Towards a Provably Correct Hardware Implementation of Occam

He Jifeng*, Ian Page and Jonathan Bowen**

Oxford University Computing Laboratory, Programming Research Group
11 Keble Road, Oxford OX1 3QD, England
Email: {Jifeng.He,Ian.Page,Jonathan.Bowen}@comlab.ox.ac.uk

Abstract. This paper shows how to compile a program written in a subset of occam into a *normal form* suitable for further processing into a netlist of components which may be loaded into a *Field-Programmable Gate Array* (FPGA). A simple state-machine model is adopted for specifying the behaviour of a synchronous circuit where the observable includes the state of the control path and the data path of the circuit. We identify the behaviour of a circuit with a program consisting of a very restricted subset of occam. Algebraic laws are used to facilitate the transformation from a program into a normal form. The compiling specification is presented as a set of theorems that must be proved correct with respect to these laws. A rapid prototype compiler in the form of a logic program may be implemented from these theorems.

1 Introduction

The development of systems containing software and hardware requires many levels of abstraction from requirements, through design and compilation to the underlying hardware itself. For confidence in the overall design, each level must be related and its correctness demonstrated [2, 3]. This is especially important in safety-critical systems where mistakes could cost lives [4]. Reduction in the overall complexity of the system is a key to increasing its likelyhood of correctness. One way to do this is to compile high-level programs directly into hardware, thus spanning several levels of abstraction at a stroke.

Here we show how to compile programs written in a subset of occam [17] (a particularly convenient language for the description of hardware because of its parallel programming constructs [8]) into a form suitable for implementation directly in hardware via a series of provably correct transformations. Crucial to our method is the use of *normal form* occam programs which refine the semantics of the user program and yet provide a representation close to the desired hardware. A final transformation is from the normal form into a *netlist* (a list of logic gates and latches, together with their interconnections) which is a standard form of hardware description. These netlists can be implemented in hardware

* Funded by ESPRIT Basic Research **ProCoS** project (3104 and 7071).
** Funded by the UK Science and Engineering Research Council (SERC) under the Information Engineering Directorate SAFEMOS project (IED3/1/1036).

in many ways. Currently, we use FPGAs which can be dynamically reconfigured by software [22, 28]. This enables us to build hardware implementations of modest-sized programs entirely by a software process.

Our source language is a small subset of occam which can be compiled efficiently into hardware and which can also serve as a target for a *front-end* compiler for a fuller version of occam, or indeed any other convenient language. Our compilation process preserves true concurrency which is represented in the user program by expressions, simultaneous assignment, and explicit parallelism. A significant feature of our hardware implementations is that only assignment and (ready-to-run) communication take time to execute, and they each take precisely one clock cycle. A particularly simple and elegant timing calculus results which enables our programs to meet real-time guarantees.

1.1 Background

This work builds upon previous results on provably correct compilation [13, 15, 16]. There is a strong relationship between our method and that used by Hoare in software compilation. However, our method handles communication and parallel composition and preserves true concurrency in the implementations.

Related work has shown that an occam program can be implemented as a set of special-purpose computers (one per process), each with just sufficient resources and microcode [20]. Martin has developed a method of compiling a concurrent program into a circuits using semantic-preserving program transformations [19]. A project at Cornell University aims to produce a multipass compiler through several levels of abstraction, but with much the same goal in mind [18]. Brown has suggested the possibility of compiling CSP or occam into asynchronous delay-insensitive circuits [5]. Further work on a process algebra called Joy has produced encouraging results [27]. Another working example of a 'silicon compiler' that synthesizes asynchronous circuits is [26].

Page has developed a prototype compiler in the functional language Standard ML which converts an occam-like language, somewhat more expressive than the one presented here, to a netlist [22]. This has been successfully applied to the control of a robot arm, amongst other applications. After further processing by vendor software, the netlist can be loaded into a Xilinx FPGA [28]. However, the normal form approach in this paper offers the significant advantages of providing a provably correct compiling method, and it is also expected to support a wide range of design optimization strategies.

2 A Language of Communicating Processes

In this section we present a simple language of communicating processes and provide a set of semantic-preserving program transformation rules. Our language is a subset of occam [17] from which local declarations have been excluded. Furthermore we do not consider skip-guarded alternatives. This subset is sufficient to illustrate our compiling method.

2.1 Syntax

For clarity of exposition and algebraic manipulation, the syntax of our language does not follow that of occam. In the following BNF-style syntax description, *ch*

will stand for a channel name, e for an expression, b for a Boolean expression, and x for a program variable.

$$P ::= \mathbf{skip} \mid \mathbf{stop} \mid x := e \mid ch\,?\,x \mid ch\,!\,e \mid P\,;\,P \mid P \parallel P \mid P \lhd b \rhd P \mid b * P \mid \mathbf{alt}\,(G)$$

$$G ::= ch\,?\,x \rightarrow P \mid G \,\Box\, G$$

Informally, the process terms stand for the following processes:

skip is a process which terminates immediately with all variables unchanged.

stop is the deadlock process, which may lose the values of its variables.

$x := e$ is a process which assigns the value of e to variable x, and its execution time is unspecified.

$ch\,?\,x$ is an input process which is willing to accept an input from channel ch and assigns it to variable x.

$ch\,!\,e$ is a process which is ready to output the value of e to channel ch.

$P\,;\,Q$ is the sequential composition of P and Q.

$P \parallel Q$ is the concurrent composition of P and Q. All communications between P and Q are concealed.

$P \lhd b \rhd Q$ is a process which first evaluates b; then if b is true it executes P, otherwise it executes Q.

$b * P$ is a process which is executed by first evaluating b; if b is false, execution terminates successfully, and nothing is changed. If b is true, it executes $P\,;\,(b * P)$.

alt (G) is an alternation of guarded commands. G can be either $ch\,?\,x \rightarrow P$ or $G_1 \,\Box\, G_2$. In the first case the process is prepared to input along channel ch and then behaves like P. Otherwise a choice is made between input actions on either side of the operator $\Box$. The standard interpretation is that the first guarded command to become ready is selected for execution.

Legal occam programs must satisfy further syntactic restrictions. In particular, no program variable can be shared by two concurrently executed processes if either of them can possibly modify it, and furthermore parallel processes can share neither input channels nor output channels.

2.2 Algebraic Laws

The basic laws defining occam programs are given in [25]. This section lists some example algebraic laws relating to normal form reduction selected from [11]. For simplicity we assume that all expressions always deliver a value.

Law 1: Refinement. We define a relation $\sqsupseteq$ between programs P and Q such that $P \sqsupseteq Q$ holds whenever, for any purpose, the observable behaviour of P is as good as, or better than, that of Q. $\sqsupseteq$ is an ω-complete partial order, i.e. it is reflexive, transitive and antisymmetric, and any ascending chain $\{P_n\}$ has a least upper bound $\sqcup_n P_n$ satisfying

$$\sqcup_n P_n \sqsubseteq Q \quad \textit{iff} \quad \textit{for all } i : P_i \sqsubseteq Q\ .$$

The aborting program $\bot$ is the bottom of the relation $\sqsupseteq$, and the miracle program $\top$ is the top. $\sqsupseteq$ has a greatest lower bound operator $\sqcap$, representing nondeterministic choice:

$(P \sqsupseteq R \text{ and } Q \sqsupseteq R) \quad \textit{iff} \quad (P \sqcap Q) \sqsupseteq R$

All occam constructors are continuous; i.e., they preserve the least upper bound of the ascending chain.

Law 2: Assumption and Assertion. We define an assertion as causing abortion if false

$b_\perp \stackrel{def}{=} \mathbf{skip} \lhd b \rhd \perp$

and define an assumption as a miracle if false

$b^\top \stackrel{def}{=} \mathbf{skip} \lhd b \rhd \top .$

The following laws apply:

2.1 $b^\top \sqsupseteq \mathbf{skip} \sqsupseteq b_\perp$

2.2 $b^\top ; b_\perp \sqsupseteq \mathbf{skip} \sqsupseteq b_\perp ; b^\top$

2.3 If e does not mention x then

$$x := e \;=\; (x := e ; (x = e)_\perp) \text{ and } x := e \;=\; (x := e ; (x = e)^\top)$$

Law 3: Loop Merge. The loop program $b * P$ is defined as the least fixed point of the following equation

$X \;=\; (P ; X) \lhd b \rhd \mathbf{skip}$

The following law is surprisingly important, mainly in proving the correctness of sequential composition:

3.1 $(b \vee c) * P \sqsupseteq (b * P) ; (b \vee c) * P$

Law 4: Loop Simplification. The notation $*(b \rightarrow P \,\Box\, c \rightarrow Q)$ represents the loop program $(b \vee c) * (\mathbf{if}\ b \rightarrow P \,\Box\, c \rightarrow Q\ \mathbf{fi})$.

4.1 If $b \wedge c \;=\; \mathbf{false}$ then $*(b \rightarrow P) \;=\; b * (\mathbf{if}\ b \rightarrow P \,\Box\, c \rightarrow Q\ \mathbf{fi})$

Law 5: Scope. The command $\mathbf{var}\, x$ introduces a new variable, and the command $\mathbf{end}\, x$ ends the scope of x. Variable introduction and end commands obey the following laws:

5.1 $\mathbf{end}\, x ; \mathbf{var}\, x \sqsubseteq \mathbf{skip} \;=\; \mathbf{var}\, x ; \mathbf{end}\, x$

5.2 If P does not mention variable x then

$$\mathbf{var}\, x ; P \;=\; P ; \mathbf{var}\, x$$
$$\mathbf{end}\, x ; P \;=\; P ; \mathbf{end}\, x$$

Law 6: Assignment. Assignment obeys the following laws:

6.1 $(x := e ; x := f) \;=\; (x := f[e/x])$

6.2 $(x, y := e, y) \;=\; x := e$

6.3 $x := e ; \mathbf{var}\, y ; y := f ; \mathbf{end}\, y \;=\; (x := e)$

2.3 Timed Processes

In the normal form used to describe the behaviour of a synchronous circuit we need to specify the execution time (in clock cycles) of assignments which mediate the state change of both control path and data path of the circuit. This allows reasoning about the real-time properties of the implemented programs. Let $n > 0$, the notation $(x := e)_n$ stands for the assignment $x := e$ whose execution takes n clock cycles. Let $\mathbf{skip}_n$ stand for a process which does nothing but delays execution for n clock cycles, and $(x := e)_0$ for the assignment which terminates immediately. We then have

$$(x := e)_n \;=\; \mathbf{skip}_n \,;\, (x := e)_0 \;.$$

The timed assignment $(x := e)_n$ can be regarded as a refinement of the untimed assignment $x := e$ since the latter does not impose any restriction on its execution time, thus

$$(x := e) \;=\; \sqcap_n \, (x := e)_n \;.$$

Law 7: Timed Assignment. Timed assignment obeys the following laws:

7.1 $(x := e) \sqsubseteq (x := e)_n$

The notation $P \lhd b \rhd_n Q$ represents a conditional which takes n clock cycles to evaluate its condition b.

$$P \lhd b \rhd_n Q \stackrel{def}{=} (\mathbf{skip}_n \,;\, P) \lhd b \rhd (\mathbf{skip}_n \,;\, Q)$$

In a similar way we define

$$b *_n P \stackrel{def}{=} \mu X . (P \,;\, X) \lhd b \rhd \mathbf{skip}_n$$

Further algebraic laws relating to real-time programming language aspects may be found in [9].

3 Normal Form Implementation of Occam

Normal form programs are a bridge between programs in a subset of **occam** and hardware implementations of them. The theorems presented in this section are sufficient to reduce a user program to normal form, where the normal form program is in an even more restricted subset of **occam**. Normal form programs can be interpreted as 'netlist' hardware specifications via a further transformation, which can be implemented using FPGAs, or by other conventional methods.

3.1 Normal Form Definition

A normal form program comprises three sequential programs where the first one designates the initial control state of the circuit, and the last one the final state. The other program is a loop with a simultaneous assignment as its body which specifies state changes of the computation, and the time delay caused by those changes. The normal form is essentially a state machine model for the system behaviour of a synchronous computation where the observables correspond to the following variables:

- a variable c representing the state of the control path of the circuit, which ranges over a set K of possible control states,
- a variable v representing the state of the data path of the circuit (for simplicity we assume v is of type integer),
- a K-indexed family C of expressions describing the next control state,
- a K-indexed family V of expressions specifying the new value of v.

A circuit with initial control state s and final control state $f \notin K$ can then be described by

$$\begin{aligned} \mathcal{N}(s, f, K, C, V) \stackrel{def}{=} \; & \mathbf{var}\, c \,;\, (c \;=\; s)^{\top} \,; \\ & *(\Box_{l \in K}\, c = l \rightarrow ((c, v := C(l), V(l))_1) \,; \\ & (c \;=\; f)_{\perp} \,;\, \mathbf{end}\, c \end{aligned}$$

where

- the assumption $(c = s)^\top$ means the circuit must be activated in state s,
- the assignment $((c, v := C(l), V(l))_1$ states that $C(l)$ is the successor of the control state l, and $V(l)$ is the value of v at that new state. Here the expression $V(l)$ does not mention variable c,
- the assertion $(c = f)_\perp$ guarantees that if the circuit terminates it will do so in state f.

The following lemma states that the real behaviour of the circuit is more predictable than that described by the normal form.

Lemma.

$$\mathcal{N}(s, f, K, C, V) \sqsubseteq \textbf{var } c ; (c = s)^\top ; \\ (c \neq f) * (\Box_{l \in K}\, c = l \rightarrow ((c, v := C(l), V(l))_1) ; \\ \textbf{end } c$$

The theorems given in the following section enable the automatic transformation of a user program to normal form. A compiler soundly based on these theorems can make some claim to being provably correct.

3.2 Normal Form Reduction Theorems

This section presents some of the reduction theorems by which an **occam** program can be transformed into a normal form, together with two sample proofs. The first three theorems handle primitive processes, and illustrate how to construct the corresponding normal forms directly. The remaining theorems deal with constructed processes with normal form programs as their operands.

Theorem 1: Skip.

$$\textbf{skip} \sqsubseteq \mathcal{N}(s, s, \emptyset, \emptyset, \emptyset)$$

Theorem 2: Stop.

$$\textbf{stop} \sqsubseteq \mathcal{N}(s, f, \{s\}, \{s \mapsto s\}, \{s \mapsto v\}) \quad \text{where } s \neq f.$$

Theorem 3: Assignment.

$$v := e \sqsubseteq \mathcal{N}(s, f, \{s\}, \{s \mapsto f\}, \{s \mapsto e\}) \quad \text{where } s \neq f.$$

Proof:

$$\begin{aligned} & v := e \\ = & \{ \text{ laws 6.3, 5.2 } \} \\ & \textbf{var } p ; v := e ; p := s ; \textbf{end } p \\ = & \{ \text{ laws 6.1, 6.2 } \} \\ & \textbf{var } p ; v, p := e, s ; \textbf{end } p \\ \sqsubseteq & \{ \text{ laws 2.1, 2.3 } \} \\ & \textbf{var } p ; (p = s)^\top ; p, v := f, e ; (p = f)_\perp ; \textbf{end } p \\ = & \{ \text{ law 7.1 and definition of loop } \} \\ & \textbf{var } p ; (p = s)^\top ; *(p = s \rightarrow (p, v := f, e)_1) ; (p = f)_\perp ; \textbf{end } p \\ = & \{ \text{ definition of } \mathcal{N} \} \\ & \mathcal{N}(s, f, \{s\}, \{s \mapsto f\}, \{s \mapsto e\}) \end{aligned}$$

Theorem 4: Sequence. In this and following theorems we state that language constructors are closed in the set of normal forms in the sense that if all the components of a constructor are in normal form, their composition can be reduced to normal form.

$$\mathcal{N}(s, h, K_1, C_1, V_1); \mathcal{N}(h, f, K_2, C_2, V_2) \sqsubseteq \mathcal{N}(s, f, K_1 \cup K_2, C_1 \cup C_2, V_1 \cup V_2)$$

provided that $K_1 \cap K_2 = \emptyset$ and $f \notin K_1$.

Proof:

$$\begin{aligned}
& LHS \\
= & \{ \text{ law 4.1 } \} \\
& \mathcal{N}(s, h, K_1, C_1 \cup C_2, V_1 \cup V_2); \mathcal{N}(h, f, K_2, C_1 \cup C_2, V_1 \cup V_2) \\
\sqsubseteq & \{ \text{ law 2.2 and assumption } \} \\
& \mathcal{N}(s, h, K_1, C_1 \cup C_2, V_1 \cup V_2); \mathcal{N}(h, f, K_2, C_1 \cup C_2, V_1 \cup V_2); \\
& \mathcal{N}(f, f, K_2, C_1 \cup C_2, V_1 \cup V_2) \\
\sqsubseteq & \{ \text{ laws 2.2, 3.1 } \} \\
& RHS
\end{aligned}$$

Theorem 5: Loop.

$$b(v) *_1 \mathcal{N}(s_1, f_1, K_1, C_1, V_1) \sqsubseteq \mathcal{N}(s, f, K, C, V)$$

if $s, f \notin \{s_1, f_1\} \cup K_1$, and

$$K \stackrel{def}{=} \{s, f_1\} \cup K_1$$

$$C \stackrel{def}{=} \{s \mapsto s_1 \lhd b \rhd f\} \cup \{f_1 \mapsto s_1 \lhd b \rhd f\} \cup C_1$$

$$V \stackrel{def}{=} \{s \mapsto v\} \cup \{f_1 \mapsto v\} \cup V_1$$

Theorem 6: Conditional. If $K_1 \cap K_2 = \emptyset$, and $f_1 \notin K_2$, and $f_2 \notin K_1$, and $s, f \notin \{s_1, f_1, s_2, f_2\} \cup K_1 \cup K_2$, then

$$\mathcal{N}(s_1, f_1, K_1, C_1, V_1) \lhd b(v) \rhd_1 \mathcal{N}(s_2, f_2, K_2, C_2, V_2) \sqsubseteq \mathcal{N}(s, f, K, C, V)$$

where

$$K \stackrel{def}{=} \{s, f_1, f_2\} \cup K_1 \cup K_2$$

$$C \stackrel{def}{=} \{s \mapsto s_1 \lhd b \rhd s_2\} \cup \{f_1 \mapsto f\} \cup \{f_2 \mapsto f\} \cup C_1 \cup C_2$$

$$V \stackrel{def}{=} \{s \mapsto v\} \cup \{f_1 \mapsto v\} \cup \{f_2 \mapsto v\} \cup V_1 \cup V_2$$

The theorems for communication, alternation and the parallel construct (and their proofs) are considerably more complicated that those presented here and thus cannot be included because of lack of space. However they are presented in the report on which this paper is based [11], for those who are interested in the full language.

4 Rapid Prototype Compiler

The compiling theorems shown here may easily be transformed into Horn clauses. Thus it is feasible to prototype them as a logic program [1]. However, to produce

an executable compiler, it is necessary to constructively generate each of the constructs of the language. This is relatively easy for the sequential aspects of the language. Theorems 1 to 3 may be transliterated very directly into a language such as Prolog [7]:

Clause 1: Skip.

```
skip <= n(S,S,[],[],[]).
```

Clause 2: Stop.

```
stop <= n(S,F,[S],[S->S],[S->v]) :- {S\=F}.
```

Clause 3: Assignment.

```
v:=E <= n(S,F,[S],[S->F],[S->E]) :- {S\=F}.
```

Constraints are encoded in curly brackets {...} for clarity.

Theorems 4 to 6 apply if all the components of the constructs are in normal form. However sequence, loop and conditional are monotonic w.r.t. $\sqsubseteq$:

If $P \sqsubseteq R$ and $Q \sqsubseteq S$ then $P\,;\, Q \sqsubseteq R\,;\, S$.

If $P \sqsubseteq R$ and $Q \sqsubseteq S$ then $P \lhd b(v) \rhd_n Q \sqsubseteq R \lhd b(v) \rhd_n S$.

If $P \sqsubseteq Q$ then $b(v) *_n P \sqsubseteq b(v) *_n Q$.

Also $\sqsubseteq$ is transitive:

If $P \sqsubseteq Q$ and $Q \sqsubseteq R$ then $P \sqsubseteq R$.

From these laws and theorems 4 to 6 we can derive the following new theorems:

Theorem 4a: Sequence.

$P\,;\, Q \sqsubseteq \mathcal{N}(s, f, K_1 \cup K_2, C_1 \cup C_2, V_1 \cup V_2)$

provided that $P \sqsubseteq \mathcal{N}(s, h, K_1, C_1, V_1)$, and $Q \sqsubseteq \mathcal{N}(h, f, K_2, C_2, V_2)$, and the constraints of Theorem 4 apply.

Theorem 5a: Loop.

$b(v) *_1 P \sqsubseteq \mathcal{N}(s, f, K, C, V)$

if $P \sqsubseteq \mathcal{N}(s_1, f_1, K_1, C_1, V_1)$, and the constraints and definitions of Theorem 5 apply.

Theorem 6a: Conditional. If the constraints and definitions of Theorem 6 apply, and $P \sqsubseteq \mathcal{N}(s_1, f_1, K_1, C_1, V_1)$, and $Q \sqsubseteq \mathcal{N}(s_2, f_2, K_2, C_2, V_2)$, then

$P \lhd b(v) \rhd_1 Q \sqsubseteq \mathcal{N}(s, f, K, C, V)$

From these we can formulate Prolog program equivalents very directly:

Clause 4: Sequence.

```
(P;Q) <= n(S,F,K,C,V) :-
  P<=n(S,H,K1,C1,V1), Q<=n(H,F,K2,C2,V2),
  {K1 disjoint K2}, {F notin K1},
  {K=K1\/K2}, {C=C1\/C2}, {V=V1\/V2}.
```

Clause 5: Loop.

```
B*P <= n(S,F,K,C,V) :-
  P<=n(S1,F1,K1,C1,V1),
  {[S,F] notin [S1,F1]\/K1},
  {K=[S,F1]\/K1},
  {C=[S->S1<B>F]\/[F1->S1<B>F]\/C1},
  {V=[S->v]\/[F1->v]\/V1}.
```

Clause 6: Conditional.

```
P<B>Q <= n(S,F,K,C,V) :-
  P<=n(S1,F1,K1,C1,V1), Q<=n(S2,F2,K2,C2,V2),
  {K1 disjoint K2}, {F1 notin K2}, {F2 notin K1},
  {[S,F] notin [S1,F1,S2,F2]\/K1\/K2},
  {K=[S,F1,F2]\/K1\/K2},
  {C=[S->S1<B>S2]\/[F1->F]\/[F2->F]\/C1\/C2},
  {V=[S->v]\/[F1->v]\/[F2->v]\/V1\/V2}.
```

Note that in the normal mode of usage, the high-level program will be supplied and the normal form derived. Without the disjointness constraints on variables, free (uninstantiated) variables will be returned. To satisfy the disjointness constraints, it is simply necessary to instantiate these to *different* values. Many versions of Prolog (e.g., Quintus [23]) provide a built-in clause to do just this. Using this technique avoids the otherwise very computationally expensive problem of checking the disjointness constraints. This results in a usable compiler in practice, at least for experimental purposes.

Compilation of the constructs associated with parallelism is less direct than from the theorems presented in this paper. However we plan to produce a fuller compiler based on proven theorems for hardware compilation. We feel that this is tractable since an unverified hardware compiler for the majority of occam, including all the constructs presented here, has already been produced in Standard ML [22], and is proving very successful in the practical production of netlist descriptions for FPGAs. The fact that logic (and other) programs can be considered as representing predicates [14] is a great help in producing a provably correct compiler. Logic program synthesis and transformation is a very active topic of research [6] and application of these results is likely to increase the confidence in and efficiency of the prototype hardware compiler. Prolog has been shown to be relatively efficient for compilation [21] and has even been used successfully in the compilation of the VHDL hardware description language [24].

5 Mapping Normal Form into Hardware

There are a number of ways in which a normal form program can be mapped into hardware. We have been using Xilinx FPGA chips to implement globally synchronous circuits which directly mimic the normal form programs. Firstly, we allocate latches corresponding to both control variables P and program variables v which together record the total state of the computation at each instant. Secondly, a set of combinational logic gates is generated to implement expressions V and C. It is also straightforward to develop theorems which refine the arithmetic and other operators in the language into Boolean operations only, so that the translation into hardware becomes trivial. Every latch in the implementation is triggered by the rising edge of the global clock and the clock cycle is defined such that the (loop-free) combinational hardware has settled well before the next rising edge which latches the next program state.

In practice the designer may adopt a specific method tailored to the physical resources at hand. For example, each control state can be given a latch when

the combinational circuits are the main concern in the implementation. Another extreme case is a set of combinational gates will be allocated to encode the control states.

Note that the mapping from normal form into hardware must also be proved correct for complete confidence in the compilation process. However the decomposition of the task into two or more phases helps to make the overall problem more tractable.

6 Conclusions

We have presented a normal form which acts as a bridge between a user program and its realization in a particular style of synchronous circuit. The normal form consists of very simple **occam** commands. We have shown some of the algebraic laws and theorems for transforming a user program into normal form. In processing synchronous communications, we have also extended the programming language by introducing the notation of state-based parallel which mimics the true concurrency of the underlying digit circuits [11].

It is possible to develop a hardware simulation program for the netlist interpretation of normal form programs. If this program is shown to refine the normal form program then we have a proof of correctness of the hardware netlist itself using the simulation as the defining semantics of the hardware components.

We have built an ad hoc compiler which directly generates hardware descriptions in a manner at least consistent with these theorems. We have also prototyped a small compiler in Prolog where there is very little code to obscure the application of the transformation theorems. We hope to build on this work to produce a compiler which is soundly based on our transformation laws and which can be trusted, with a high degree of confidence, to apply them validly. Even with such a compiler, it is prudent that we have another route by which the output of the compiler can be proven to refine its input. We will investigate ways in which we might also provide this facility.

Our objective is to produce a set of provably correct compiling theorems which enables us to implement **occam** programs as hardware circuits. Instrumental in our success for this study is the use of a simple normal form, which we have developed as an extension of some earlier work dealing with compiling specification in the **ProCoS** project [15, 16]. Our experience with this study is that while it was very difficult to establish the link between the event-based parallel paradigm and the state-based parallel one, the use of algebraic laws has aided this process.

The techniques above allow a microprocessor such as a **transputer** to be compiled into hardware from an interpreter description (specification) of the processor. What is more, the design may easily parameterized for different word lengths, sets of instructions, etc. Since the compiling process itself may be proved correct, confidence in *all* the processors produced is increased. This is in marked contrast to the more traditional formal verification techniques, in which only a single processor is proved correct, and represents a novel aspect of the proposed work. The approach is also derivational rather than proof-oriented in nature.

For our future work we plan to also consider hardware/software co-design. Currently only relatively small programs can be fully compiled in programmable hardware. Realistically, many programs will need to be compiled into a combination of machine object code and hardware. The split could be automated to some extent, although human guidance may well be desirable as well. An advantage of the approach is that new compilation strategies, such as optimizations, may be included as new theorems, without affecting existing theorems [10]. The ultimate aim is to provide a good interface with the engineer.

We see hardware compilation becoming increasingly important over the next decade. Currently FPGAs are mostly used for the implementation of glue logic. However, we envisage many more possible applications, such as direct implementation of algorithms in hardware to rival the speed of conventional supercomputers at a fraction of the cost. It will be important that appropriate software support is available to allow the convenient programming of such hardware. Currently size is a limiting factor, but since the technology is improving exponentially this will be of less concern in the future.

Acknowledgement. Sincere thanks are due to Prof. C.A.R. Hoare and Wayne Luk at Oxford University for comments and encouragement.

References

1. J.P. Bowen, From programs to object code using logic and logic programming, in R. Giegerich and S.L. Graham (eds.), *Code Generation – Concepts, Tools, Techniques*, Springer-Verlag, Workshops in Computer Science, pp. 173–192, 1992.
2. J.P. Bowen (ed.), *Towards Verified Systems*, Elsevier, Real-Time Safety Critical Systems Series, 1993. In preparation.
3. J.P. Bowen, M. Fränzle, E.-R. Olderog and A.P. Ravn, Developing correct systems, *Proc. 5th Euromicro Workshop on Real-Time Systems*, Oulu, Finland, 22–24 June 1993. IEEE Press, 1993. To appear.
4. J.P. Bowen and V. Stavridou, Formal methods and software safety, in H. Frey (ed.), *Safety of Computer Control Systems 1992 (SAFECOMP'92)*, Pergamon Press, pp. 93–98, 1992.
5. G.M. Brown, Towards truly delay-insensitive circuit realizations of process algebras, in G. Jones and M. Sheeran (eds.), *Designing Correct Circuits*, Springer-Verlag, Workshops in Computing, pp. 120–131, 1991.
6. T.P. Clement and K.-K. Lau (eds.), *Logic Program Synthesis and Transformation*, Springer-Verlag, Workshops in Computing, 1992.
7. W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 3rd edition, 1987.
8. G.V. Collis and E.J. Kappos, Occam as a hardware description language, *Software Engineering Journal*, **2**(6), 213–219, November 1987.
9. He Jifeng and J.P. Bowen, Time interval semantics and implementation of a real-time programming language, *Proc. 4th Euromicro Workshop on Real-Time Systems*, IEEE Press, pp. 110–115, June 1992.
10. He Jifeng and J.P. Bowen, *Specification, Verification and Prototyping of an Optimized Compiler*, Draft, Oxford University Computing Laboratory, 1992. Submitted for publication.

11. He Jifeng, I. Page and J.P. Bowen, *A Provably Correct Hardware Implementation of Occam*, ESPRIT **ProCoS** project document [OU HJF 9/5], Draft, Oxford University Computing Laboratory, UK, November 1992.
12. C.A.R. Hoare (ed.), *Developments in Concurrency and Communication*, Addison-Wesley, University of Texas at Austin Year of Programming Series, 1990.
13. C.A.R. Hoare, Refinement algebra proves correctness of compiling specifications, in C.C. Morgan and J.C.P. Woodcock (eds.), *3rd Refinement Workshop*, Springer-Verlag, Workshops in Computing, pp. 33–48, 1991.
14. C.A.R. Hoare, Programs are predicates, in ICOT (ed.), *Proc. International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, pp. 211–218, 1–5 June 1992.
15. C.A.R. Hoare, He Jifeng, J.P. Bowen and P.K. Pandya, An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language, *ESPRIT '90 Conference Proceedings*, Kluwer Academic Publishers, pp. 804–818, 1990.
16. C.A.R. Hoare, He Jifeng and A. Sampaio, Normal form approach to compiler design, *Acta Informatica*, 1993. To appear.
17. Inmos Ltd., *Occam 2 Reference Manual*, Prentice Hall International Series in Computer Science, 1988.
18. M.E. Leeser *et al.*, *BEDROC91: The Cornell Hardware Synthesis Project*, Technical Report EE-CEG-91-9, School of Electrical Engineering, Cornell University, Ithaca, New York, USA, 1991.
19. A.J. Martin, Programming in VLSI: from communicating processes into delay-insensitive circuits, chapter 1, in [12].
20. D. May, Compiling Occam into silicon, chapter 3, in [12].
21. J. Paakki, Prolog in practical compiler writing, *The Computer Journal*, **34**(1), 64–72, 1991.
22. I. Page and W. Luk, Compiling Occam into field-programmable gate arrays. in W. Moore and W. Luk (eds.), *FPGAs*, Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK, pp. 271–283, 1991.
23. *Quintus Prolog – Sun 4 User Manual*, Release 2.5, Quintus Computer Systems, Inc., Mountain View, California, USA, 1990.
24. P.B. Reintjes, A set of tools for VHDL design, in K. Furukawa (ed.), *Logic Programming: Proc. 8th International Conference*, The MIT Press, pp. 549–562, 1991.
25. A.W. Roscoe and C.A.R. Hoare, *Laws of Occam programming*, Theoretical Computer Science, **60**, 177–229, 1988.
26. C.H. van Berkel, J. Kessels, M. Roncken, R.W.J.J. Saeijs and F. Schalij, The VLSI-programming language Tangram and its translation into handshake circuits, *Proc. European Design Automation Conference*, 1991.
27. S. Weber, B. Bloom and G. Brown, Compiling Joy into silicon, in T. Knight and J. Savage (eds.), *Advanced Research in VLSI and Parallel Systems*, The MIT Press, 1992.
28. Xilinx Inc., *The Programmable Gate Array Data Book*, San Jose, California, USA, 1991.