The `keyshift` function makes use of the Magma `ror` and `rol` functions to generate a key schedule. Other invocations of the Magma `perm` function can be seen in the bodies of SAFL-level functions: `round` and `main`. We find the use of higher-order Magma functions (such as `perm`, `ror` and `rol`) to be a powerful idiom.

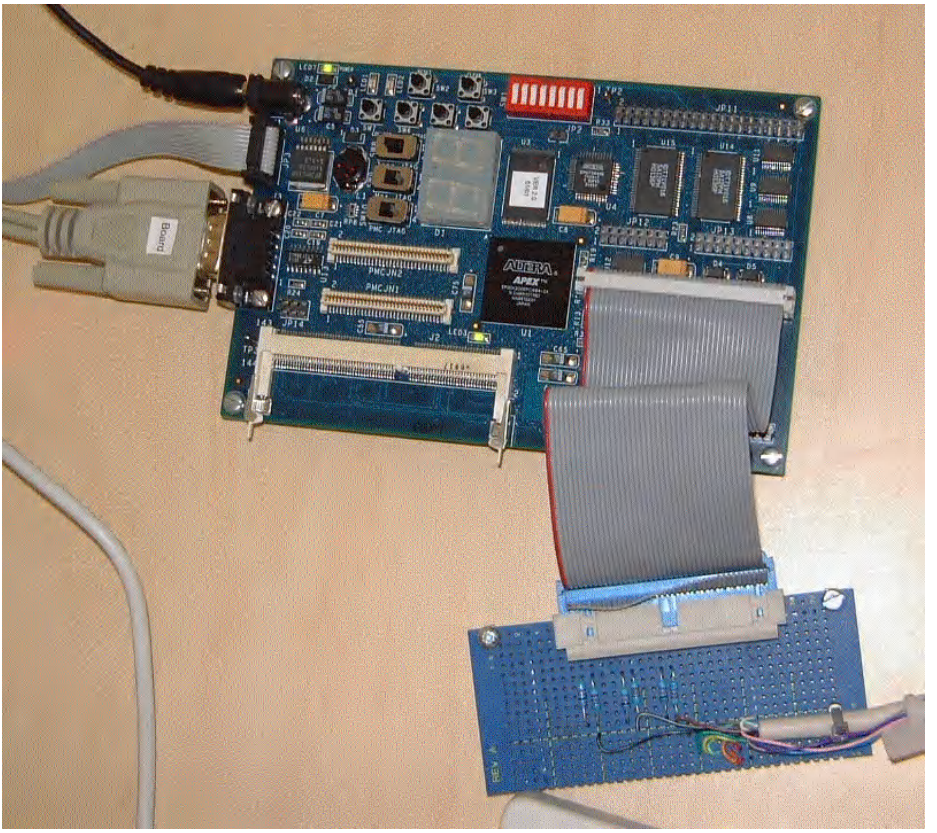


Fig. 10.5. The Altera "Excalibur" Development Board containing an Apex-II FPGA with our simple VGA interface connected via ribbon cable

We used our SAFL compiler to map the DES specification to synthesisable RTL-Verilog. A commercial RTL-synthesis tool (Leonardo from Exemplar) was used to synthesise the RTL-Verilog for an Altera Apex E20K200E FPGA (200K gate equivalent). The resulting circuit utilised 8% of the FPGA's resources and could be clocked up to 48MHz. The design was mapped onto an Altera Excalibur Development Board and, using the board's 33MHz reference clock a throughput of 15.8Mb/s (132 Mbits/s) was achieved. The performance figures of our DES implementation compare favourably to a hand-coded DES implementation written in VHDL by Kapps and Paar [82]. In practice our implementation runs 30% faster; however this is probably, at least in part, due to the fact that we are using different FPGA technology. A more meaningful comparison is to observe that both implementations take the same number of cycles to process a DES block.

Figure 10.5 shows the Altera Excalibur development board onto which we loaded our synthesised DES circuit. We wrote a synthesisable Verilog wrapper which instantiates and tests our SAFL-generated DES design. Two DES circuit

are instantiated in series: one which performs encryption and one which performs decryption. Data is continuously fed into the input of the first DES block and read back from the output of second. The input to the first DES block is compared with the output of the second and checked for equality. Two status LEDs on the development board are used: one blinks every time 1×10^6 data blocks are processed; the other comes on only if an error is detected (i.e. the output of the second DES circuit is not equal to the input of the first DES circuit).

10.2.1 Adding Hardware VGA Support

To extend the output capability of the DES circuit we designed a VGA interface for the development board and modified our SAFL program to drive a monitor, displaying the values of the DES input and output data on a VGA monitor. The addition of the VGA interface is more significant than merely making the circuit more impressive to demonstrate: it gives an example of a SAFL program interfacing with a timing-critical system (VGA output) and demonstrates the use of SAFL `extern` functions for interfacing with lower-level components. In the remainder of this section we briefly describe how the VGA interface was constructed and, more importantly, how we interfaced it to the SAFL part of the design.

VGA Interface: Low-Level Details

To control a VGA monitor one must drive 5 analog signals referred to as R, G, B, H and V. Signals R, G and B transmit the intensity of the Red, Green and Blue components of the current pixel (respectively) as an analog voltage between 0V (black) and 0.7V (maximum intensity). Our VGA interface supports 6-bit colour (2-bits for each of R, G and B) allowing us to use pairs of binary-weighted resistors as crude 2-bit DACs. Since the output pins of our FPGA are LVTTL (Low Voltage TTL: 0–3.3V) and a VGA monitor has a load-resistance of 75Ω we use resistances of 500Ω and $1k\Omega$ for our 2-bit resistor DACs.

At the hardware-level we built 3 such DACs and connected them via a ribbon cable to the Excalibur development board (see Figure 10.5). The other end of the circuit is connected to the R,G,B,H,V (and earth) wires of a standard VGA monitor cable. Signals H and V, which respectively supply periodic horizontal and vertical synchronisation pulses, are connected directly to two separate output pins of the FPGA[1]. Figure 10.5 shows the VGA interface connected to the Excalibur development board; Figure 10.6 shows the circuit driving a test image onto a VGA monitor.

A VGA driver, responsible for outputting pixel and synchronisation information from the FPGA, was implemented in RTL Verilog. To ensure the frame

[1] The VGA specification require that H and V signals must use TTL-levels. However, we are able to drive them directly using LVTTL levels since 3.3V is within the acceptable voltage range of a TTL logic-1. (It also helps that monitors are designed to handle noisy signals from cheap graphics cards.)

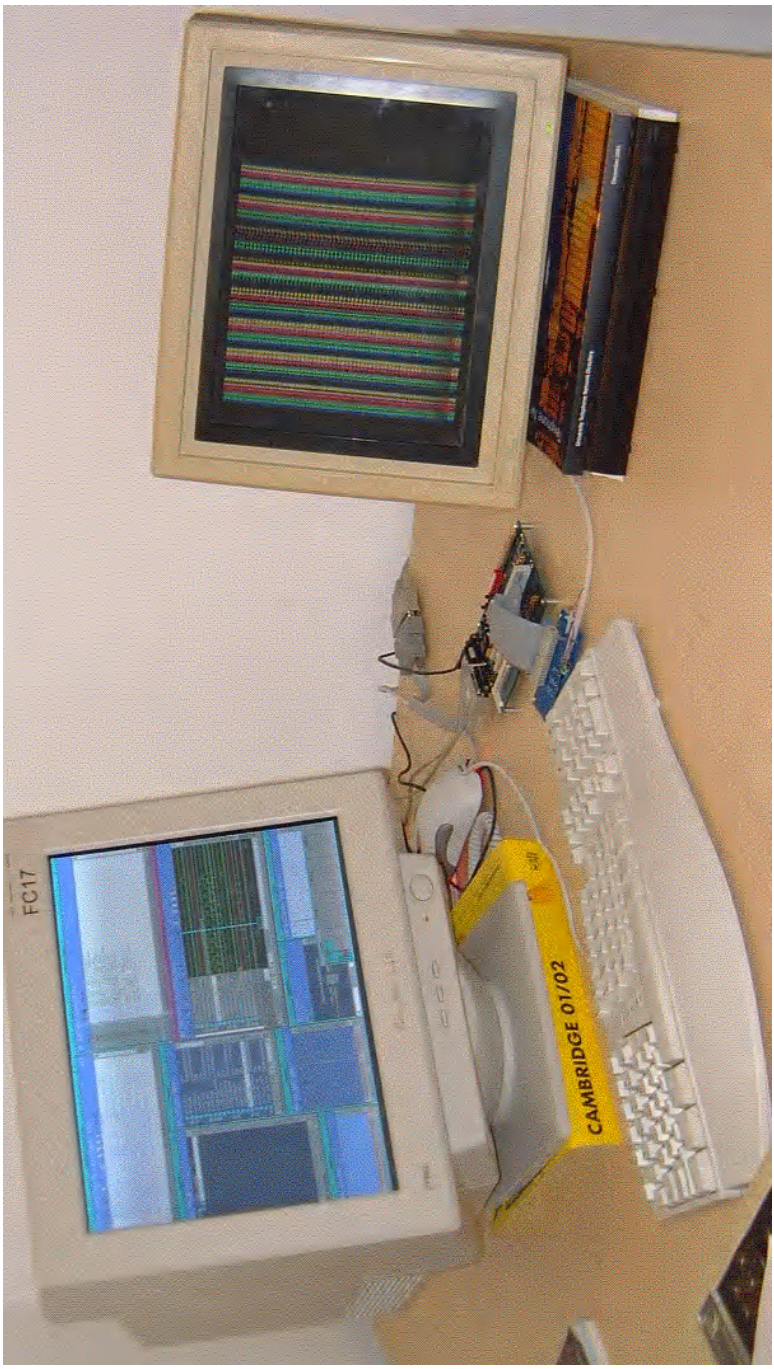


Fig. 10.6. The Altera Development Board driving a test image onto a VGA monitor

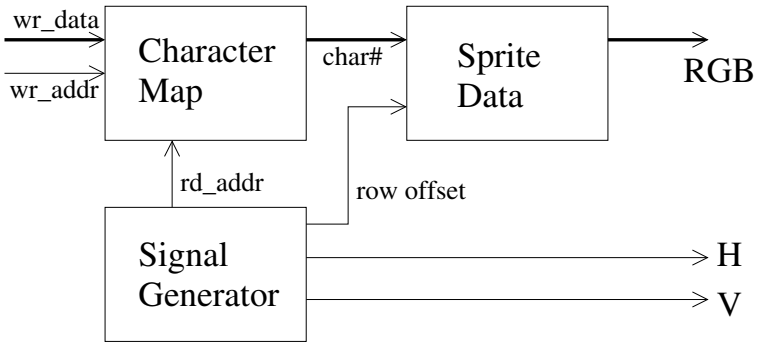


Fig. 10.7. The SAFL DES block connected to the VGA signal generation circuitry

buffer is small enough to fit in the FPGA[2] the screen is divided into 8×8 character squares and the frame buffer split into two parts accordingly:

- a character map (which stores the character code and colour for character square); and
- sprite data (which stores the 8×8 sprite representing a particular character).

The character map is implemented as a dual-ported RAM. One port is used to write new characters to the screen, the other port is used to continuously read character data for display purposes. The sprite data is stored in single-ported ROM: supplying a character number and row offset returns an 8-bit value representing the requested row (8 pixels) of the specified character. (Both the dual-ported RAM and the single-ported ROM are implemented on the FPGA by instantiating vendor supplied IP blocks.)

A separate *signal generator* is responsible for the generation of synchronisation signals (H,V) and also contains the control logic required to lookup character data and output pixel data (R,G,B) to the screen. Figure 10.7 shows a block diagram of the VGA interface. Input wires `wr_addr` and `wr_data` are used to write new characters to the screen: `wr_addr` specifies the character square to write to; `wr_data` specifies the character code and colour to write. The VGA circuit is pipelined so as the sprite-data for the previous character is being looked up and outputted, the next character code and colour are being read from the character map in parallel.

The VGA interface requires a pixel clock of 25.175 MHz. We provide this by using the FPGA's on-chip PLLs (as clock multipliers) and clock divide circuitry to generate the required frequency from the development board's 33.3 MHz reference clock.

[2] Unfortunately a straightforward 640x480 frame-buffer is too much for our Altera EP20K200 FPGA.

Interfacing SAFL to the Verilog VGA Interface

We start by declaring an `extern` function:

```
extern write_char (char_square, char_code, colour)
```

where `char_sqaure` is the number of the character square to write to (since we are using a 60×80 character display 0 is top left and 4799 is bottom right), `char_code` is a numerical value representing the character to write and `colour` is the 6-bit colour code (2 bits for each of R, G, B). The externally provided body of `write_char` (see Section 5.2.4) is the Verilog VGA circuitry described in the previous section. Formal parameters `char_square` and `char_code` map directly onto the write port of the character map, supplying values for `wr_addr` and `wr_data` respectively.

```
fun write_hex(v:64, char_square, colour):unit =
   static fun do_write(count, square, value) =
              write_char(square, value[63:60], colour);
              if count>0 then do_write(count-1, square+1, value<<4)
                         else ()
    in do_write(15, char_square, v)
   end
```

Fig. 10.8. The definition of function `write_hex`

Next we write a function which prints a 64-bit value as a 16 character hex string starting at a specified character square (see Figure 10.8). For the sake of simplicity, we arrange the first 16 character codes (numbered 0 to 15) to be the hex digits $0, 1, \ldots, F$ respectively.

We modify the DES program of Appendix Appendix A to display every 16^{th} (input,output) pair on the monitor whenever microswitch-7 (on the development board) is pressed. First we rename the `main` function as `compute_DES` and define a number of constants:

Constant	Meaning
`Encrypt`	Perform DES encryption [rather than decryption]: (Set to 1)
`Red`	Colour code for red
`Yellow`	Colour code for yellow
`Row_Length`	The number of character squares in a row: (Set to 80)
`Max_Square`	The character square beyond which vertical wrap occurs

Figure 10.9 gives the SAFL code which defines an external channel[3] (which is used to read whether microswitch-7 is currently depressed) and a new `main` function which stimulates the DES circuit, displaying the inputs and outputs on a monitor if switch-7 is pressed.

[3] Recall that external channels can be read without blocking.

```
channel external switch7

fun main(in_block, key, char_square) =
  let val switch_on = switch7?
      val des_result = compute_des(in_block, key, Encrypt)
   in if (switch_on and in_block[3:0]=0) then
          (write_hex(in_block, char_square, Red);
           write_hex(des_result, char_square + 32, Yellow);
           main(in_block+1, key+1,
                  if char_square>Max_Square then 0
                       else char_square+Row_Length))
      else main(in_block+1, key+1, char_square)
  end
```

Fig. 10.9. Displaying the DES circuits inputs and outputs on a monitor whenever a micro-switch is pressed

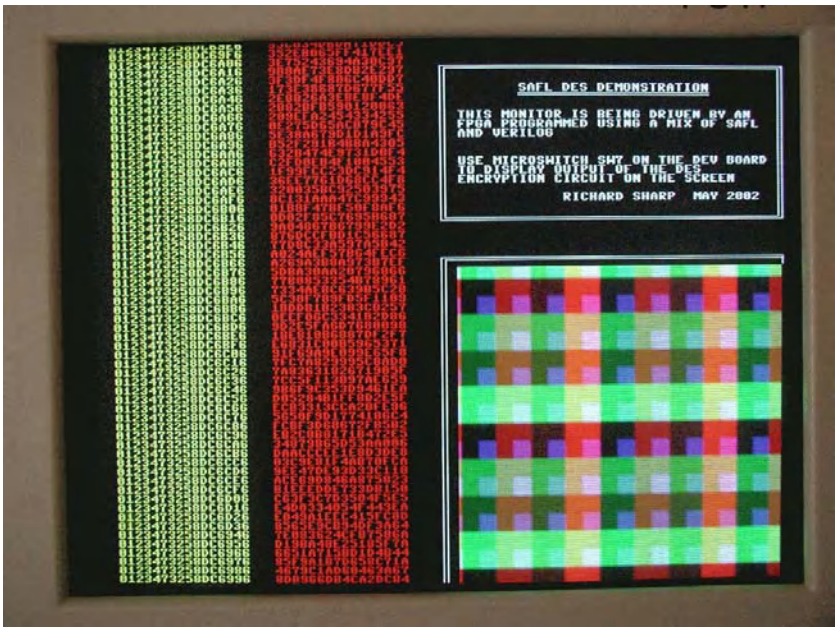


Fig. 10.10. A screenshot of the DES circuit displaying its inputs and outputs on a VGA monitor

Figure 10.10 shows a screenshot of the DES circuit in operation. The left column shows inputs to the DES block; the right column shows outputs to the DES block. The character map is initialised with data which represents the text shown on the screen. The animated pattern at the bottom right of the screen is integrated into the Verilog VGA signal generator; it was initially used for

debugging vertical synchronisation issues in an early version of the VGA driver and subsequently remained for aesthetic reasons!

A Note on GALS

It is worth revisiting the GALS issue at this stage as the DES/VGA design provides a good opportunity to demonstrate the benefits of supporting multiple clock domains in a high-level HDL.

We have already seen that the maximum clock frequency of the DES circuit is 48MHz, however the VGA driver requires a pixel clock of 25.175MHz. In our initial design, which contains only a single clock domain, we simply reduce the clock frequency of the DES circuit to match the 25.175MHz pixel clock.

The dual-ported RAM is a *clocked memory* which operates at a frequency of 25.175MHz; all read and write accesses to the character map must be synchronised with this clock. Thus we can partition our DES specification into two separate clock domains:

- *Clk_DES*: contains the DES functions; and
- *Clk_VGA*: contains the `extern` function `write_char` and the SAFL function `write_hex`

Even when the increased latency of calls to `write_hex` (incurred due to synchronisation delay when switching clock domains) is taken into account, the circuit as a whole still runs faster since DES blocks can now be computed nearly twice as fast (as the DES part of the circuit is now clocked at 48MHz as opposed to 25MHz).

10.3 Summary

We have presented our SAFL-to-silicon tool chain implementation with reference to a realistic example. There are two interesting points which we have demonstrated in this chapter which we feel are worth emphasising here:

1. SAFL designs (where precise timing information is not made explicit) can be interfaced with timing-critical systems by connecting them via a shared buffer. In this example we saw our SAFL DES circuit communicating with a timing-critical VGA system by means of a dual-ported RAM.
2. Mapping separate `extern` functions onto separate ports of a multi-ported memory provides a useful way to implement resource sharing without having to worry about scheduling. Since SAFL will treat separate `extern` functions independently no arbitration circuitry will be generated, even though in practice they access the same resource. Here we used this technique to bind the `extern` function `write_char` to one port of a shared dual-ported memory.

We believe that this chapter, in particular the favourable performance comparison between the SAFL-generated DES circuit and a hand-coded DES circuit

written by a third party [82] (see Section 10.2), demonstrates that generating hardware directly from the SAFL/SAFL+ languages is a viable technique. In previous chapters we have claimed that designing hardware in SAFL(+) offers a number of advantages; here we have actually managed to realise some of those advantages in practice. Not only is our SAFL DES specification is much shorter and easier to analyse/transform than a hand-coded RTL equivalent (see Section 10.2 and Appendix Appendix B) but it also yields a more efficient implementation in silicon.

11

Conclusions and Further Work

Recall from Chapter 1 that the thesis of this work is

> *that there is scope for higher-level Hardware Description Languages and, furthermore, that the development of such languages and associated tools will help to manage the increasing size and complexity of modern circuits.*

Let us start by considering the first part of the claim: "*that there is scope for higher-level Hardware Description Languages*". We justify this statement firstly by arguing that many existing HDLs are not as high-level as they could be, particularly in their choice of abstraction primitives (see Sections 1.3.1 and 3.3.4). Having made this point we go on to describe SAFL and argue that it is a *higher-level* language than existing behavioural HDLs (see Chapter 3).

The second part of the thesis, "*that the development of such languages and associated tools will help to manage the increasing size and complexity of modern circuits*", is justified with respect to our experiences of designing the SAFL(+) language and its associated tool chain:

- in Chapter 4 we present a scheduling technique which is more suited to large system-on-a-chip designs than existing scheduling methodologies;
- in Chapter 5 we show how SAFL can be compiled to hardware and, in particular, that its high-level properties allow it to be compiled to a variety of different design styles;
- in Chapter 6 we present global compiler analyses which rely on high-level properties of the SAFL language; and
- in Chapter 9 (and also in Sections 3.3.3 and 4.4) we describe source-to-source transformations which facilitate architectural exploration. Again, it is the high-level properties of SAFL which make such transformations possible.

However, "*the development of such languages and associated tools*" will only "*help to manage the increasing size and complexity of modern circuits*" if they offer practical implementations capable of generating efficient circuits. We justify that this is the case with reference to Chapters 7 and 8 (which extend SAFL with some of the capabilities one would expect in a realistic HDL) and Chapter 10 which illustrates our SAFL-to-silicon tool chain implementation by means of a realistic case study.

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 169–170, 2004.

11.1 Future Work

We designed the SAFL/SAFL+ languages solely to investigate different combinations of language features in high-level HDLs and, to this end, they served their purpose well. However, if the languages were to be used in anger by the hardware design community we believe that they would need to be extended. In particular, although SAFL+ contains many of the features one would expect in a behavioural HDL, it is still lacking a strong type system and a module system. We believe that the addition of both of these components is more an implementation project than a research project but nevertheless time consuming.

There are a number of other directions for future work which could help develop this research further:

- One area of particular interest is compiling SAFL(+) to a combination of different flavours of asynchronous hardware (e.g. compiling function internals to matched-delay circuits for efficiency whilst using Delay Insensitive (DI) interconnect for inter-function communication. We refer the interested reader to Davis and Nowick's survey of matched-delay and DI design techniques [40] for more information.)
- A transformation tool for SAFL(+) could be implemented to provide assistance in applying source-to-source transformations to SAFL specifications. Although there is already a great deal of work on developing transformation assistants [46, 126, 7, 118], research has so far been focused on the transformation of software-based systems; it would be beneficial to see whether there are any issues which should be addressed differently in the SAFL domain.
- It would also be beneficial to develop further case studies of designing hardware in the SAFL and SAFL+ languages. Our hope is that the experiences learnt from such experiments could be used to improve the design of the languages.

A number of researchers are investigating other ways of extending the research presented in this monograph. Edwards *et al* used a compilation scheme based on ours (see Chapter 5) to convert a subset of C into a netlist and experimented with feeding the resulting circuit into a standard hardware model checker [134]. Frankau is currently investigating how lazy lists and algebraic datatypes can be integrated into the SAFL framework whilst maintaining static allocation [49, 48]. The aim is provide a pure-functional alternative to SAFL+'s synchronous channels.

Appendix A

DES Encryption/Decryption Circuit

```
(* Start of Magma Library Block -------------------------------- *)
<%
signature DES =
  sig
     val perm: int list -> 'a list -> 'a list
     val ror:  int -> 'a list -> 'a list
     val rol:  int -> 'a list -> 'a list
     val p_initial   : int list
     val p_key       : int list
     val p_compress  : int list
     val p_expansion : int list
     val p_pbox      : int list
     val p_final     : int list
     val p_inSbox    : int list
  end

functor Magma_code (B:BASIS):DES =
 struct

 (* DES permutation patterns. *)

  val p_initial   = [58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
                     62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
                     57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
                     61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7]

  val p_key       = [57,49,41,33,25,17,9,1,58,50,42,34,26,18,
                     10,2,59,51,43,35,27,19,11,3,60,52,44,36,
                     63,55,47,39,31,23,15,7,62,54,46,38,30,22,
                     14,6,61,53,45,37,29,21,13,5,28,20,12,4]

  val p_compress  = [14,17,11,24,1,5,3,28,15,6,21,10,
                     23,19,12,4,26,8,16,7,27,20,13,2,
                     41,52,31,37,47,55,30,40,51,45,33,48,
                     44,49,39,56,34,53,46,42,50,36,29,32]
```

R. Sharp: Higher-Level Hardware Synthesis, LNCS 2963, pp. 171–183, 2004.

```
    val p_expansion = [32,1,2,3,4,5,4,5,6,7,8,9,
                       8,9,10,11,12,13,12,13,14,15,16,17,
                       16,17,18,19,20,21,20,21,22,23,24,25,
                       24,25,26,27,28,29,28,29,30,31,32,1]

    val p_pbox      = [16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,
                       2,8,24,14,32,27,3,9,19,13,30,6,22,11,4,25]

    val p_final     = [40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,
                       38,6,46,14,54,22,62,30,37,5,45,13,53,21,61,29,
                       36,4,44,12,52,20,60,28,35,3,43,11,51,19,59,27,
                       34,2,42,10,50,18,58,26,33,1,41,9,49,17,57,25]

    val p_inSbox    = [1,5,2,3,4]

    (* Permutation function -- given a permutation pattern
                               (list of ints)
                               and a list of bits it returns a
                               permuted list of bits: *)

    fun perm positions input =
        let val inlength = length input
            fun do_perm [] _ = []
              | do_perm (p::ps) input =
                 (List.nth (input,inlength-p))::(do_perm ps input)
        in do_perm positions input
        end

    (* Rotate bits right by specified amount: *)
    fun ror n l =
        let val last_n = rev (List.take (rev l, n))
            val rest   = List.take (l, (length l)-n)
         in last_n @ rest
        end

    (* Rotate bits left by specified amount: *)
    fun rol n l =
        let val first_n = List.take (l, n)
            val rest    = List.drop (l, n)
        in rest @ first_n
        end
  end
  %>
  (* End of Magma Library Block ---------------------------------- *)

  (* Definitions of S-Boxes (implemented as simple lookup tables).
     The 'inline' pragma tells the compiler to inline each call to
     a function rather than treating it as a shared resource. We use
     inline here because the resources as so small they are not
     worth sharing.  *)
```

```
inline fun sbox1(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,
                 0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
                 4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
                 15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}

fun sbox2(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,
                 0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
                 4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
                 15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}:4

fun sbox3(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,
                 13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
                 13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,
                 1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12}:4

fun sbox4(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,
                 13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
                 10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,
                 3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14}:4

fun sbox5(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,
                 14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
                 4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,
                 11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3}:4

fun sbox6(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,
                 10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
                 9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,
                 4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13}:4


fun sbox7(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,
                 13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
                 1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,
                 6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12}:4
```

```
fun sbox8(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,
                 1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
                 7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
                 2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11}:4

(* Do s_box substitution on data-block: *)

fun s_sub(x:48):32 =
  join( sbox1( x[47:42] ), sbox2( x[41:36] ),
        sbox3( x[35:30] ), sbox4( x[29:24] ),
        sbox5( x[23:18] ), sbox6( x[17:12] ),
        sbox7( x[11:6]  ), sbox8( x[5:0]   ))

(* Define a record which contains the left and right halves
   of a 64-bit DES block and the 56-bit key. *)

type round_data = record {left:32, right:32, key:56}

(*  A single DES round:  *)

fun round(bl:round_data,rd:4,encrypt:1):round_data =

  static

    (*  Successive keys are calculated by circular shifts.
        The degree of the shift depends on the round (rd).
        We shift either left/right depending on whether we are
        decrypting/encrypting. *)

     fun keyshift(key_half:28,rd:4,encrypt:1):28 =
          define val shift_one = (rd=0 or rd=1 or rd=8 or rd=15)
          in
             if encrypt then if shift_one then <% rol 1 %> (key_half)
                             else <% rol 2 %> (key_half)
             else if rd=0 then key_half
                      else if shift_one then <% ror 1 %> (key_half)
                           else <% ror 2 %> (key_half)
          end
  in

      let
         val lkey = keyshift(slice(bl.key,55,28),rd,encrypt)
         val rkey = keyshift(slice(bl.key,27,0),rd,encrypt)
         val keybits   = <% perm p_compress %> ( join(lkey,rkey) )
         val new_right =
                let val after_p = <% perm p_expansion %>(bl.right)
                 in s_sub (after_p xor keybits xor bl.left)
```

```
                end
      in {left=bl.right, right=new_right, key=join(lkey,rkey)}
      end
  end

(*  Do 16 DES rounds: *)

fun des(c:4, rd:round_data,encrypt:1):round_data =
   let val new_data = round(rd, c, encrypt)
    in if c=15 then new_data
               else des(c+1, new_data,encrypt)
   end

(*  Apply initial permuations to DES block and key: *)

fun initial_perms (rd:round_data):round_data =
   let val new_block = <% perm p_initial %> (join(rd.left,
                                                  rd.right))
       val new_key = <% perm p_key %> (rd.key)
    in {left  = slice(new_block,63,32),
        right = slice(new_block,31,0),
        key   = new_key}
    end

(*  Do input/output permutations and 16 rounds of DES: *)

fun main(rd:round_data, encrypt:1):round_data =
  let
        val perm_rd = initial_perms (rd)
        val output = des(0:4, perm_rd, encrypt)
   in <% perm final %> (join(output.right, output.left))
  end
```

Appendix B

Transformations to Pipeline DES

In this section we apply a series of transformations to the DES specification in Appendix Appendix A in order to transform it into a 4-stage pipelined version (where each pipeline stage performs 4 rounds of encryption). To reduce the amount of code that needs to be written we choose to specify only a DES encryption circuit (i.e. we partially evaluate the `des` function from Appendix Appendix A with encrypt as a static parameter set to `1`).

We start by using basic equational reasoning steps and fold/unfold transformations to manipulate the DES specification into the form required by our top-level pipelining transformation (see Section 9.3).

Unfolding the recursive call in the `des` function 15 times and using the *instantiation* rule to set the formal parameter `c` to 0, yields a function `des_c0`:

```
fun des_c0(rd:round_data):round_data =
  let    val nd1  = round(rd,   0)
         ---
         val nd2  = round(nd1,  1)
         ---
         val nd3  = round(nd2,  2)
         ---
         val nd4  = round(nd3,  3)
         ---
         val nd5  = round(nd4,  4)
         ---
         val nd6  = round(nd5,  5)
         ---
         val nd7  = round(nd6,  6)
         ---
         val nd8  = round(nd7,  7)
         ---
         val nd9  = round(nd8,  8)
         ---
         val nd10 = round(nd9,  9)
```

```
            ---
            val nd11 = round(nd10, 10)
            ---
            val nd12 = round(nd11, 11)
            ---
            val nd13 = round(nd12, 12)
            ---
            val nd14 = round(nd13, 13)
            ---
            val nd15 = round(nd14, 14)
            ---
            val nd16 = round(nd15, 15)
    in
        nd16
    end
```

We replace the call, `des(0, ...)`, in the `main` function with a call `des_c0(...)` and group the calls to `round` into blocks of 4 by repeatedly replacing bindings with their declarations:

```
fun des_c0(rd:round_data):round_data =
     let val nd4  = round(round(round(round(rd,  0),
                                              1),
                                              2),
                                              3)
         ---
         val nd8  = round(round(round(round(nd4, 4),
                                              5),
                                              6),
                                              7)
         ---
         val nd12 = round(round(round(round(nd8, 8),
                                              9),
                                              10),
                                              11)
         ---
         val nd16 = round(round(round(round(nd12,12),
                                              13),
                                              14),
                                              15)
    in
        nd16
    end
```

We now introduce a new function `round_CtoN` which has a copy of the `round` function definition nested within it:

```
fun round_CtoN (c:4, n:4, rd:round_data):round_data =
   static fun round(bl:round_data,rd:4,encrypt:1):round_data =
            ... <<body of round ommitted to save space>> ...
   in
      let val new_data = round(rd)
      if c=n then new_data
             else round_CtoN(c+1, n, new_data)
      end
   end
```

and transform `des_c0` into:

```
fun des_c0(rd:round_data):round_data =
     let val nd4  = round_CtoN(0,3,rd)
         ---
         val nd8  = round_CtoN(4,7,nd4)
         ---
         val nd12 = round_CtoN(8,11,nd8)
         ---
         val nd16 = round_CtoN(12,15,nd12)
     in
         nd16
     end
```

Next we duplicate the function definition `round_CtoN` 4 times as `pipe_stage1`, `pipe_stage2`, `pipe_stage3` and `pipe_stage4`. (Note that since the `keyshift` and `round` functions are defined within `round_CtoN` these are implicitly duplicated at the same time.) We then Transform `des_c0` into:

```
fun des_c0(rd:round_data):round_data =
     let val nd4  = pipe_stage1(0,3,rd)
         ---
         val nd8  = pipe_stage2(4,7,nd4)
         ---
         val nd12 = pipe_stage3(8,11,nd8)
         ---
         val nd16 = pipe_stage4(12,15,nd12)
     in
         nd16
     end
```

and inline the call to `des_c0` in `main`:

```
fun main(rd:round_data, encrypt:1):round_data =
  let
         val perm_rd = initial_perms (rd)
         val output = let val nd4  = pipe_stage1(0,3,perm_rd)
                          ---
                          val nd8  = pipe_stage2(4,7,nd4)
```

```
                        ---
                        val nd12 = pipe_stage3(8,11,nd8)
                        ---
                        val nd16 = pipe_stage4(12,15,nd12)
                    in
                        nd16
                    end
 in <% perm final %> (join(output.right, output.left))
end
```

Collapsing the nested `let` declarations yields a function which is in a form to which we can apply our pipelining transformation (see Section 9.3):

```
fun main(rd:round_data, encrypt:1):round_data =
  let
        val perm_rd = initial_perms (rd)
        ---
        val nd4  = pipe_stage1(0,3,perm_rd)
        ---
        val nd8  = pipe_stage2(4,7,nd4)
        ---
        val nd12 = pipe_stage3(8,11,nd8)
        ---
        val output = pipe_stage4(12,15,nd12)
 in <% perm final %> (join(output.right, output.left))
end
```

Finally, applying our pipeline transformation to top-level function, `main`, yields:

```
fun main(rd, perm_rd, nd4, nd8, nd12, output)[in,out] =
  let
      val new_rd = in?

      val new_perm_rd = initial_perms(rd)
      val new_nd4     = pipe_stage1(0,3,perm_rd)
      val new_nd8     = pipe_stage2(4,7,nd4)
      val new_nd12    = pipe_stage3(8,11,nd8)
      val new_output  = pipe_stage4(12,15,nd12)

      val _ = out ! <% perm final %>
                        (join(output.right,output.left))
   in
      pipe_des(new_rd, new_perm_rd, new_d4, new_d8,
               new_nd12, new_output)
  end
```

Appendix C

A Simple Stack Machine and Instruction Memory

```
(* --------------------------- ALU ------------------------------ *)

fun alu2(op:16, a1:16, a2:16):16 =
   case op of 0 => a1+a2
            | 1 => a1-a2
            | 2 => and(a1,a2)
            | 3 => or(a1,a2)
            | 4 => xor(a1,a2)
            | 16 => a1<a2
            | 17 => a1>a2
            | 18 => a1=a2
            | 19 => a1>=a2
            | 20 => a1<=a2
            | 21 => a1<>a2

(* ---------------- Instruction memory --------------------------- *)

(* The following codes: f(x) = if x then x+f(x-1) else 0;   *)
(* i.e. it computes triangular numbers                      *)
fun load_instruction (address:16):24 = case address of
       0 => %000010010000000000000001 (*     pusha 1    *)
     | 1 => %000001010000000000000011 (*     call_int f *)
     | 2 => %000000000000000000000000 (*     halt       *)
     | 3 => %000000100000000000000001 (* f:  pushv 1    *)
     | 4 => %000001110000000000001100 (*     jz l1      *)
     | 5 => %000000100000000000000001 (*     pushv 1    *)
     | 6 => %000000100000000000000010 (*     pushv 2    *)
     | 7 => %000000010000000000000001 (*     pushc 1    *)
     | 8 => %000010000000000000000001 (*     alu2 sub   *)
     | 9 => %000001010000000000000011 (*     call_int f *)
     | 10=> %000010000000000000000000 (*     alu2 add   *)
     | 11=> %000001100000000000001101 (*     jmp l2     *)
     | 12=> %000000010000000000000000 (* l1: pushc 0    *)
     | 13=> %000001000000000000000001 (* l2: return 1   *)
```

```
 default => %10101010101010101010101 (*     illop      *)

external mem_acc (address:16,data:16,write:1):16

inline fun data_read  (address:16):16 = mem_acc(address,0,0)
inline fun data_write (address:16,data:16):16 =
                                      mem_acc(address,data,1)

fun SMachine (a1:16, PC:16, SP:16):16 =
   let var new_PC  : 16  = PC + 1
       var instr   : 24  = load_instruction(PC)
       var op_code : 8   = instr[23,16]
       var op_rand : 16  = instr[15,0]
       var inc_SP  : 16  = SP + 1
       var dec_SP  : 16  = SP - 1
    in case op_code of
            0 => (* halt, returning TOS *)
                 data_read(SP)

          | 1 => (* push constant operation *)
                 data_write(dec_SP, op_rand);
                 SMachine (a1, new_PC, dec_SP)

          | 2 => (* push variable operation *)
                 let var data:16 = data_read(SP+op_rand)
                 in  data_write(dec_SP, data);
                     SMachine (a1, new_PC, dec_SP)
                 end

          | 9 => (* push a-argument operation *)
                 data_write(dec_SP, a1);
                 SMachine (a1, new_PC, dec_SP)

          | 3 => (* squeeze operation --
                    op_rand is how many locals to pop *)
                 let var new_SP:16 = SP + op_rand
                     var v:16 = data_read(SP)
                 in  data_write(new_SP, v);
                     SMachine (a1, new_PC, new_SP)
                 end

          | 4 => (* return operation --
                    op_rand is how many actuals to pop *)
                 let var new_SP:16 = inc_SP + op_rand
                     var rv:16 = data_read(SP)
                 in  let var rl:16 = data_read(inc_SP)
                     in  data_write(new_SP, rv);
                         SMachine (a1, rl, new_SP)
                     end
                 end
```

```
        | 5 => (* call_int operation *)
                  data_write(dec_SP, new_PC);
                  SMachine (a1, op_rand, dec_SP)

        | 6 => (* jmp (abs) operation *)
                  SMachine (a1, op_rand, SP)

        | 7 => (* jz (abs) operation *)
                  let var v:16 = data_read(SP)
                  in  SMachine (a1, if v=0 then op_rand
                                           else new_PC, inc_SP)
                  end

        | 8 => (* alu2: binary alu operation --
                     specified by immediate field *)
                  let var v2:16 = data_read(SP)
                  in  let var v1:16 = data_read(inc_SP)
                      in  data_write(inc_SP, alu2(op_rand, v1, v2));
                          SMachine (a1, new_PC, inc_SP)
                      end
                  end

      default => (* halt, returning 0xffff -- illegal opcode *)
                  %1111111111111111
end
```

References

1. The national technology roadmap for semiconductors. Semiconductor Industry Association, 1999. Available from: SEMATECH, 3101 Industrial Terrace Suite 106 Austin TX 78758.
2. Handel-C language datasheet. Available from Celoxica Ltd: `http://www.celoxica.com/`.
3. Haskell98 report. Available from `http://www.haskell.org/`.
4. PHP hypertext preprocessor. See `http://www.php.net/`.
5. Afred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
6. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
7. Marco Aldinucci. The meta transformation tool for skeleton-based languages. In *Proceedings of the 2nd International Workshop on Constructive Methods for Parallel Programming (CMPP)*, 2000. Available from: `http://citeseer.nj.nec.com/486282.html`.
8. J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Proceedings of the International Symposium on Static Analysis*, volume 1694 of *LNCS*. Springer-Verlag, 1999.
9. American National Standards Institute, Inc. *The Programming Language ADA Reference Manual*. Springer-Verlag, 1983.
10. A. Appel. *Modern Compiler Implementation in Java/ML/C*. Cambridge University Press, 1998.
11. A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528, pages 1–13. Springer-Verlag, 1991.
12. Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro (Special Issue on Modeling and Validation of Microprocessors*, May/June 1999.
13. B. Preas and M. Lorenzetti. *Physical Design Automation of VLSI-Systems*. Benjamin Cummings, 1989.
14. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

15. J. Backus. Can functional programming be liberated from the von Neumann style? *Communications of the ACM*, 21(8):613–641, 1978.
16. J. Backus. The algebra of functional programs: Function level reasoning, linear equations and extended definitions. In *Proceedings of the Symposium on Functional Languages and Computer Architecture*, June 1981.
17. F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, June 1997.
18. Mario R. Barbacci and Daniel P. Siewiorek. Automated exploration of the design space for register transfer (RT) systems. In *Proceedings of the 1st Annual Symposium on Computer Architecture (ISCA)*, pages 101–106, 1973.
19. A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Proceedings of the Forum on Design Languages*, 2000. Available on request from European Electronic Chips and Systems design Initiative (ECSI).
20. G. Berry. Real time programming: special purpose or general purpose langages. Technical Report RR-1065, INRIA, August 1989.
21. G. Berry. A hardware implementation of pure esterel. *SADHANA – Academy Proceedings in Engineering Sciences*, 17:95–130, March 1992.
22. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
23. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. Technical Report 842, INRIA, 1988.
24. G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–98. Charleston, South Carolina, 1993.
25. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware description in Haskell. In *Proceedings of the 3rd International Conference on Functional Programming*, SIGPLAN. ACM, 1998.
26. B. Bose. DDD: A transformation system for digital design derivation. Technical Report 331, Indiana University, 1991.
27. B. Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Indiana University, 1994.
28. R. Brayton, R. Camposano, G. De Micheli, R. Otten, and J. van Eijndhoven. *The Yorktown Silicon Compiler System*. Addison-Wesley, 1988.
29. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. In *JACM 24(1)*, 1977.
30. L. Cardelli. The functional abstract machine. Technical Report TR-107, AT&T Bell Laboratories, April 1983.
31. C.A.R.Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
32. R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, SIGPLAN. ACM, 1991.
33. Gregory J. Chaitin. Register allocation spilling via graph coloring. In *SIGPLAN Symposium on Compiler Construction*, pages 98–105, 1982.
34. D. Chapiro. *Globally-Asynchronous Locally-Synchronous systems*. PhD thesis, University of Stanford, 1984.

35. P. Chou, R. Ortega, and G. Borriello. The Chinook hardware/software co-synthesis system. In *Proceedings of the 8th International Symposium on System Synthesis*, 1995.
36. C. Clack and S. L. Peyton-Jones. Strictness analysis – a practical approach. In *Functional Languages and Computer Architecture*, LNCS, pages 35–49. Springer-Verlag, 1985.
37. Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, pages 62–73, 1999.
38. Koen Classen, Mary Sheeran, and Stanam Singh. The design and verification of a sorter core. In *Proceedings of the 11th Advanced Working Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 355–369. Springer-Verlag, 2001.
39. B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors with Hawk. In *Proceedings of the workshop on formal techniques for hardware*, June 1998.
40. A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, University of Utah, September 1997.
41. G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc., 1994.
42. G. De Micheli and D. Ku. HERCULES - a system for high-level synthesis. In *Proceedings of the Design Automation Conference*, pages 483–488. ACM Press, June 1988.
43. G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus synthesis system for digital design. *Design & Test of Computers*, October 1990.
44. D. Edwards and A. Bardsley. Balsa 3.0 user manual. Available from `http://www.cs.man.ac.uk/amulet/projects/balsa/`.
45. C. Famsworth, D.A. Edwards, J. Liu, and S.S. Sikand. A hybrid asynchronous system design environment. In *Proceedings of the Second Working Conference on Asynchronous Design Methodologies (ASYNC 95)*. IEEE.
46. Martin Feather. *A system for developing programs by transformation*. PhD thesis, University of Edinburgh, 1979.
47. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
48. Simon Frankau and Alan Mycroft. Stream processing hardware from functional language specifications. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS)*. IEEE Computer Society Press, January 2003.
49. Simon Frankau, Alan Mycroft, and Simon Moore. Statically-allocated languages for hardware stream processing (extended abstract). In Sambuddhi Hettiaratchi, editor, *UK ACM SIGDA Workshop on Electronic Design Automation*. UK ACM SIGDA, Bournemouth University, September 2002.
50. T.D. Friedman and S.C. Yang. Methods used in an automatic logic design generator (ALERT). *IEEE Transactions in Computing*, C-18:593–614, 1969.
51. S. Furber, D. Edwards, and J. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. pages 329–334. IEEE, 2000.
52. Daniel Gajski, T. Ishii, V. Chaiyukul, H. Juan, and T. Hadley. A design methodology and environment for interactive behavioural synthesis. Technical Report 96-26, Department of Information and Computer Science, University of California, Irvine, June 1996.

53. D.D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *Design & Test of Computers*, 11(4), 1994.
54. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, September 2000. ISSN 0038-0644.
55. E.F. Girczcy. *Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Descriptions*. PhD thesis, Carleton University, Ottowa, Canada, July 1984.
56. Lance Glasser and Daniel Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.
57. C.K. Gomard and P. Sestoft. Globalization and live variables. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 166–177. ACM Press, 1991.
58. P. Le Guernic, M. Le Borgne, T. Gautier, and C. Le maire. Programming real time applications with Signal. *Proceedings of the IEEE*, 79:1321–1336, September 1991.
59. S. Guo and W. Luk. Compiling Ruby into FPGAs. In *Field Programmable Logic and Applications*, volume 975 of *LNCS*, pages 188–197. Springer-Verlag, 1995.
60. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Press, 1993.
61. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1321, September 1991.
62. N. Halbwachs, A. Lonchampt, and D. Pilaud. Describing and designing circuits by means of the synchronous data-flow programming language Lustre. In IFIP Working Conference: From HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, September 1986.
63. K. Hammond. Parallel functional programming: An introduction. In *Proceedings of the International Symposium on Parallel Symbolic Computation*. World Scientific, 1994.
64. D. Harel and A. Pnueli. On the development of reactive systems. In K.R.Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI*. Springer-Verlag, 1985.
65. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
66. Paul Havlak. Construction of thinned gated single-assignment form. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pages 477–499. Springer-Verlag, 1993.
67. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.
68. James Hoe, Martin Rinard, and Arvind. An exercise in high-level architectural description using a synthesizable subset of term rewriting systems, 1997. Computation Structures Group Memo 403.
69. J.C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proceedings of X IFIP International Conference on VLSI*, 1999.
70. M. Hofmann. A type system for bounded space and functional in-place update. In *Proceedings of the 9th European Symposium On Programming*, LNCS. Springer-Verlag, 2000.
71. J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. *Proceedings of the International Conference on Functional Programming (ICFP)*, 34(9):70–81, 1999.

72. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages*, pages 410–423, 1996.
73. IEEE. Standard VHDL Reference Manual, 1993. IEEE Standard 1076-1993.
74. IEEE. Verilog HDL language reference manual. IEEE Draft Standard 1364, October 1995.
75. IEEE. Standard for VHDL Register Transfer Level (RTL) Synthesis, 1999. IEEE Standard 1076.6-1999.
76. Inmos (Ltd.). *Occam 2 Reference Manual*. Prentice Hall, 1998.
77. Wolfgang Fichtner Jens Muttersbach, Thomas Villiger. Practical design of globally-asynchronous locally-synchronous systems. In *6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Press, 2000.
78. E.L. Johnson and G.L. Nemhauser M.W.P. Savelsbergh. Progress in integer linear programming: an exposition. Technical Report LEC-97-02, Georgia Institute of Technology, School of Industrial and Systems Engineering, January 1997.
79. S.D. Johnson and B. Bose. DDD: A system for mechanized digital design derivation. Technical Report 323, Indiana University, 1990.
80. G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI design*, pages 13–70. North-Holland, 1990.
81. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
82. Jens-Peter Kaps and Christof Paar. Fast DES implementation for FPGAs and its application to a universal key-search machine. In *Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 234–247. Springer-Verlag, 1998. URL `citeseer.nj.nec.com/119314.html`.
83. B. Kernighan and D. Ritchie. *The C Programming Language. Second Edition*. Prentice Hall, 1988.
84. Joep Kessels, Kees van Berkel, Ronan Burgess, Marly Roncken, and Frits Schalij. An error decoder for the compact disc player as an example of VLSI programming. In *Proc. European Conference on Design Automation (EDAC)*, pages 69–74. IEEE, 1992.
85. Anne T. Kohlstaedt. Daisy 1.0 reference manual. Technical Report 119, Indiana University Computer Science Department, 1981.
86. D. Ku and G. De Micheli. HardwareC—a language for hardware design (version 2.0). Technical Report CSL-TR-90-419, Stanford University, 1990.
87. D. Ku and G. De Micheli. High-level synthesis and optimization strategies in Hercules and Hebe. In *Proceedings of the European ASIC Conference*, pages 124–129, May 1990.
88. D. Ku and G. De Micheli. Constrained resource sharing and conflict resolution in Hebe. *Integration—The VLSI Journal*, December 1991.
89. D. Ku and G. De Micheli. Relative scheduling under timing constraints: Algorithm for high-level synthesis of digital circuits. *Transactions on CAD/ICAS*, pages 697–718, June 1992.
90. P. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6 (4):308–320, 1964.
91. R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, Boston, MA, 1992.
92. H. De Man, J. Rabaey, P. Six, and L. Claesen. Cathedral-II: A silicon compiler for digital signal processing. *Design & Test of Computers*, December 1986.

93. G. Marchioro, J. Daveau, and A. Jerraya. Transformation partitioning for co-design of multiprocessor systems. In *Proccedings of the International Conference on Computer Aided Design (ICCAD)*. IEEE, 1997.
94. S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN. ACM, 2001.
95. P. Marwedel. A new synthesis algorithm for the mimola software system. In *Proceedings of the 23rd Design Automation Conference*, pages 271–277. ACM Press, 1986.
96. J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Proceedings of the IEEE International Conference on Computer Languages*, 1998.
97. M. McFarland. Using bottom-up design techniques in the synthesis of hardware from abstract behavioral descriptions. In *Proceedings of the 23rd Design Automation Conference*, pages 474–480. ACM Press, 1986.
98. M. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2), February 1990.
99. R. Milner. A theory of type-polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.
100. R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.
101. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
102. S.W. Moore, G.S. Taylor, P.A. Cunningham, R.D. Mullins, and P.Robinson. Using stoppable clocks to safely interface asynchronous and synchronous subsystems. In *Proceedings of the AINT (Asynchronous INTerfaces) Workshop*, July 2000.
103. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.
104. A. Mycroft and R. Sharp. Higher-level techniques for hardware description and synthesis. *Software Tools for Technology Transfer (STTT)*. To Appear.
105. A. Mycroft and R.W. Sharp. A statically allocated parallel functional language. In *Proceedings of the International Conference on Automata, Languages and Programming*, volume 1853 of *LNCS*. Springer-Verlag, 2000.
106. A. Mycroft and R.W. Sharp. Hardware/software co-design using functional languages. In *Proceedings of TACAS*, volume 2031 of *LNCS*. Springer-Verlag, 2001.
107. T. Nijhar and A. Brown. Source-level optimisation of VHDL for behavioural synthesis. *Proceedings on Computers and Digital Techniques*, 144(1):1–6, January 1997.
108. Rishiyur Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann, May 2001.
109. John O'Donnell. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382. North-Holland, April 1987.
110. John O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming, Workshops in Computing, Proceedings*, pages 178–194. Springer-Verlag, 1992.
111. S. Olcoz and J. Colom. Towards a formal semantics of IEEE std. VHDL 1076. In *Proceedings of 1993 European Design Automation Conference with Euro-VHDL*. IEEE, 1993.
112. I. Page. Compiling Occam into Field-Programmable Gate Arrays. In Moore and Luk, editors, *FPGAs*, pages 271–283. Abingdon EE&CS Books, 1991.

113. I. Page. Parameterised processor generation. In Moore and Luk, editors, *More FPGAs*, pages 225–237. Abingdon EE&CS Books, 1993.
114. I. Page. Reconfigurable processor architectures. *Microprocessors and Microsystems*, 20(3):185–196, May 1996.
115. Samir Palnitkar. *Verilog HDL: A guide to digital design and synthesis.* Prentice Hall, 1996. ISBN 0-13-451675-3.
116. B. M. Pangrle. Splicer: A heuristic approach to connectivity binding. In *Proccedings of the 25th Design Automation Conference*, pages 536–541. ACM Press, June 1988.
117. N. Park and A. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design*, pages 356–370, March 1988.
118. Helmuth Partsch, Wolfram Schulte, and Ton Vullinghs. System support for the interactive transformation of functional programs. In *Proceedings of the 5th Workshop on Tools for System Design and Verification (FM-TOOLS)*, 2002. Available from `http://citeseer.nj.nec.com/336476.html`.
119. P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE transactions on Computer-Aided Design*, 6:661–679, July 1989.
120. Lawrence Paulson. *ML for the working programmer.* Cambridge University Press, 1996.
121. Ad Peeters. Re: Tangram and balsa. Personal communication by email. (Peeters is a Senior Scientist at Philips Research Eindhoven).
122. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
123. S. Pommier. *An Introduction to APL.* Cambridge University Press, 1983.
124. R. Chapman and Deok-Hyun Hwang. A process-algebraic semantics for VHDL. In W. Ecker, editor, *SIG-VHDL Spring '96 Working Conference*, pages 157–168. Shaker Verlag, Dresden, Germany, 1996.
125. Jonathan Rees, W. Clinger, et al. The revised report 3 on the algorithmic language SCHEME. *SIGPLAN Notices*, 21(12):37–79, 1986.
126. S. Renault, A. Pettorossi, and M. Proietti. Design, implementation, and use of the MAP transformation system. Technical Report R. 491, IASI-CNR, Roma, Italy, December 1998.
127. Richard L. Rudell. Tutorial: Design of a logic synthesis system. In *Proceedings of the Design Automation Conference*, pages 191–196. ACM Press, 1996.
128. V. Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 5th International Workshop in Languages and Compilers for Parallel Computing*, volume 757 of *LNCS*, pages 16–30. Springer-Verlag, 1992.
129. V. Sarkar and B. Simons. Parallel program graphs and their classification. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *LNCS*, pages 633–655. Springer-Verlag, 1993.
130. Bruce Schneier. *Applied cryptography: protocols, algorithms, and sourcecode in C.* John Wiley and Sons, New York, 1994. ISBN 0-471-59756-2.
131. R.W. Sharp and A. Mycroft. A higher-level language for hardware synthesis. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*. Springer-Verlag, 2001.
132. M. Sheeran. muFP, a language for VLSI design. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, 1984.

133. D. Springer and D. Thomas. Exploiting the special structure of conflict and compatibility graphs in high-level synthesis. In *Proceedings of the International Conference on Computer Aided Design*, pages 254–257, 1990.
134. Tony Ma Stephen A. Edwards and Robert Damiano. Using a hardware model checker to verify software. In *Proceedings of the 4th International Conference on ASIC (ASICON)*. IEEE Press, 2001.
135. Donald E. Thomas, E. M. Dirkes, Robert A. Walker, Jayanth V. Rajan, J. A. Nestor, and Robert L. Blackburn. The system architect's workbench. In *Proceedings of the 25th Design Automation Conference*, pages 337–343. ACM Press, 1988.
136. C. Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design*, 5(3), July 1986.
137. C. Van Berkel. *Handshake circuits: An Intermediary Between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.
138. K. Van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.
139. Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80C51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.
140. J. van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant*. PhD thesis, University of Cambridge, 1992.
141. P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.
142. R. Walker and D. Thomas. Behavioral transformations for algorithmic level IC design. *IEEE Transactons on Computer-Aided Design*, 8(10):1115–1127, 1989.
143. Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI design: A systems perspective (second edition)*. Addison-Wesley, 1993.
144. Philip A. Wilsey. Developing a formal semantic definition of VHDL. In Jean Mermet, editor, *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, pages 245–256. Kluwer Academic Publishers, 1992.
145. Z. Zhu and S.D. Johnson. An example of interactive hardware transformation. Technical Report TR383, Computer Science Department, Indiana University, 1993.
146. G. Zimmerman. Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA. *Informatik-Fachberichte*, 5, 1976.

Index

ADA, 20
ALERT, 8
Allocation, 9, 12
Arbitration circuitry, 51, 78, 85, 102
Architecture-neutral analysis, 64, 87, 111
Architecture-specific analysis, 65, 87, 99
Ariadne, 23
Array construct (SAFL+), 120
ASAP scheduling, 11
Asynchronous circuit, 15, 32

Balsa, 32
Behavioural language, 1, 2
Behavioural synthesis, 1
Behavioural-level transformation, *see* Source-level transformation
Binding, 9, 12
Black-box synthesis, 15, 44, 141
BUD, 12, 13

Callee-save, 97
Caller-save, 97
Cathedral-II, 13
Ceres, 23
Channel circuit, 118
Channel passing, 113, 116
Chemical abstract machine, 123
Compatability graph, 12
Congruence, 123
Context, 123
Control edge, 67, 88
Control flow graph, 72
CSP, 31, 33
Cycle counting, 99

Daisy, 28
Data dependence graph, 72
Data edge, 67, 88
Data producer, 68, 88
DDD system, 30, 50
DES, 155, 160
Discrete-event model, 20
Dual flip-flop synchroniser, 83
Dual-ported RAM, 164, 167

Elf, 12
Esterel, 33
Evaluation state, 123
Expl, 11
Explicit module definition, 137
External Call Control Unit (ECCU), 76, 104
External channel, 116, 165

Facet, 13
FLaSH compiler, 65, 155
Fold/unfold transformation, 44, 149
FPGA, 155
Functional Abstract Machine, 144
Functional programming language, 26, 129
Functional unit, 75
`functor`, 131

Gated single assignment, 72
General recursion, 151
Globalization, 50
Globally Asynchronous Locally Synchronous (GALS), 81, 110, 167

Handel, 31
Handel-C, 31
Hardware description language, 1
Hardware/software co-design, 142
HardwareC, 23, 52
Haskell, 28, 131
Hawk, 30
HDRE, 28
Hebe, 13, 23
Hercules, 23, 25
Heterogeneous multiprocessor architecture, 143
High-level synthesis, 1
Higher-order function, 26

Implicit module definition, 138
Integer Linear Programming (ILP), 13
Intermediate code, 87
Intermediate graph, 67, 88
Interval analysis, 100

Lava, 28, 129
Lazy evaluation, 29
Leonardo, 155
Library block, 134
List Scheduling, 12
Logic synthesis, 8
Lustre, 33

Magma, 130
Mercury, 23
Metastability, 81
MIMOLA, 9, 13
ML, 38, 113, 131
ModelSim, 160
Monad, 130
muFP, 26, 129

Netlist, 6
Non-determinism, 126

Occam, 31, 33
Olympus Synthesis System, 23, 53
Operational semantics, 121

Parallel conflict analysis, 56, 87
Parallel program graph, 72
Parameterised processor, 143
Partial evaluation, 153
Partitioning function, 143
Perl, 138
Permanising register, 89, 90
Phase order problem, 13
Physical layout, 8
π-calculus, 113
Pipelining transformation, 142, 151
Pixel clock, 167
Place and route, 8
Polymorphic type, 26
Process calculus, 113
Processor instance, 144
Processor template, 144
Program dependence graph, 72
Program state, 122

Quartus-II, 158

Reactive system, 33
Register placement analysis, 88
Relative scheduling, 53
RTL Language, 3
RTL synthesis, 1
Ruby, 27

S-box, 160
SAFL
  circuit area, 37, 107
  interfacing with external functions, 80, 165
  resource awareness, 43
  scheduling, 51
  side-effects, 38
  software compilation, 146
  static analysis of, 56, 87
  type system, 39
SAFL+, 113, 151
Scheduling, 9, 11, 44, 51
Scheme, 30
SECD machine, 144
Sequencing graph, 52
Sharing conflict, 89
Signal, 33
Signal generator, 164
`signature`, 131
Sized types, 50
SML/NJ, 155
Soft scheduling, 51
Soft typing, 51
Source-level transformation, 16, 44, 62, 141
Splicer, 13

Stack machine, 143, 144
Statecharts, 33
Static allocation, 37
Structural block, 14, 20, 47
`structure`, 131
Synchronisation failure, 81
Synchronous channel, 113, 115
Synchronous language, 33
Synchronous timing analysis, 88
Synthesis constraint, 3
System Architect's Workbench, 13, 141
System-on-a-Chip, 14

Tangram, 31, 52
Term-rewriting system, 30
TRAC, 30
Transformation function, 143
Transistor density, 1
Transition relation, 122

Verilog, 14, 19, 79, 129, 137
VGA interface, 162
VHDL, 19, 129, 137
VLIW architecture, 150

Yorktown Silicon Compiler, 11, 13