HCircal: A Hardware Compiler for Circal

Oliver Diessel and George Milne
Advanced Computing Research Centre
School of Computer and Information Science
University of South Australia
Adelaide SA 5095

March 2, 2000

Abstract

Reconfigurable computers based on field programmable gate array technology allow applications to be realized directly in digital logic. The inherent concurrency of hardware distinguishes such computers from microprocessor–based machines in which the concurrency of the underlying hardware is fixed and abstracted from the programmer by the software model. However, reconfigurable logic allows the potential to exploit "real" concurrency. We are therefore interested in knowing how to exploit this concurrency, how to model concurrent computations, and which languages allow us to control the hardware most effectively.

The purpose of this report is to describe a FPGA compiler for the Circal process algebra. In so doing, we demonstrate that behavioural descriptions expressed in a process algebraic language can be readily and intuitively compiled to reconfigurable logic and that this contributes to the goal of discovering appropriate high–level languages for run–time reconfiguration.

The introductory sections of this report motivate our work and describe the Circal system modelling framework before describing a technology-independent circuit representation for system behaviours specified in Circal. The latter sections of the report describe a mapping of these circuits to the XC6200 series FPGA chips of a SPACE.2 reconfigurable computer. We describe the input language, circuit realization, and driver programs in more detail in appendices. Finally, we describe the organization and maintenance of the computer-based files constituting the system.

1 Introduction

The term *reconfigurable computer* is currently used to denote a machine based on field programmable gate array (FPGA) technology. This chip technology is programmable at the gate level thereby allowing any discrete digital logic system to be instantiated. It differs from the classical von Neumann computing paradigm in that a program does not reside in memory but rather an application is realized directly in digital logic. Since this logic is repeatedly programmable, the underlying FPGA platform may repeatedly be instantiated to create completely different custom computer realizations for each distinct application. Certain FPGA technologies are also *dynamically reconfigurable* in that part of the FPGA logic may be reconfigured while another part is running and, even more radically, this technology permits part of the programmable logic to be used to reconfigure another part in real time.

For some computing and electronic control applications we are able to exploit the inherent concurrency of digital logic to directly realize algorithms as custom hardware to gain a performance advantage over software executing on conventional microprocessors. The advent of field-programmable logic, in the form of FPGAs allows us to achieve these goals. It has been shown that FPGAs can host particular classes of high performance applications very successfully, and at relatively low cost [2, 5].

It is the inherent presence of concurrent activity that distinguishes hardware from software. Software operates at a conceptual level of abstraction such that the concurrency in the underlying microprocessor hardware is unknown to the software programmer and is unseen when the code runs.

But reconfigurable computing allows the potential for exploiting "real" concurrency. Given this observation, we may ask a wide range of questions, such as: How do we exploit this concurrency? How do we harness it to perform computations in perhaps a quite different manner to sequential von Neumann computation? What form should this computation take? For example should we adopt a style of evolutionary or neural computation and map it directly into FPGA circuits?. How do we model such computation? And what programming languages should we use to help programmers/designers?

This report demonstrates that we can intuitively and rapidly compile a high–level language that is oriented to describing concurrency and communication into reconfigurable logic. We show how the core features of process algebras [6, 9, 4] and the Circal process algebra in particular [6, 7] can be mapped into reconfigurable logic. A brief description of the Circal process algebra is given in Section 2.

The rationale for focusing on using a process algebra as the basis of a language for specifying reconfigurable logic are that it expresses the behaviour of a design in an abstract, technology–independent fashion and it emphasizes computation in terms of a hierarchical, modular, and interconnected structure, which supports the concepts of partial and dynamic reconfiguration where we may swap in and replace individual processes rather than performing complete reconfigurations. It should be noted that process algebra have an extensive track record in the expression and representation of highly concurrent systems including digital hardware [1, 7] and are thus a good basis for a high–level language.

A high–level language based on a process algebra is quite different from classical hardware description languages, such as VHDL and Verilog, that are oriented towards register–transfer and gate–level descriptions. Instead, this approach provides designers with a design paradigm focused on behavioural process modules and their interconnection. Because of its modular focus, our approach aids the rapid compilation and partial reconfiguration of designs at run–time. Our approach also presents us with the potential for formally verifying the compilation algorithm, thereby allowing us to state that all designs expressed in the source language are correctly implemented in the underlying reconfigurable computing engine. Related research on verifiable compilation to FPGAs was performed by Shaw and Milne [11] while Page and Luk [10] also constructed an Occam to FPGA compiler.

Circal models emphasize the control of and communication between processes. The rapid compilation of Circal models allows assemblies of interacting finite state machines, upon which logic control paths are based, to be implemented quickly. Apart from logic controllers, we may thus be able to build and quickly modify test pattern generators that function at near hardware speed. This project also aims to support dynamic structures that may facilitate the control of dynamically reconfigurable logic.

In the following section we provide an overview of the Circal process algebra and the source language for our compiler. Section 3 introduces our contribution with an overview of the compiler. We describe a technology–independent circuit model of Circal processes in Section 4. The mapping of these circuits to FPGAs, and Xilinx XC6200 chips in particular, is discussed in Section 5. The derivation of the mapping from behavioural Circal descriptions is outlined in Section 6. A summary of the results and directions for further study are presented in Section 7.

2 The Circal process algebra

This section presents Circal as a descriptive medium for reconfigurable computing. We describe the key language concepts and how they are used to describe concurrent systems.

Circal is a formal language for modelling the behaviour of systems by modelling the behaviour of its component processes and how those components interact based on the occurrence of events. Systems, and processes in turn, are described hierarchically and in a modular fashion. The description of a system thus typically proceeds in a top-down manner with the elaboration of component processes leading to further decomposition until the desired level of description is reached. In principle it is possible to elaborate designs to the level of logic gates and signals (see, for example, [7]), however, it is desirable to remain at a slightly more abstract level so as not to lose descriptive power.

Circal is an event–based language; processes interact by participating in events, and sets of simultaneous events are termed actions. For an event to occur, all processes that include the event in their specification must be in a state that allows them to participate in the event. The Circal language primitives are:

State Definition $P \leftarrow Q$ defines process P to have the behaviour of term Q. Process Q is given the name P.

Termination Δ is a deadlock state from which a process cannot evolve.

Guarding $a\,P$ is a process that synchronizes to perform event a and then behaves as, or evolves to, P. $(a\,b)\,P$ synchronizes with events a and b simultaneously and then behaves as P.

Choice $P + Q$ is a term that chooses between the actions in process P and those in Q, the choice depending upon the environment in which the process is executed. Usually the choice is mediated through the offering by the environment of a guarding event.

Non–determinism $P \,\&\, Q$ defines an internal choice that is determined by the process without influence from its environment. Either branch might be taken by the process, the reason for the choice being unobservable.

Composition $P * Q$ runs P and Q in parallel, with synchronization occurring over similarly named events. When P and Q share a common event, both must be in a state in which they can accept that event before the event and synchronous state evolution can occur. P and Q may independently respond to events that are unique to their specification. Should such events occur simultaneously, the processes respond independently and simultaneously.

Abstraction $P - a$ hides event set a from P, the actions in a becoming encapsulated and unobservable. Unobservable events internal to a process lead to non-deterministic behaviour.

Relabelling $P[a/b]$ replaces references to event b in P with the event named a. This feature is similar to calling procedures with parameter substitution.

Processes and the events they respond to are central to PAs such as Circal. In Circal, the set of events a process can respond to is known as its sort. Process sorts have an important bearing on the behaviour of systems that are composed of more than one process. A process that is part of a composition does not care about events that aren't in its sort. However, if the environment of a system offers an event that is in the sort of one or more processes in a composition, those processes must be in a state that can respond to that event before the event can occur. The system thus constrains and responds to the occurrence of events in its environment.

State evolution proceeds whenever events in the sort of a process happen to be guarding a possible next state for the current state. When processes are composed, it is useful to visualize similarly labelled event ports on constituent processes as being wired together. For an event to occur, the composition law requires that all components that can participate in the event (those that include that event in its sort) be in a state from which they may individually evolve. This mechanism allows the system modeller to impose constraints on the behaviour of the system, mediated by the need for all components to accept an event before any component may evolve. The constraint may be relaxed by abstracting the event (hiding the event and encapsulating the response) or by naming the event differently for the participating processes.

Circal is unique among PAs in that it has a strict interpretation of the response of processes to the simultaneous occurrence of events, and is therefore well-suited to modelling synchronous devices such as FPGAs. Some of the difficulty experienced in previous Occam to FPGA compilers arose from the need to express simultaneous hardware behaviour using the essentially interleaving formalism of Occam.

Circal has been used to describe systems composed of interacting finite state machines, to describe the control paths for digital systems, and to specify cellular automata for FPGA implementation. To demonstrate the modelling concepts, consider a mobile phone system that is also capable of receiving and recording television or radio broadcast signals. We shall examine the description and composition of parts of the broadcast receiver subsystem, B, and the phone subsystem, P. Consider a mode of operation in which the user initiates broadcast reception by the action, denoted s, of selecting a channel. Should an urgent phone call arrive, the system is able to buffer reception of the broadcast while the user answers the phone with action a. When the user hangs the phone up, the broadcast may be resumed by event r from the time it was interrupted and the remainder of the reception is buffered until the user flushes the buffer by selecting a new channel.

These subsystems and their composition may be described using Circal and/or state transition diagrams. In this presentation, we depict a functional block diagram of the composed subsystems of the mobile phone M in Figure 1.

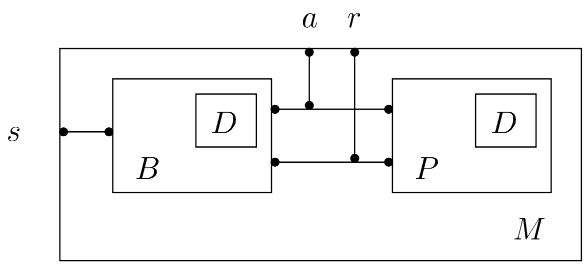

Figure 1: Mobile phone system comprising broadcast and phone subsystems.

The D box enclosed by the B and P boxes of Figure 1 represents a decompression function that is not further decomposed here, but could be shared between both subsystems. Initially, the broadcast signal is decompressed before presentation. When the phone is answered, the incoming audio signal is decompressed. After the phone call is complete, the buffered broadcast signal is decompressed while the incoming compressed broadcast signal is buffered. The description of the dynamic links between the D block and communicating sub-components of the B and P blocks is the subject of an ongoing investigation into the language of dynamically structured Circal [8].

This report is concerned with describing the compilation into static circuits of behavioural process specifications given in Circal. This work is intended to lay the foundation for an interpreter that will use the concepts outlined here to translate bahaviours into circuits, but will also allow those circuits to change during the course of the execution to reflect dynamic behaviours and manage a hardware resource of limited size.

3 Overview of the compiler

The Circal system implements the XCircal language, which extends the descriptive facilities available to the Circal user to readily build large or complex models [7].

XCircal has many features derived from the programming language C. The principle features that ease the construction of process models are: predefined data–types for Circal action sets, `Event`, and Circal processes, `Process`, arrays of which may also be used; parameterised functions that build and then return processes; and control structures, such as `for` loops and `if` statements, that can be used in the construction of processes.

This paper describes our efforts to implement a subset of Circal suited to the instantiation of Circal process models as reconfigurable logic circuits. The implementation of the hardware compiler is referred to as HCircal to distinguish it from the XCircal system. One of our long–term goals is for HCircal to accept XCircal source files. At present, HCircal accepts a subset of XCircal that does not include parameterised functions or control structures.

An HCircal source file consists of a declaration part, a process definition part, and an implementation part. Events and processes must be declared before use. The definition part consists of a sequence of process definitions adhering to the Circal BNF. The implementation part is introduced with the `Implement` declarative and is followed by a comma–delimited list of process compositions that is to be implemented in hardware. Processes must be defined before they are referred to in an `Implement` statement. HCircal does not currently allow the user to model non–determinism, abstraction, or relabelling. However, implementations of abstraction and relabelling are straightforward extensions to the current system. A complete description of the Circal BNF may be found in Appendix A.

In outline, the HCircal compiler operates as follows:

1. The user inputs an HCircal specification of the system to be implemented.
2. A compiler analyzes the specification to produce a hardware implementation and a driver program for interacting with the hardware model
    - The current hardware model is in the form of a Xilinx XC6200 FPGA configuration bitstream [12] suitable for loading onto XC6200–based reconfigurable coprocessors such as the SPACE.2 board [3].
    - The driver program is a C program that executes on the host. The program loads the configuration onto the coprocessor and allows the user to interact with the implemented system.
3. The user runs the driver program and interacts with the hardware model by entering event traces and observing the system response.

The following sections describe the mapping from behavioural descriptions to technology–independent circuits, the decomposition of the circuits into modules for which FPGA configurations are readily generated, and the derivation of the module parameters from the Circal specification. The generation of the host program is a straightforward specialization of a general program that obtains appropriate event inputs, loads the input registers, and reads the process state registers. The generation of the host program is described in further detail in Appendix C.

4 A circuit model of Circal

The aim of the model is to represent, as faithfully as possible, Circal semantics in hardware. The design concentrates on the representation of the Circal composition operator, which is of central importance because it is through the composition of processes that interesting behaviour is established. When processes are composed in hardware they are executed concurrently.

The hardware implementation of the Circal system follows design principles that aim to generate fast circuits quickly. The first of these is that, for the sake of speed and scalability, the hardware representation of Circal aims to minimize its dependence upon global computation at the compilation and execution phases. The second principle is that we choose to design for ease of run–time instantiation and computational speed over area minimality. The motivation for these choices is the desire to leverage the speedup afforded by concurrently executing the Circal system in hardware; they are supported by the ability to reconfigure the gate array at run–time in order to provide a limitless circuit area. Finally, we desire a reusable design because we believe that will facilitate design synthesis, circuit reconfiguration, and future investigations into dynamically structured Circal.

4.1 Design outline

A block diagram of a digital circuit that implements a composition of Circal processes in hardware. is shown in Figure 2(a).

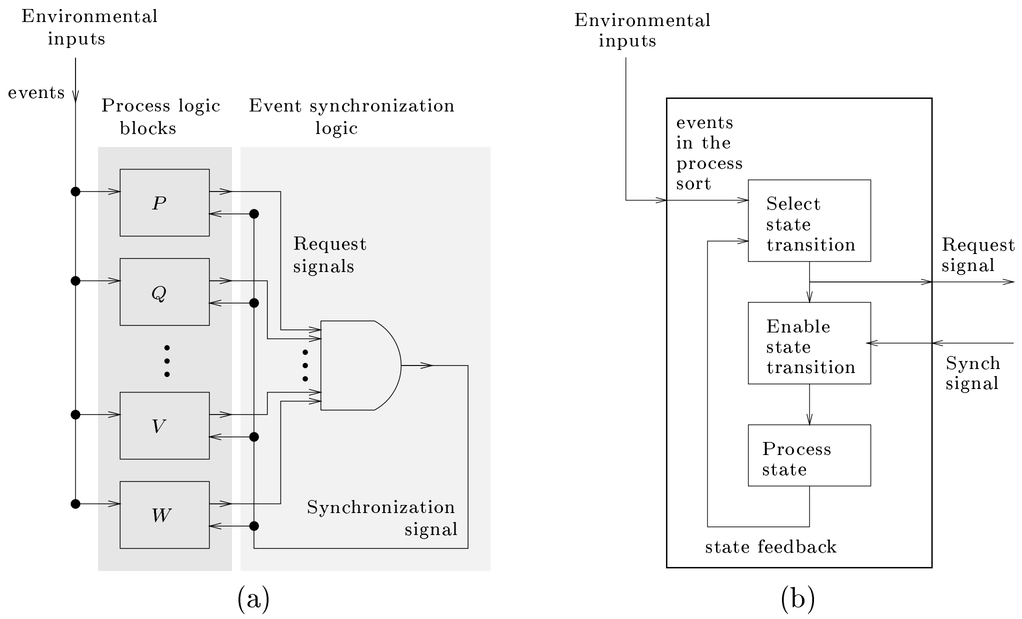

Figure 2: (a) Circuit block diagram, and (b) Circal process logic block.

The circuit consists of a set of interconnected processes that respond to inputs from the environment by undergoing state transitions.

Processes are implemented as blocks of logic with state. In a given state, each process responds to events according to the Circal process definitions. Individual processes examine the event offered by the environment and produce a "request to synchronize" signal if the event is found to be acceptable. The request signals for all processes are then reduced to a single synchronization signal that each process responds to independently.

Implementing Circal in synchronous FPGA circuits leads us to assume that: each process is in a known state during each clock period; state transitions occur on clock edges; an event occurs at most once during a clock period; the next state is determined by the events that occurred during the previous clock period; events are sampled on positive clock edges; and, if no event occurs between consecutive positive clock edges, then the idling transition $P \rightarrow P$ occurs upon the second clock edge by default. Since process evolutions are dependent upon timed samples of events, the presence of clock ticks is implicit in all guards.

4.2 Process logic design

Process logic blocks are derived from the process definition syntax and represented as compact localized blocks of logic to simplify the placement and routing of the system. A high–level view of a process logic block is given in Figure 2(b).

A process is designed to respond to events in the environment that are acceptable to all processes in the composed system. In order to perform this function, the process logic first checks whether the event is acceptable to itself. If all processes find the event acceptable, the event synchronization logic returns a synchronization signal that is used by individual process logic blocks to enable the state transition guarded by the event.

The current state of the process is stored in a process state register. Each state is represented by a flip–flop that becomes active when the process enters that state and remains active while the process remains in that state.

In a given state, each process is prepared to synchronize on some subset of the events in its sort. However, in a composed system, for any event to occur, all processes must agree on the acceptability of the event.

Three steps are needed to agree upon the events that are permitted. Taken together these steps comprise the state renewal cycle:

1. In the first step, each process independently and in parallel determines whether the combination of events offered by the environment is a valid combination for its current state. This check is performed by determining whether the combination of events that intersects its sort forms a valid guard for the current state. If so, the process flags its willingness to accept the event by raising a request signal. Otherwise the signal is not raised.

2. The second step checks whether the event combination is acceptable to all processes in the composed system by forming the global AND of the request signals generated during the first step. The result of this calculation is broadcast to all processes in order for them to synchronize their transitions to a new state.

3. In the third step, each process determines its next state based on the combination of events offered by the environment and whether or not all processes accepted the offer. The default is to remain in the current state if no events occurred or the combination of events was not accepted by all processes during the second step.

The following subsections describe the process logic design in more detail.

4.2.1 Determining the validity of event combinations

We construct a combinational circuit that checks whether the events in the sort of the process form a valid guard for the current state. The process also accepts a null event (an event not in its sort) in order to allow other processes to respond to events it does not care about. The current state of the process is recycled if an unacceptable or null event is offered by the environment.

Let us assume at most k possibly recursive definitions $P_0, P_1, \dots, P_{k-1}$ are necessary to describe the evolution of process P with sort $S = \{e_0, e_1, ..., e_{n-1}\}$, and that P_i, with $0 \leq i \leq k-1$, is defined as $P_i \leftarrow g_{i,0}\, P_{i,0} + ... + g_{i,j}\, P_{i,j} + ... + g_{i,j_i}\, P_{i,j_i}$, where index i refers to the current state, $P_{i,j}$ is the next state, $P_0, ..., P_{k-1}$, the state P_i evolves to under guard $g_{i,j} \subseteq S$, and $g_{i,j}$ is interpreted as the simultaneous occurrence of the events in $g_{i,j}$. The definition for P_i consists of $j_i + 1$ guarded terms where the $g_{i,j}$ are all distinct. Note that there may be at most k distinct next states but $2^n - 1$ distinct guards.

If we think of the events and states as *boolean* variables, then in state P_i the process responds to event combinations in the set $\{\gamma_{i,j}\} \cup \{\nu_S\}$, where $\gamma_{i,j} = \varepsilon_0\, \varepsilon_1 \dots \varepsilon_{n-1}$ and $\varepsilon_l = e_l$ or $\varepsilon_l = \overline{e_l}$, for $0 \leq l \leq n-1$, depending upon whether or not $e_l \in g_{i,j}$, and where $\nu_S = \overline{e_0}\, \overline{e_1} \dots \overline{e_{n-1}}$ is the null event for sort S. Process P in state P_i therefore accepts the boolean expression of events $\nu_S + \sum_{0 \leq j \leq j_i} \gamma_{i,j}$.

The request for synchronization signal, r_P, is thus formed from the expressions for all states: $r_P = \sum_{0 \leq i \leq k-1} (\nu_S + \sum_{0 \leq j \leq j_i} \gamma_{i,j}).P_i$.

4.2.2 Checking the acceptability of an event

The request signals for all processes are ANDed together in an AND gate tree that is implemented external to the individual process logic blocks. The output of the tree is fed back to each process as the synchronization signal, s.

4.2.3 Enabling state transitions

The state of the process is stored in flip–flops — one for each state. Let $D_{P_l}, 0 \leq l \leq k-1$, denote the boolean input function of the D–type flip–flop for state P_l. Then we can derive the following boolean equations from the process definitions: $D_{P_l} = s.(\nu_S.P_l + \sum_{0 \leq i \leq k-1} \sum_{P_{i,j} = P_l} \gamma_{i,j}.P_{i,j}) + \overline{s}.P_l$, for $0 \leq l \leq k-1$.

In the above equations, the terms in parentheses are enabled when the synchronization signal, s, is asserted. These terms correspond to the guards on state transitions and to state recycling if a null event was offered to this process. The last term in the equations forces the current state to be renewed if the processes could not accept the event combination offered by the environment. By observing the synchronization signal, the environment can determine whether or not an event was accepted and can thus be constrained by the process composition.

To initialize the process state in a known state, we designate one of the states as the initial state. Since D–type flip–flops may be cleared but not set, the input function for the initial state is complemented and the complement of the output is used as the value for that state.

4.3 The complete process logic block

The disjunction of the parenthesized terms in the flip–flop input functions implements the same boolean function as that to obtain the request signal. This is necessarily so since in the latter we need to form all guards for each state, and in the former, we need to combine each guard with the state it occurs in to obtain the correct next state. We therefore use the state selection circuits to form the request signal and use the synchronization signal to enable the selection.

4.4 Design example

Consider the process P with sort $\{a, b, c\}$ defined by the equations

$$\begin{array}{lcl} P_0 & \leftarrow & a\,P_1 + (a\,b)\,P_1 + c\,P_1 \\ P_1 & \leftarrow & a\,P_0 \end{array}$$

The request for synchronisation signal, r_P, is

$$r_P = (\overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c + a\overline{b}\overline{c} + ab\overline{c}).P_0 + (a\overline{b}\overline{c} + \overline{a}\overline{b}\overline{c}).P_1$$

The boolean input functions for the P_0 and P_1 state flip–flops are

$$\begin{array}{lcl} D_{P_0} & = & s.(a\overline{b}\overline{c}P_1 + \overline{a}\overline{b}\overline{c}P_0) + \overline{s}P_0 \\ D_{P_1} & = & s.([a\overline{b}\overline{c} + ab\overline{c} + \overline{a}\overline{b}c]P_0 + \overline{a}\overline{b}\overline{c}P_1) + \overline{s}P_1 \end{array}$$

If we choose state P to be the initial state, then

$$D_{P_0} = \overline{s.(a\overline{b}\overline{c}P_1 + \overline{a}\overline{b}\overline{c}P_0) + \overline{s}P_0}$$

and

$$P_0 = \overline{Q_{P_0}}.$$

The resulting circuit is shown in the diagram of Figure 3.

5 Mapping the circuits to reconfigurable logic

In this section we consider the placement and routing of the circuits derived in Section 4. The derivation of circuit requirements from the specification is discussed in the next section.

Our primary compilation goal is to generate FPGA configurations rapidly. We also want to be able to replace circuitry at run–time to explore changing process behaviours and to overcome resource limitations. For this reason we're interested in mapping to Xilinx XC6200 technology because its open architecture allows us to produce our own tools and because the chip is partially reconfigurable.

Difficulties with placing and routing the Circal models satisfactorily with XACTStep, the Xilinx automatic place and route tool for XC6200, led us to consider decomposing the circuits into modules that can be placed and routed under program control. These modules serve as an attractive intermediate form since they are easily derived from the specification, they completely describe the circuits to be implemented in a hardware–independent manner, and the FPGA configuration can be generated from them without further analysis.

The circuits described in Section 4 are specified in terms of parameterised modules that communicate via adjoining ports when they are abutted on the array surface. To simplify the layout of the circuits, all modules are rectangular in shape. The internal layout and dynamic reconfiguration of modules is also simplified by only using local interconnects. The module representation of the

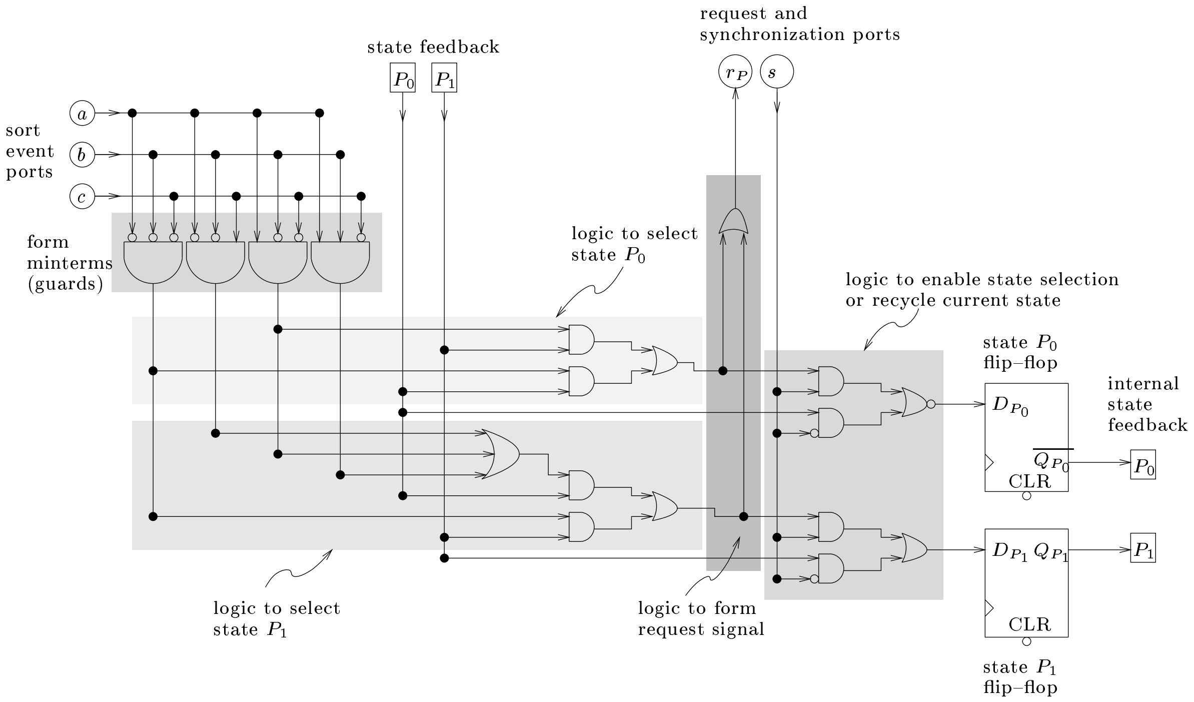

Figure 3: Process logic for process P.

circuits is readily mapped to a particular hardware technology by suitable module generators. The compiler can thus be ported to a new FPGA type by implementing a new set of module generators.

We distinguish between 10 module types. Each module type implements a specific combinational logic function using a particular spatial arrangement. Modules are specified in terms of their location on the array, input and/or output signal vectors, and the specific function they are to implement, e.g., minterm number.

The logic and routing for each module can be generated in time proportional to the area of the module. We have attempted to optimize the area of each module with the constraint that local routing be used. An additional constraint we impose is that the sense of all outputs of a module be true. This simplifies the composition and dynamic replacement of modules.

For a complete description of the function, layout, and specification of the modules, see Appendix B.

6 Deriving modules from process descriptions

For each unique process that is to be implemented, a process template that consists of the modules comprising the process logic is constructed. The module parameters for a process template are independently calculated using relative offsets. Once the size of the logic for each template is known, a copy with absolute offsets (final placement of modules) is made for each process to be implemented. When all the parameters are known, the FPGA configuration is generated.

The steps in the derivation of the module representation of the required circuit or its equivalent FPGA configuration are as follows:

1. Identify the processes to be implemented by scanning the `Implement` statement in the specification, which lists the process compositions to be implemented.

2. For each process to be implemented:
   (a) Determine its sort by determining the events guarding state transitions that are found by spanning the state tree from the initial state given in the `Implement` statement.
   (b) Compute the minterms needed by expressing the guards on state transitions in normalized form.
   (c) Form the disjunction of minterms that lead to the same next state for each current state. These groupings are formed in the so-called Guard blocks referred to in Appendix B.
   (d) Compute the state logic, comprising:
      i. state registers — one flip-flop is allocated per state;
      ii. request logic, that forms the conjunction of the output of logic described in (2c) above with the corresponding current state; and
      iii. state transition logic that enables transition to the appropriate next state if the synchronization signal is asserted.
3. For each process composition:
   (a) Determine the absolute offsets for each process by stacking their logic blocks below one another and leaving room for buses carrying input signals on the left.
   (b) Compute the synchronization logic needed from the absolute process logic offsets.
4. Compute input registers and input broadcast buses from the process offsets and process sorts.

For more detail on the specification, function, and arrangement of the circuit components, please refer to Appendix B.

The compilation approach places and routes the design entirely under program control in time proportional to the area of the circuit. A worst case assessment of the area required by the implementation assumes there are n environmental inputs, p processes, that the sort size of each process is $O(n)$, that the maximum number of minterms for any process is $k \leq 2^n$, and that no process has more than k states. In that case, the width of the design is $O(n + k \log n + \log p) = O(2^n \log n + \log p)$, and the height of the design is $O(p \times (n + k^2)) = O(2^{2n} \times p)$.

Currently the compilation is performed off–line and the configurations generated are static. In future implementations we plan to experiment with replacing modules at run–time to overcome resource limitations and implement dynamically changing process behaviours. Minor behavioural changes may simply involve replacing minterms or guard modules which could be done very quickly. The regular shapes and small sizes of modules may allow us to distribute them and finalize the module positioning at run–time in order to maximize array utilization.

7 Conclusions

We have shown how to model Circal processes as circuits that can be mapped to blocks of logic on a reconfigurable chip. Modelling system components as independent blocks of logic allows them to be generated independently, to be implemented in a distributed fashion, to operate concurrently, and to be swapped to overcome resource limitations. The model thus exploits the hierarchy and modularity inherent in behavioural descriptions to support virtualization of hardware.

We have shown how to instantiate a circuit by decomposing it into parametric modules that perform functions above the gate level. To simplify the layout, modules are mapped to rectangular regions that are wired together by abutting them on a chip. Since the modules completely describe the circuits to be implemented in a hardware–independent yet readily mapped manner, they could serve as a mobile description of Circal processes that can be transmitted and instantiated remotely.

The circuit model is readily derived from Circal behavioural specifications. A natural mapping from the circuit model allows modules to be generated quickly — mapping to different technologies should be straightforward. The time to calculate, emit and configure module logic is proportional to its area. Thus the time to instantiate a circuit is proportional to its area, which is a desirable property since it is the minimum time needed to configure the circuit onto the reconfigurable logic chip surface with current technology.

Future work will investigate developing an interpreter that adapts to resource availability and supports dynamic process behaviour. We also intend assessing the usability of process algebraic specifications for a number of applications. A further direction is to enhance the HCircal language to support stream–oriented and data–parallel computations.

Acknowledgements

We gratefully acknowledge the helpful comments and suggestions made by Alex Cowie, Martyn George, and Bernard Gunther.

References

[1] A. Bailey, G. A. McCaskill, and G. Milne. An exercise in the automatic verification of asynchronous designs. *Formal Methods in System Design*, 4(3):213–242, 1994.

[2] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, editors. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[3] B. K. Gunther. SPACE 2 as a reconfigurable stream processor. In N. Sharda and A. Tam, editors, *Proceedings of PART'97 the 4th Australasian Conference on Parallel and Real–Time Systems*, pages 286 – 297, Singapore, Sept. 1997. Springer–Verlag.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International series in computer science. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1985.

[5] E. Lemoine and D. Merceron. Run time reconfiguration of FPGA for scanning genomic databases. In P. Athanas, editor, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95)*, pages 90 – 98, Los Alamitos, CA, Apr. 1995. IEEE Computer Society Press.

[6] G. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, 1985.

[7] G. Milne. *Formal Specification and Verification of Digital Systems*. McGraw–Hill, London, UK, 1994.

[8] G. Milne. Modelling dynamic structures. Technical report ACRC–99–01, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Adelaide, Australia, Jan. 1999.

[9] R. Milner. *Communication and Concurrency*. Prentice–Hall, Inc., New York, NY, 1989.

[10] I. Page and W. Luk. Compiling Occam into FPGAs. In W. R. Moore and W. Luk, editors, *FPGAs, Edited from the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, pages 271 – 283, Abingdon, England, 1991. Abingdon EE&CS Books.

[11] P. Shaw and G. Milne. A highly parallel FPGA–based machine and its formal verification. In H. Grünbacher and R. W. Hartenstein, editors, *Field–Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping, Second International Workshop on Field–Programmable Logic and Applications*, volume 705 of *Lecture Notes in Computer Science*, pages 162–173, Berlin, Germany, Sept. 1992. Springer–Verlag.

[12] Xilinx. *XC6200 Field Programmable Gate Arrays.* Xilinx, Inc., Apr. 1997.

Appendices

A A BNF–like definition of the HCircal syntax

This appendix describes the proposed syntax for an HCircal specification. The current status of the implementation is indicated following the BNF.

A specification consists of three parts.

```
<spec>::==<decsec><procdefsec><impsec>
```

Note that the compiler ignores white space in a specification.

Events and processes must be declared before use. In particular, events must be declared before they appear in an explicit sort.

```
<decsec>::==<eventdec><decsec> | <procdec><decsec> | <procdec>
```

Note that reserved words are capitalized.

```
<eventdec>::==Event <eventlist>\n
<eventlist>::==<eventid>, <eventlist> | <eventid>
<eventid>::==<identifier>
```

Identifiers are formed from any string of letters and digits commencing with a letter.

```
<identifier>::==<letter><string> | <letter>
<string>::==<letter><string> | <digit><string> | <letter> | <digit>
<letter>::==A | B |...| Z | a | b |...| z
<digit>::==0 | 1 |...| 9
```

Sorts need not be explicit. Implicit sorts are derived from the union of the events all states in a state transition tree respond to as well as any explicitly listed events in the sort of a process declaration.

```
<procdec>::==Process <proclist>\n
<proclist>::==<procid>, <proclist> | <procid>
<procid>::==<procname> | <procname><sort> | /\<sort>
<procname>::==<identifier>
<sort>::==(<eventlist>)
```

The process definitions define the behaviour of processes.

```
<procdefsec>::==<procdef><procdefsec> | <procdef>
<procdef>::==<procname> <- <expression>\n
<expression>::==<term> | <term> + <expression>
<term>::==<procname> | <guardlist><procname> | /\ | <guardlist>/\
<guardlist>::==<guard><guardlist> | <guard>
<guard>::==<eventid> | (<eventlist>)
```

The implementation section describes the composed systems of processes that are to be implemented. In particular, a comma delimited list of systems may be implemented. Process IDs may be used in more than one composed system, and each reference results in the implementation of an additional corresponding process logic block.

```
<impsec>::==Implement <implist>\n
<implist>::==<compsys> | <compsys>, <implist>
<compsys>::==<procname> | <procname> * <compsys>
```

A.1 Current status of the implementation

The compiler currently provides a simplified interface that implements the structure, if not the syntax, described above.

The current implementation takes as input an ascii file containing 4 lists. In order, from the beginning of the file to the end, these are a list of declared events, a list of declared processes, a list of process transitions, and a list of process compositions that is to be instantiated.

Each list begins with an integer, l, representing the number of items in the list. The items are listed on the following l lines of the file, one item per line, i.e., one event name, process name, process definition, or process composition per line.

The format of an event or process name is as in the BNF for an `eventid` or `procid` respectively.

A process definition consists of the of the process being defined, a guard consisting of a single event name of a comma delimited list of event names enclosed by parentheses, and the name of the guarded process. Empty parentheses indicate an unguarded transition.

A process composition consists of an asterisk delimited list of process names. The names used to identify the processes in the composition correspond to the initial states of the components.

As an example, the HCircal specification of Figure 4 is actually entered as in Figure 5.

```
Event a, b, c, d
Process P, P1, Q, Q1, R(a,b)

P <- a P1 + (ab) P1 + c P1
P1 <- a P

Q <- a Q1 + d Q1
Q1 <- b Q

R <- a /\

Implement P*Q, R, P*R
```

Figure 4: Sample proposed HCircal specification

```
4
a
b
c
d
6
P
P1
Q
Q1
R
R1
9
P a P1
P (a,b) P1
P c P1
P1 a P
Q a Q1
Q d Q1
Q1 b Q
R a R1
R1 () R1
3
P*Q
R
P*R
```

Figure 5: Specification as actually entered

B Circal Modules

There are 10 different types of modules. In the following we briefly describe their function, layout, and specification in the order in which they are encountered as signals flow through the circuit.

B.1 Environmental Inputs

Inputs to a composed system are provided through a column of registers in the first module, known as the "Environment Inputs" (EI) block.

The EI block consists of an input register and wires to feed into process logic blocks to the right and below the EI block. It is specified by the coordinates of its top left corner, the number of inputs in the register that occupies its left–most column, and bit vectors that specify which of the inputs emerge to the right and/or to the bottom of the block. The block is square — its side length in cells being given by the number of inputs. This block is intended to occupy the top left corner of a chip.

Processes are stacked on top of each other. The process logic immediately to the right of an EI block is one of the processes in the set of composed systems that is to be implemented. The remaining processes to be implemented occupy adjoining regions of the chip below this process.

B.2 Minterms

Inputs to a process logic block, be they from an EI block or an Input Junction block (described below), are packed into a contiguous set of rows at the top-left corner of the process logic block. These feed into so called "Minterm" (MT) blocks that detect the combinations of event inputs a process can respond to.

Each MT block detects a minterm of its inputs, passes the inputs to the right, and outputs the result of the minterm detection at the bottom-left corner of the block. It is specified by its top left corner, the number of inputs n, which corresponds to its height, and the function $f, 0 \leq f \leq 2^n - 1$ it detects. The width of an MT block is a logarithmic function of the number of its inputs.

Usually a number of contiguous MT blocks are required to detect all of the guards for a process. It is intended that they be arranged in decreasing function order from left to right across a chip.

B.3 Guards

Often a process in a given state will respond to several different guards. In order to group the guards that cause a given current state to evolve to the same next state, a so–called "Guard" (G) block is used. These form the logical disjunction (OR) of selected MT block outputs arriving from above and produce an output at the top-right corner of the block. They are specified by the top left corner, the number of inputs m, their spacing (given MT blocks with differing numbers of inputs have different widths), and a function $g, 0 \leq g \leq 2^m - 1$ indicating which inputs are to be combined. The height of the blocks is logarithmic in the number of inputs. The inputs are also routed in columns to the bottom edge of the block to allow several guard blocks to be vertically stacked.

B.4 Requesters

Signals flow from the G blocks to the right through a set of "Requester" (R) blocks, one of which is associated with each possible process state.

A requester serves several functions. The first is to form the logical conjunction between its associated state and the sets of guards the process responds to in that state. An output to the right is produced for every possible next state. A second requester function is to transmit signals arriving from the left through the block to the right. These signals include guards that are to be combined with states that are laid out to the right of the requester and signals selecting the next state for requesters that are laid out to the left. A third function is to transmit signals arriving from the right through the block to the left. The bottom–most of these is routed to the state logic below to form the select signal for the state associated with the requester. The remaining signals routed to the left are the select signals for states that are laid out in columns to the left of the requester.

R blocks are specified by the coordinates of the top left corner, their height, and 3 bit vectors that identify the rows on which guard signals are arriving from the left to form request signals, the rows on which signals that are to be routed through the block are arriving from the left, and the rows on which signals that are to be routed through the block are arriving from the right. R blocks are a single column of cells wide.

B.5 State Logic

The set of states associated with a process is stored in a row of contiguous registers below and to the right of the guard blocks.

One register is associated with each state (one–hot encoding). One of the registers is associated with the initial state of the process. The logic for this state is slightly different.

Both the "Initial" and the "Non–Initial State" (IS & NIS) logic blocks occupy a single column. However, while NIS blocks are a single cell high, IS blocks are two cells high. The blocks are specified by the coordinates of the top left corner of the block and the sense of two wires that enter the block: the state selection wire that enters from the north and a synchronization wire that enters from the east. These are the only inputs in the implementation whose sense cannot be guaranteed to be true when they emerge from intervening blocks.

The function of a state register is to latch the value on the selection wire if the synchronization signal is asserted when a clock pulse occurs, otherwise the current state is renewed. Since registers are initially cleared, the output of the IS block is inverted.

B.6 OR Trees

In order to form the select signal for a process state, the request signals that lead to the same next state for each current state are combined by logical disjunction. The "OR Tree" (OT) module forms the logical disjunction of its inputs. An OT block is specified by the coordinates of its top left corner, its height, a switch indicating whether output to the left as well as to the right in its bottom-most row is required, and a vector of bits indicating which rows inputs are arriving on. The inputs are assumed to arrive from the left and output is by default produced to the right. The width of the block is logarithmic in the number of its inputs.

An OT block is also used to combine the select signals for all possible next states into a single request signal for the process block.

B.7 Buses & Input Junctions

In order to provide multiple process logic blocks with input we use two module types. "Bus" (B) blocks are used to connect the outputs from the base of the EI block to the inputs of one or more "Input Junction" (IJ) blocks below the EI block.

A bus block specifies a rectangular region by its top left corner, width and height. The block is crossed by a set of wires oriented to the north, south, east or west, according to a direction switch, and the columns or rows that carry signals are given by a bit vector of active inputs.

Modules are designed to be abutted. However, when the arrangement of modules precludes their abutment, bus modules are used to transport signals to their destination.

An input junction allows the input corresponding to the sort of a process to be tapped from a bus heading south from the base of the EI block. An IJ block is specified by the coordinates of its top left corner, its width, a bit vector indicating which inputs are to be tapped off to the right, and a bit vector indicating which inputs are to be output to the next bus segment below.

B.8 Synchronization Logic

Finally, the request signals for each process are combined by logical conjunction in a "Synchronization Logic" (SL) block. This block is specified by the coordinates of its top left corner, its height, and a bit vector indicating the rows containing process request signals arriving from the left.

The width of the block is logarithmic in the number of request signals. A synchronization signal corresponding to the AND of all inputs is produced in the row below each request signal and output to the left so that the synchronization signal can be broadcast to each state register of all process logic blocks in a system of composed processes.

B.9 Module arrangement

The module arrangement for a typical system $P * Q * \ldots$ is depicted in Figure 6.

The **E**nvironmental **I**nputs block, which provides input to process P on its right, is aligned with the top-left corner of the chip. Below it, **B**us blocks route inputs southward to **I**nput **J**unction blocks for inputing events to subsequent process logic blocks laid out below P.

Within the logic block for process P, the inputs at the top-left edge are routed to the right through **M**in**T**erm blocks, which produce functions at their southern edges. These are routed southwards through **G**uard blocks that form the disjunction of one or more minterm functions. Output of the **G**uard blocks from their top-right corners are routed to the east through **R**equester blocks, where they are combined with the appropriate current state to form components of the process request signal. Those components leading to the same next state are further reduced in the leftmost column of **O**R **T**ree blocks, and the request signal proper is formed in a second level of **O**R **T**ree on the eastern border of the process logic block. Feedback from the first level OT blocks is routed back to the west through the R blocks and down towards the state logic to serve as state selectors.

The request signals of all process logic blocks are reduced to a global synchronization signal in the **S**ynchronization **L**ogic block that abuts the process logic on the right. This signal is routed back into each process block, where a **B**us block connects it with the rightmost state logic block. From here, it is routed westwards through **N**on-**I**nitial **S**tate and **I**nitial **S**tate logic blocks to enable the state transition selected by the event. When the process logic does not abut the SL block, **B**us blocks are used to convey signals between the two.

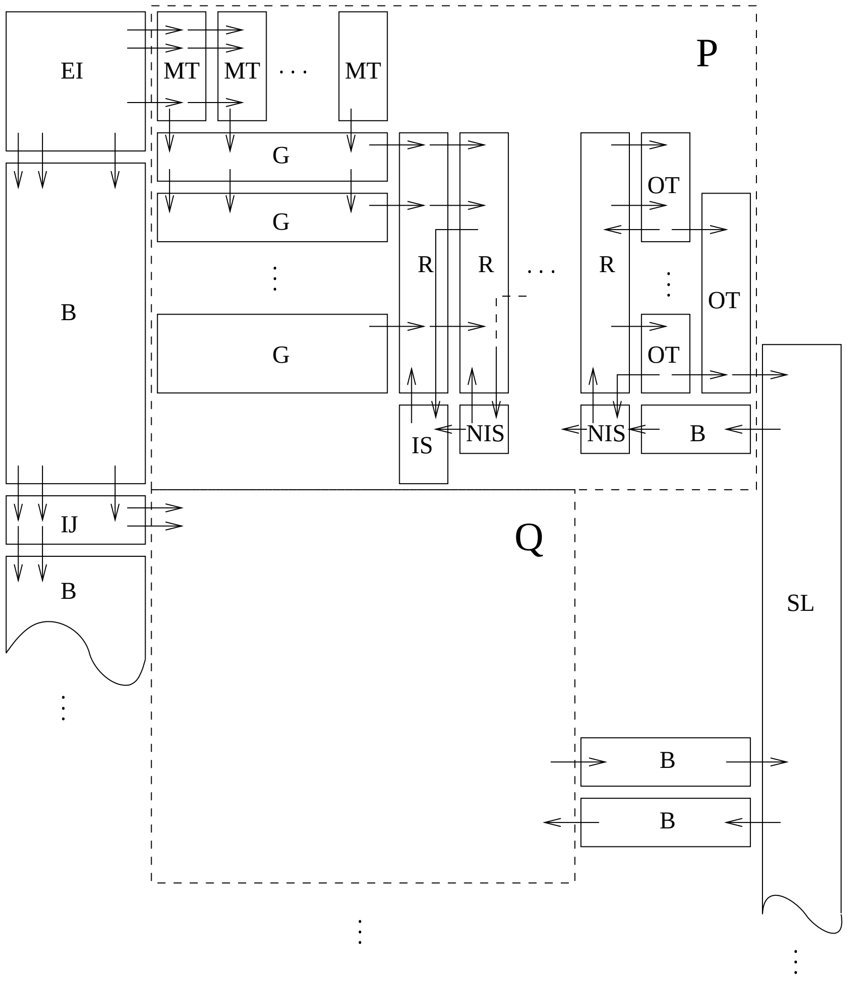

Figure 6: Typical circuit module arrangement

C Generating the host driver program

The purpose of the host driver program is to load the configuration for the hardware model and to provide an interface for the user to interact with the model. The pseudocode for the program is listed in Figure 7.

```
 1   initialize buffers for I/O
 2   set up board (clocks etc)
 3   load configuration
 4   clear registers
 5   get user input
 6   while not done
 7       if reset requested
 8           clear registers
 9       else
10           get current state of system
11           load input and pulse clock
12           get next state of system
13           report current state, event input
                 and next state
14       get user input
15   finish up
```

Figure 7: Pseudocode for driver program.

In the following, we describe the pseudocode steps that are unique to each implementation:

1. We establish a unique I/O buffer for each process in the implementation. We therefore need to know what the process IDs are for each process and where the state logic for each process is located on the chip (row, column, number of states, location of the initial state register). A buffer for the environmental inputs is created as well.

5. User input is in the form of an integer whose binary coding indicates the inclusion or exclusion of events as they appear in an ordered list. This binary code is loaded into the input register.

10. The program reads the contents of each state flip–flop for every process.

11. The user event input is loaded into the appropriate registers.

12. As for 10, but results are stored to next state variables.

13. The program reports the current state, event input, and next state for each process and every composition in the implementation.

D System description

This appendix describes the organization of, and procedures for, use of the computer-based files implementing the system. In the following sections, the file structure, instructions for use, and development tools are discussed.

D.1 File structure

Source and executable files for the HCircal compiler are located in subdirectories of the directory `$HOME/circal/compiler/`, with `$HOME` currently set to `/home/cisofd/` on the SPACE.2 host `pegasus` at the School of Computer and Information Science, University of South Australia.

This directory contains 6 subdirectories:

1. The source and executable for the front-end analysis and synthesis phases of the compiler are located in the `fe/` directory. This directory contains earlier versions in the `fe/rev/` directory and sample specifications in the `fe/spec/` directory.
2. The source and object library for the module generators is located in the `hclib/` (HCircal library) directory.
3. The `hcslib/` (supplementary HCircal library) directory contains a bit-stream generator source file and object library for an on-chip counter used by the host program to check correct clock pulsing of the SPACE.2 board.
4. Development and test source files for the back-end module generators are located in the `be/` directory. The `be/rev`*n*`/` directory with highest number n contains the latest version of the code.
5. The `dump/` directory contains the source and executables for utilities to check the configuration of the SPACE.2 board.
6. The `bin/` directory contains utilities and libraries used throughout the system.

Navigation, description, and use of the system is guided by the contents of `readme` files located throughout the directory structure.

D.2 Instructions

D.2.1 Compilation procedure

Front-end

The source for the compiler is the file `fe/hcc.c`. The `makefile` in the `fe/` directory may be used to compile this and link it with necessary libraries to build the HCircal compiler, `hcc`.

Back-end

The module `xx`, where `xx` is one of `b, ei, g, ij, is, mt, nis, ot, r, sl`, as described in Appendix B, has an associated `gen_xx.c` source file in the `hclib/` directory. The `makefile` in this directory can be used to compile these modules and to build the object library archive `hc.a`.

D.2.2 Running the compiler

Simple method (running on pre-specified source files)

Copy a specification from `fe/spec/*.hc` to `fe/t.hc` and type `make`. The `makefile` in the `fe/` directory will run the compiler, convert the generated XC6200 configuration bit-stream, `t.cal`, to the SPACE.2 configuration file, `t.els`, and will compile the generated host program, `t.c` before running it, thereby allowing the user to interact with the implementation via keyboard inputs.

Do it yourself (running on your own specification)

Create your own specification in the `fe/` directory. Name a copy of your specification `t.hc` and type `make`. See the "Simple method" section above for a description of the result.

Alternatively, enter the commands listed in `fe/makefile` with obvious filename substitutions to carry out the above procedure manually.

D.2.3 Resetting the SPACE.2 board

Occasionally the SPACE.2 board will hang or not provide expected results. To overcome this problem it usually suffices to clear and disable the FPGAs and/or to reinitialize the device driver with the commands `$HOME/circal/compiler/bin/resetFPGA` and `/usr/local/bin/resetspace 0` respectively. Should these procedure not clear the problem, a reboot of the host should be tried.

D.3 Tools

D.3.1 Tools for testing module generators

The module `xx`, where `xx` is one of `b, ei, g, ij, is, mt, nis, ot, r, sl`, as described in Appendix B, has associated with it in the `be/rev`*n*`/` directories one or more of the following files:

`xx.c` a standalone source file for entering module parameters and producing an XC6200 bit-stream fragment for the module. The `gen_xx.c` file in the `hclib/` directory is derived from this file.

`xxtest.c` which drives an automated test regime that generates a series of random parameters, calls upon one or more module generators to produce a bit-stream, and executes `testxx` to exhaustively test the module produced for each parameter set.

`testxx.c` loads a bit-stream and exhaustively tests the module for a given set of parameters by presenting it with all possible inputs and comparing its response with the predicted output.

`man_xxtest.c` allows the user to specify a parameter set that is used by `testxx` to implement and test the resulting module.

The `makefile` found in these directories can be used as an aid to build the corresponding executable files.

D.3.2 Tools for checking SPACE.2 configurations

The `dump/` directory contains files to check the configuration bits for a single cell, `c.c`, a utility, `dumpc.c` to list the configuration settings for switches and/or cells in a nominated region of a designated chip, and a utility, `dumpf.c`, that interprets the configuration bits and lists the boolean function and routing implemented by the cells in a nominated region of a designated chip.