PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Occam in the Specification and Verification of Microprocessors

A. W. Roscoe

Phil. Trans. R. Soc. Lond. A 1992 **339**, 137-151
doi: 10.1098/rsta.1992.0030

Occam in the specification and verification of microprocessors

By A. W. Roscoe

Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, U.K., and *Formal Systems (Europe) Ltd, Unit 7, The S.T.E.P. Centre, Osney Mead, Oxford OX2 0ES, U.K.*

Occam is a parallel programming language which can be used to describe VLSI circuits at several levels of abstraction. For a given piece of hardware one might have one Occam description which is close to the implementation level and another which is close to a specification in a notation such as Z or CSP. Thus a design can be substantially verified by a proof of equivalence of such descriptions. This can sometimes be achieved by using transformations based on the algebraic semantics of Occam and, even where this is not possible, the clean design and semantics of Occam make other formal techniques either easier or more reliable.

We discuss several case studies: the floating point unit of the IMS T800 transputer and various aspects of its successor, the IMS T9000. Despite the close relationship, on the surface, of these cases studies, radically different techniques were required for them. Based on this and other evidence, the author believes that one of the most important possessions when starting to tackle problems in hardware verification, at least at current levels of knowledge and technology, is an open mind.

1. Introduction

Very large scale integration (VLSI) design has become one of the most popular subjects for work in formal methods in recent years. The reasons for this are the increasing complexity of the circuits we are capable of building, the critical nature of the tasks many of them have to do, and the extremely high cost of a design error either before or after a product is released. Microprocessors have frequently been the subjects of this, though the greatest part of this effort has been devoted towards the complete verification of necessarily simple processors (see, for example, Cohn 1987; Hunt 1985). While we fully endorse the philosophy that eventually all microprocessors should be verified to the standard to which that work has aspired, it is for the moment true that the complexity of modern state-of-the-art processors such as the IMS T9000 keeps this type of work out of range. The increase in complexity of microprocessors seems at least to keep pace with our capabilities at verification.

Over the past six years I have participated in a number of collaborative projects with Inmos, the manufacturers of the transputer and the originators of Occam. Simply because it is not practicable to apply formal methods to a complete transputer, this should not stop us from using them sensibly in the development of these chips. This is what we have done, generally seeking to identify the parts of the development process which have most to gain from formal work, perhaps because of their novelty, subtlety or complexity, and to find the most appropriate formal

Phil. Trans. R. Soc. Lond. A (1992) **339**, 137–151
Printed in Great Britain

approach to the tasks chosen. One of the most important lessons we have learned is that it is vital to keep an open mind about what sort of formal techniques should be applied in these circumstances. For many different styles of design are used in a complex component such as a transputer, and the conceptual difficulties which give rise to the need for formal methods appear at several levels of abstraction.

The next section describes why Occam is well adapted for VLSI design and why it is comparatively tractable for formal verification. Since we are putting forward the thesis that there is no single approach which is always (or even usually) appropriate for the tasks that appear in this work, we illustrate by describing the wide range of techniques used by us so far in practical work on transputers. In the final section we attempt to form what general conclusions we can for all of this and point the way to future work.

2. Occam for VLSI design and formal verification

Occam is known as the language of the transputer, because it is implemented almost exclusively on that platform and matches very well with the transputer model of concurrency. It is in many ways a fairly low-level language, giving relatively simple ways of manipulating data and forming programs (insisting, for example, on statically determined memory requirements) with rather limited data-structuring capabilities. It is, in general, a language where access to store is carefully controlled with numerous anti-aliasing conventions. The most fundamental of these is its model of communication between parallel components: handshaken communication only – no communication via shared memory. This means that to model a parallel process it is enough to detail its potential communications (its explicit interactions with the outside world) and its final state (passed on to its sequential successors). It is not necessary to worry about the sequence of intermediate states which it passes through, all contingent on what might have been written to its variables by others during its run.

Almost any language can be given a formal semantics these days, and in principle these semantics could all be used as the basis of verification. The difference with Occam is that its relative simplicity, its strong anti-aliasing and its simple model of communication lead to a semantics which is elegant and relatively tractable in use. These features together with its predictable memory requirements make it well suited to programming safety critical applications, provided one allows for the possibilities of non-determinism which arise in concurrency, by showing it is absent or by formal verification.

Occam is well suited to use at intermediate stages of VLSI design because of its low-level features (providing convenient access to operations such as shift and bit-wise booleans) and because it allows one to write parallel programs naturally and easily. VLSI circuits are, after all, composed of a number of components all acting in parallel. It also has the considerable advantage of having an efficient implementation for running simulations. It has been used as an integral part of the design process for all versions of the transputer, there having been various tools created within Inmos for the automatic or semi-automatic translation of Occam (actually specific 'clichés' or sublanguages which map naturally onto specific types of circuit) to silicon via intermediate forms.

Though its communication mechanism maps most obviously onto self-timed logic, and this indeed has been the subject of several pieces of work on silicon compilation

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

(see, for example, Brunvand & Starkey 1991; Brown 1991), it is entirely possible to model clocked circuitry in Occam, something we shall come back to in discussing the T9000 pipeline later. It is almost certainly easier to model synchrony in an asynchronous language like Occam than the reverse, and it would be an excellent language for describing mixed-mode systems with locally clocked components linked together. Page & Luk (1991) indicate how an Occam program can potentially be compiled into a mixture of synchronous and asynchronous circuitry and ordinary assembly language on processor(s).

The semantics of Occam are rather like the product of those of CSP (Hoare 1985), a calculus dealing with pure communication and no state, and of a simple imperative sequential language (perhaps Pascal without pointers). This comes across most clearly in denotational semantics (Roscoe 1985). The clarity and usability of these is greatly helped by the facts that it is not necessary to use continuations, and that the treatment of errors can be unusually straightforward. As one might expect, given this 'product', the techniques required for specifying and verifying Occam programs are either those developed for dealing with one of the communication and state sides in isolation or need to be able to handle both sides at once. Examples of the former are, on the one hand, using Z with pre/post assertions for programs using no communications or parallelism and, on the other, the analysis of communication patterns to prove deadlock- or livelock-freedom.

The methods we have used for dealing with communication and state aspects together include some fairly straightforward extensions to techniques developed in CSP for analysing the traditional ills of concurrency: non-determinism, deadlock and livelock. The more interesting ones include program transformation based on algebraic semantics and symbolic execution; these are dealt with under the IMS T800 and IMS T9000 pipeline later.

In making the design and application of these mixed methods possible and safe, the simplicity and anti-aliasing of Occam have been crucial.

3. Formal methods and the IMS T800

Our first case-study is provided by the design of the Floating Point Unit (FPU) of the IMS T800 transputer. This has been well reported elsewhere (see, for example, (Barrett 1987; May & Shepherd 1988; Shepherd 1988)), but it is worth repeating a few technical details here, both as an illustration of our philosophy and to allow the reader to compare it with the T9000 work described later. The great majority of the specification and verification work described in this section was carried out by Geoff Barrett and David Shepherd.

The FPU was built like a CPU with a 64 bit datapath (divided into sections for dealing with the mantissa and exponent of floating-point numbers), with appropriate registers and wiring for manipulating the data. Floating point instructions were implemented on this engine as microcoded routines, so that each FPU operation was in fact implemented as the combination of a number of simpler ones. It was felt that the greatest risks in the design of the FPU were not in the construction of this engine for implementing the microcode, but rather in the development of microcode which correctly implemented IEEE Standard 754: a document written in English which sets out to specify the effect of every floating-point operation. (A design decision had been taken not to seek to implement all of the exception handling required by the standard.)

Even though these microcode operations were at a much higher level than a gate-level analysis of the circuit, they were still remote from the specification. This gap was bridged firstly by the translation of the IEEE specification into the Z notation, which put the specification into a format where it was capable of formal proof by established correctness techniques. This specification was then used to derive a suite of proven-correct Occam routines for executing floating-point instructions. These routines were then proven equivalent, via a series of transformations, to very stylized Occam programs which were a direct model of the microcode.

It was only in this last part of the work that a machine assistant (tool) was used, the Occam Transformation System (Goldsmith 1987; Goldsmith & Roscoe 1988). This is a tool built around the algebraic semantics of Occam (Roscoe & Hoare 1988), a complete set of laws which by their nature preserve the semantics of programs. It can be programmed with high-level strategies such as normalization, but in the IMS T800 work it only played the role of applying series of simple transformations designed by the human user (so playing much more the part of a proof checker or assistant rather than a theorem prover). While tools are certainly very useful in the application of formal methods, we believe that this piece of work demonstrates that, with well-chosen methods and levels of abstraction, it can be both possible and cost-effective to proceed without them. Indeed it would be a great mistake to feel constrained to use a particular method simply because one possessed a particular tool.

The work on the FPU proved very successful as compared to traditional development techniques (Shepherd & Wilson 1989). It has recently (1990) led to the Queen's Award for Technological Achievement being jointly conferred on Oxford University Computing Laboratory and Inmos.

4. Approaching the IMS T9000

The IMS T9000 is a much more sophisticated chip than its predecessors. Thanks to developing technology it is now possible to get far more transistors onto one chip. It is not, however, possible to get comparable increases in clock speed (the advertised clock rates of the IMS T800 series and the IMS T9000 range up to 30 MHz and 50 MHz respectively). The extra processor speed expected of a microprocessor today, has therefore, had to be largely bought by increased design complexity rather than clocking. It is clear that this trend is going to continue at least into the medium term, so that much better techniques are going to be required to modularize the designs, and to be able to reason about overall behaviour knowing only the high-level interface specifications of the modules. It is increasingly impossible for one person to have more than a high level understanding of how one of these chips works.

One might reflect that the source of all this complexity is the perceived need to have a component which is able to execute sequential code very fast. The sort of techniques which are resorted to are illustrated by our discussion of the pipeline later. The same amount of processing power can be developed rather more cheaply by simply replicating parallel units to run on the same or separate chips.

Our work on the T9000 was previously reported in Roscoe *et al.* (1991). This paper has a slightly different emphasis and brings the story more up to date.

Given that formal methods could not be used on the whole design, it was decided they would be best applied in areas where the new chip does things in different and more complex ways than previous ones, not ones where it was simply faster or had

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

more of something. The following major areas of increased sophistication can be readily identified in the T9000.

(*a*) *Communications support*

The introduction of more-or-less unlimited connectivity, to override previous transputers' interprocessor constraints of communication with only four or eight nearest-neighbour processes, through virtual channels and the IMS C104 dynamic crossbar switches, is probably the major qualitative difference in the new generation of parts. By allowing convenient and efficient routing around an arbitrary-sized network, and given the transputer's fast process-switching capability, these new features will make the T9000 very close to an ideal component for a general-purpose parallel computer in the sense of Valiant (1990).

The VCP (virtual channel processor) is the part of the T9000 which manages this. A process which is communicating off-chip has control passed to the VCP, which multiplexes a practically unlimited number of logical channels down the four physical ones. The VCP, the IMS C104 and the communications mechanisms they support all present interesting problems in formal verification.

The various local proofs of desirable properties of parts of the system should eventually combine to establish conformance with an overall specification: roughly speaking, that as far as communications are concerned, an T9000/C104 network should have the (untimed) behaviour that a corresponding network of directly connected transputers from earlier generations would, if only they had enough links.

A number of minor verification tasks were attempted here, such as proving the deadlock-freedom of the wormhole routing strategy used. A substantial exercise is being carried out by Geoff Barrett and Victoria Griffiths of Inmos, with the support of tools developed by Barrett, Michael Goldsmith, David Jackson and the author to prove that the implementation of communication multiplexing with associated features in the VCP is correct. This has involved formalizing the behaviour of the implementation in CSP, creating a specification level CSP description, and formulating abstract correctness conditions (simple liveness and safety properties). The basic tool used has been a refinement checker for finite-state CSP: one shows that the implementation refines the specification and that the specification satisfies the correctness conditions. The tool uses a combination of normalization (at the 'specification' end of the refinement) and model-checking techniques. This work will be described in more detail elsewhere; lack of space precludes them here.

(*b*) *Memory model*

The memory model of the T9000 is rather different from previous transputers, since it is based around using the on-chip RAM as a cache. The user can decide to use all, some or none of the 16K of memory as a cache. All memory accesses are then made through this cache, which makes a fetch from off-chip memory in the case of a cache miss. There are quite a large number of different components of the T9000 which can make demands on the cache, and it is necessary to perform arbitration between these to decide who gets served in which orders. Possible problems which could arise if this were got wrong would include deadlocks or livelocks if the arbiter indefinitely preferred other parts of the T9000 over the ones which were actually holding up progress-in-the-large pending a memory operation. Other obvious problems include the integrity of the cache and making sure that individual load and store operations are correctly executed.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

A number of issues concerning the memory model have arisen from the work on the pipeline (see below): consistency issues and contention for resources between different parts of the pipeline for memory resources. Work on the more general interactions with, and contention for, memory really forms part of a general model of interactions between parts of the chip. Only a limited amount of work has gone on at this level, unfortunately. It is probably here, at this highest level where we are trying to establish the shape of the overall behaviour of the chip, that the relatively late start to the formal methods work has made things most difficult. If we are to reason about and specify high-level interactions between components, it is vital to get a methodology established where the high-level specifications, interfaces and interdependencies are established as early as possible in the design process.

(c) *Pipelining*

Probably the greatest challenge arises in the pipelined design of the CPU. Instead of executing its instructions one at a time, the T9000 sends them through a pipeline consisting of the following stages.

1. Local cache (or workspace cache), which builds up a copy of the area of memory immediately above the current value of the workspace pointer. This allows a local read from this frequently accessed portion of memory usually to be performed cheaply at this stage, reducing the load on cache and memory bandwidth and expediting the availability of data for the following stages.
2. Address calculation, which carries out the linear combinations necessary for computing the address in memory which a given access needs (whether a read or write, and whether or not it is an array access).
3. Read stage, where data is fetched from the cache (and, through this, from the external memory) in response to non-local load instructions.
4. ALU/FPU, which carries out arithmetic operations on the data held in the integer and floating-point stacks respectively.
5. Write stage, to which most operations which have an irreversible effect on machine state are delayed, including stores to memory. The reason for needing to delay such things is that the pipelined nature of the CPU means that an instruction can make its way down the pipeline even though, ahead of it, there is another one which will make a (conditional) jump or generate an exception which means that it should not actually get executed at all. It is thus necessary that we can *flush* the pipeline from various points and make the effects of all instructions which were flushed disappear.

To get the best advantage from this pipelined structure, the various resources in it must be used most of the time. If single instructions were to pass down the pipeline then each would probably use only one or two stages. Therefore, instructions are *grouped*, and passed down the pipelines in bundles of up to eight. The groups must be selected from the instruction stream in such a way that they can in fact be executed correctly as a group as they pass down the pipeline together. Roughly speaking, the next group to enter the pipeline should be the longest prefix of the instruction stream with this property.

Thus, aside from the stages which fetch instructions from the cache (the instruction buffer) and form groups (the instruction grouper/decoder) there can potentially be 40 instructions in the groups active in the pipeline at any one time. This represents a high degree of actual parallelism, but if the pipeline design is correct we should not notice this. For the main specification of the pipeline is that

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

it should execute a given program with precisely the same effect as though it had been executed on a simple machine which executed them one at a time.

A separate, though closely related, issue comes from the class of single instructions (for example, ones which communicate) which have a relatively complex effect and thus need to be microcoded. In other words, the execution of one of these instructions actually involves running a simple program held in a microcode ROM. It is necessary to check that these microcode programs are correct, that they are executed properly on the pipeline, and that the way in which these programs fit into the general stream of groups on the pipeline has the desired effect.

The formal methods work we have carried out on the pipeline is discussed in the next section.

5. The pipelined CPU

The work described in this section has been carried out by Formal Systems personnel and consultants, with the active cooperation of the design team within Inmos.

(*a*) *Decomposing the problem*

The chief targets of our verification efforts on the pipeline have been a pair of simulators written in Occam. One (the instruction simulator) gives a specification of the effects of each instruction on the state of the machine (its memory, stacks and other registers). These specifications are given both in a Z-like notation (in the style of Inmos (1988)) and executable Occam. This simulator can be thought of as the specification of the pipeline: running a program on the T9000 should always affect the states in the same way, modulo any non-determinism which is allowed resulting from accessing undefined stack values, etc.

The other simulator provides a detailed model of the pipeline as implemented, with each stage being implemented as a separate process in the overall parallel structure, with communications taking place on a large number of channels each cycle to represent the transfers of data between them (the clock signal also being distributed by special channels). The production of this simulator has been an integral part of the design process at Inmos.

The simulator models the silicon closely in all areas directly relating to the pipelining process, including group recognition and decoding, data switching and feedback, and the decisions which are taken at various points in the pipeline as to which operations are to be carried out on data. The pieces which remain relatively remote from the silicon are chiefly those which carry out these functions, such as arithmetic and FPU operations. These were all thought to be relatively well understood, and their only functions within the pipeline are to carry out self-contained operations on data. Thus our analysis will be able to show that the T9000 pipeline executes the correct function in the address stage, ALU or FPU, but will not show that these functions are executed correctly. If it were desired to verify these parts as well, the correct way to do it, given their self-contained nature, would be to look at each of them separately rather than as part of the pipeline. Then we could combine these results with the overall pipeline result we are discussing, to obtain a stronger theorem.

The two simulators are large programs, remote from each other in structure. Much the more complex is, of course, the pipeline simulator. Given the scale of the problem it was not feasible to attempt any sort of direct proof of the congruence (via program transformation, for example). Rather it was necessary to address the proposed result

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

about equivalence on every sequence of instructions directly, by gaining a thorough and formal understanding of how the pipeline simulator (and thus the pipeline itself) works. It was decided to factor the analysis into three levels, which are summarized below.

(*b*) *High-level progress*

At the highest level we have constructed an abstraction in CSP of the overall behaviour of the pipeline, in which the behaviour of each pipeline stage is represented by the actions it can take which affect the overall progress of groups through the system.

At this level we are not concerned with the constitution of individual groups or of the data they manipulate, merely with the shape of inter-stage communications. Thus we are interested in the different ways in which one stage can communicate with another, and how a group might do such things as split, recombine, stall, flush or be flushed.

It is important to remember that the real pipeline is a clocked component, like the rest of the T9000, and that all of the wires joining its parts take a value on every cycle, with no implementation of a handshake at this level. However, most of these signals are either invalid, are repetitions or form part of some higher-level protocol implementing something like a handshake, and it is generally possible to get closer to the 'essence' of progress in the pipeline by CSP. The style of CSP descriptions used – each component is represented as a one-step tail recursion apart from its flushing behaviour – is specifically tailored to this situation. On a given cycle one of these tail-recursive processes may make no progress at all, make one move or might go through several.

Of all the communication in the pipeline, the part which is most obviously analogous to a handshake is the progress of groups between stages. In the absence of splitting, recombination or externally generated stalls, this would be like the passage of data through a sequence of one-place buffers. Other forms of communication represent what one might term a 'virtual handshake' which is valid because (for example) either the receiving end is invariably willing to accept a communication (the validity of which is indicated with the communication) or because it can be guaranteed that an output will be continuously available until accepted.

In constructing the CSP model we provide a commentary on how the various handshakes contained therein reflect the actual interactions in the pipeline (in other words, under what circumstances we deem the communication to have taken place). Provided that the requirements which it sets out are properly implemented, it becomes straightforward to show that the pipeline itself always makes progress in some useful sense. Also by forcing us to think in a slightly different way about how groups and stages interact, it provides a valuable source of insight into the verification process for inter-group communication. The need to formulate conditions for CSP communication gives rise to a number of natural correctness conditions on the communications of the clocked system. These are being checked by manual and automated techniques as appropriate.

We do not have the space to give the full description here, and it is not appropriate to do so for reasons of confidentiality. Nevertheless we can give a flavour of the style by describing the behaviour of a generic stage relative to the input and output of groups and fedback stack information.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

We will describe processes $Empty(S)$ (representing the stage when it is empty) and $Full(S_{in}, S_{out})$ (with a group in) where the parameters are S, the stack information (by which we mean valid data from the group ahead or final state) fed back to here but not yet communicated further back. Thus it is waiting to be fed back further or to be picked up by the next group that arrives. S_{in} and S_{out} are the stacks so far input, and so far output, by the group currently occupying a full stage.

All three are partial functions from the stack registers accessible to this stage (SR), perhaps $\{A, B, C\}$, to $Data$. The stack registers accessible to a given stage can be divided into two sets: BTH (back to here) is the set of fedback values which are fed back no further than this stage, and BBH (back beyond here).

$$Empty(S) =$$
$$\text{in}\,?\,\langle S_{in}, S_{out}\rangle \rightarrow Full(S_{in} \cup S, S_{out})$$
$$\Box(\Box\{Fb\,In.X\,?d \rightarrow Empty(S \cup \{(X,d)\}) \mid X \in SR \backslash dom(S)\})$$
$$\Box(\Box\{Fb\,Out.X\,!d \rightarrow Empty(S \backslash \{(X,d)\}) \mid X \in BBH \wedge (X,d) \in S\})$$
$$Full(S_{in}, S_{out}) = Stack \Box (Groups \nleftarrow dom(S_{in}) = SR \nrightarrow (STOP \lhd Groups).$$

The conditional expression used here reflects the fact that a group may stall if it has not yet got all fed back information. Here (taking the parameters implicitly over to the macro definitions of $Groups$ and $Stack$).

$$Groups = out\,!\,\langle S_{in}, S_{out}\rangle \rightarrow Empty(\varnothing)$$
$$Stack = (\Box\{FbIn.X\,?d \rightarrow Full(S_{in} \cup \{(X,d)\}, S_{out}) \mid X \in SR \backslash dom\,(S_{in})\})$$
$$\rhd (\Box\{FbOut.X\,!\hat{d} \rightarrow Full(S_{in}, S_{out} \cup \{(X,d)\}) \mid X \in BBH \backslash dom(S_{out})\}).$$

The operator $P \rhd Q\,(= Q \lhd P)$ is shorthand for $Q \sqcap (P \Box Q)$: the implementation may choose to offer the first steps of P, but it need not. It must offer the first steps of Q. Notice that this says that the output of a stack value by a group at this stage is optional: it may not be ready yet. The special notation $!\hat{d}$ in $FbOut.X\,!\hat{d}$ means that the choice of the output stack value is, from the point of view of this abstraction, non-deterministic. The point of having it there at all is only to let us formulate consistency conditions on communication.

The full model has to treat a variety of other behaviours, and the various stages all have their own special characteristics. The above should allow the reader to see the essence of our approach. Certainly the final result gives a pleasingly concise description of the high-level behaviour of the pipeline.

The CSP tools constructed for the verification of the VCP will be able to analyse the pipeline description for deadlocks (which would represent states where the system was making no 'real' progress), livelocks (which would represent situations such as memory contentions) and similar misbehaviours.

(*c*) *The inter-group model*

At an intermediate level, it is necessary to check that the interactions between groups and the management of memory are correct. Essentially this comes down to showing that the effect of the stream of groups is the same as it would have been if the groups had been executed one at a time on a sequential processor for which they were (complex) instructions.

We can break down the work in this section into a number of tasks, each of which

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

has a number of subtasks. The major points which require verification here are as follows.

1. In the execution of any sequence of groups, the various inputs (in a high level sense) received by each group should be exactly the same as they would have been if we had waited until all earlier groups had finished their execution before inserting it. In other words, the various stack and other registers accessible to the group, and the values of memory locations it may read from the caches, must have the same values when it sees them as they would have done if we had waited as described above.

2. The integrity of the caches (including the relationships between them).

3. The sequence of instructions contained in the groups actually executed but not flushed is correct, and that any group which is flushed does not change the state.

The analysis at this level consists of a large number of pieces of work which are very varied. Many of them involve the verification of individual pieces of code, while others are at a higher level. A number of them are generated from the CSP discussed above. Typical exercises which have been, or are being, carried out include a detailed demonstration that the group-movement régime implemented is a correct refinement of that implied by the CSP specification, and an automated check that the communications meet the consistency conditions implied by the commentaries on the virtual handshakes.

(*d*) *Individual groups*

The final part of our factorization is to show that, given that the information it gets from other groups in the pipeline and from elsewhere is consistent, that the effect of passing a single group of instructions down the pipeline is the same as executing the same instructions on the instruction simulator.

This work splits into three parts: the first is to examine the formation of groups, the second is to prove our theorem for those groups of non-microcoded instructions which might get into the pipeline proper, and the last is to check the execution of the microcoded instructions.

(i) *Instruction fetching and decoding*

The 'front end' of the pipeline has a number of components for fetching and buffering instructions from the cache, and for forming groups from the resulting instruction stream.

The analysis of these components breaks into three separate parts. The first involves checking the harness which feeds and holds the grouping logic itself, and which executes the latter's decisions. This turns out to be largely an examination of the way in which the instruction buffer and decode stages refine their parts of the CSP description, together with a number of fairly unexciting proofs of parts of them. The most critical aspects are perhaps concerned with the resetting of instruction pointers upon jumping and flushing. The second, which we discuss below, involves verifying the grouping logic. The third is the verification of the outputs from the decoder to the pipeline. This is largely subsumed into the checking of the pipeline proper discussed below.

The logic which decides which groups can be passed down the pipeline has to be checked, both against the known capabilities of the pipeline and against written specifications. This is done by a combination of manual analysis and specially constructed tools. More details can be found in Roscoe *et al.* (1991).

(ii) *The actions of groups and symbolic execution*

The last part of our factorization is to prove that the groups of instructions which pass down the pipeline together are implemented correctly: produce the same effect as the same sequence of instructions on the instruction simulator.

The T9000 pipeline executes groups by carrying them down the pipeline, carrying out operations on their data and switching data around in the various stages as demanded largely by control wires. Data is created and modified in relatively few places, but *which* data is acted on at various stages is often hard to discern because of the complexity of the switching of data which occurs between various registers and parts of the datapath. Each group can be thought of as having a number of inputs and a number of outputs. The inputs are the operands of any primary opcodes, the stack data which the group inherits, the workspace pointer W and instruction pointer I. (Since any reads from the memory access pre-existing values, it is far more interesting, for this analysis, *where* values are read from than what the values are; and so we do not need to regard the state of the memory as an input.) The outputs are these same objects (apart from the operands) plus the address, type and data of any write, and possibly a flush signal. These outputs are all calculated in a few steps from the inputs at well-defined places in the pipeline, the most obvious places where values are changed (rather than just switched) being (i) the introduction of constants at various places (*ldc*, *mint* and floating-point zeros); (ii) local address calculation ($W+4n$) and loads ($@(W+4n)$) in the local cache stage; (iii) address calculation (linear combinations such as $A+4B+4n$, from *wsub* and *ldnlp*, etc.) in the address stage; (iv) indirection (e.g. $@A$) in the read stage; (v) the application of arithmetic and logical operations in the ALU and FPU.

Under the assumption of some correctness conditions surrounding inter-group interactions, the behaviour of each group is completely described by a symbolic representation of the various outputs which describes how they have been calculated in terms of the inputs and operations. Individual instructions (for this purpose including their operands) have representations which can readily be computed and automatically combined (at least, subject to the condition that writes are more-or-less final in a group).

We adapt the simulator to compute a symbolic representation of all the output values it produces, without otherwise affecting its behaviour, by a technique we term 'ghosting'. This involves carrying around, together with each identifier which contains relevant information, a separate *ghost* identifier, which is assigned or communicated over a channel whenever the original is, and which has symbolic versions of any operations carried out on it shadowing ones carried out on the original. Ghosting is a sophisticated batch transformation, described in detail in Roscoe *et al.* (1991). The only significant difference between the version used and that reported there is that we have found it essential to automatically generate the symbolic representations of all operations carried out on a ghosted type, rather than leaving them to be done manually as was described in the earlier paper.

It is worth noting that the strong anti-aliasing rules in Occam make ghosting far more worthwhile and reliable, since it is possible to identify in a program exactly the places where a particular value is being modified. Pointers, in particular, would make our job almost impossible.

The ghosting transformation requires a second one, *defunctionalizing*, which factors expressions with function uses with array result types into cascaded

wsub	ldnl	ldc	ldlp	stnl	gajw
A = +A<<B2	A = @+<<i2A	A = i	A = +W<<i2	A = C	A = W
B = C	B = B	B = A	B = A	B = u	B = B
C = u	C = C	C = B	C = B	C = u	C = C
I = +I1	I = +I1	I = +I1	I = +I1	I = +I1	I = +I1
W = W	W = W	W = W	W = W	W = W	W = A
				WriteAddr = +<<i2A	
				WriteData = B	

Figure 1. Some sample instruction specifications.

procedure calls. It is necessary because Occam, in common with several other languages, does not allow such functions, but they arise naturally from several aspects of ghosting.

It is important to remember that the symbolic execution will prove that the pipeline does the right thing to a group only on the assumption that the various functions which are carried out on data (adding, FPU operations, etc.) are correct. In other words, it simply proves that the data routing and control signals are having the desired effects.

In addition to the ghosting process described above, the simulator is also altered to allow us to control the flow of groups through it and to extract the symbolic values produced. This is, however, done in a way which minimally intrudes on the rest of the behaviour of the simulator, as it is important to be sure that nothing works in the ghosted simulator only because of the monitoring.

Having constructed this transformed simulator, it is tested by passing representatives of a carefully selected subset of possible instruction groups through it. (The criteria for selection are broadly in line with those already discussed under the verification of the grouper. All groups not tested must be known to be covered in a formal sense by one or more which is. The size of the subset of groups dealt with depends on exactly what is being verified, but has typically been of the order of 10^4–10^5 of the approximately 2×10^6 allowed groups.) It is also necessary to test some of these groups under different conditions to achieve certain varying control flow that can arise from tests on data values.

The symbolic outputs of each group are automatically compared with the symbolic outputs obtained from composing the symbolic representations of the individual instructions it contains. These will have been obtained by running the pipeline on the instructions by themselves and checking the results against Z specifications and the instruction simulator. In some cases the Z specifications state that one or more of the stack registers becomes undefined, and if so the implementation can place any values it chooses in them. The symbolic compositions of the specifications of such instructions may produce undefined values in resulting groups. The comparison of the specification and implementation allows appropriate refinements.

Some examples of the specifications of individual instructions and the results of

```
ldc ldnl ldc wsub          ldc ldnl stnl

A = +k<<@+<<j2i2           A = B

B = A                      B = B

C = A                      C = B

I = ++I31                  I = ++I21

W = W                      W = W

                           WriteAddr = +<<i2@+<<j2i

                           WriteData = A
```

Figure 2. Two sample results of groups.

typical groups are given in figures 1 and 2. The format is prefix notation (sometimes called Polish); this saves symbolic space by not requiring brackets. The three stack registers, the workspace pointer and the instruction pointer have the obvious symbolic representations (`A`, `B`, `C`, `W`, `I`). The various operands to a group are represented by the lower-case letter `i`, `j`, `k`...(`i` being the first, and so on). A dereference (i.e. a read of a memory location) is represented by `@`. `u` represents an undefined output value. The symbolic effects on the instruction pointer in these cases are simply to add the number of bytes required for the particular example of that instruction in use when the computation was done. (Primary transputer instructions vary in length depending on the number of bytes required to construct the operand.)

At the time of writing we have been carrying out this analysis on successive iterations of the simulator for a period of approximately seven months. Each time a complete set, or large representative sample, of the possible groups were tested symbolically. It has uncovered a number of small errors, and has also been able to demonstrate the correctness of large parts of the design.

Though combinatorial problems, as well as non-determinism in relative speeds of passage through the pipeline, mean that it is not such a powerful tool here, the ghosted version of the simulator might be used to test a number of features of inter-group interaction either as part of, or to increase confidence in, that part of our work.

(iii) *Microcode*

As described earlier, some of the more complex instructions, such as *call* and *lend* and those which execute communications, are microcoded. This means that they are run as programs held in a ROM. These programs are a combination of ordinary and special instructions executed on the pipeline. There are two main parts of the proof of microcoded instructions: proving that the microcode used is correct, and proving that it is implemented properly on the pipeline. The second of these tasks is similar to that discussed above, with similar techniques used to carry it out.

The first is closely related to that carried out for the IMS T800 floating-point microcode described earlier.

Symbolic execution has proved invaluable in dealing with microcode, just as with the groups of simple instructions. It has to be treated more carefully, though,

because of the much greater range of conditional behaviour in microcoded instructions.

6. Conclusions

Perhaps the most obvious thing about the case studies above is the diversity of techniques applied. Even when viewed at the same level of abstraction as a typical high-level program, a microprocessor such as the T9000 is incredibly complex and heterogeneous, so we should not be surprised at this.

We would have wished to have begun the formal development work in every one of the cases earlier than was in fact done. In each case the formal effort joined an existing informal one, early enough to ensure that the design conformed to specifications at some level, but not early enough to greatly influence the general concept or to establish the specification *ab initio*. (This statement is only partly true for the IMS T800 FPU since the IEEE specification already existed.) There is good reason to believe that more could have been achieved if formal effort had begun earlier, since (i) it is generally accepted that it is easier to build a program correctly from a formal specification than to verify one that already exists, and (ii) a lot more could have been done at the level of the interfaces of the components of the chips, and therefore to overall behaviour rather than simply dealing with individual components.

Nevertheless we have generally been very pleased with the way our techniques have coped with the great complexity of the problems tackled, particularly with respect to the T9000 pipeline.

It is worth noting that, although extensive use has been made of automated tools in our work on the T9000, almost all of these have been written specifically for this project.

The even greater complexity of the generation of microprocessors beyond the T9000 will make the establishment of clear specifications of their components even more important. The complexity of the designs will lead to larger teams, very possibly across a number of companies because of the cost. The cost factor will place greater requirements on the management of the design effort and will create a much stronger incentive to re-use components. We believe that all of this will be a good deal less difficult to achieve with the proper integration of formal specification and associated verification/auditing into the design process.

The establishment of the mechanisms and notation for these interconnect standards is now an important task. It will require methods of dealing with the protocols of communications – whether in detailed timing, virtual handshake or actual handshake – and the algorithmic or 'data' aspects. The work described in this paper gives an indication about what each of these might look like, though more integration is desirable. One thing we believe is very important, whether the interface is formally specified or not, is to make all interfaces as clean and free from unnecessary interdependencies as possible.

Standards or not, we believe that if one wants to get results from formal verification it is necessary to be flexible in the level of abstraction at which one works, and the techniques used. We have succeeded in obtaining good results in perhaps unlikely circumstances by adopting this philosophy.

The work reported in this paper is that of a number of people besides the author, all of whom have worked for him at Oxford University or Formal Systems, or been close collaborators at Inmos (in

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

several cases two of these). In alphabetical order these are Geoff Barrett, Anthony Cox, Jim Davies, Michael Goldsmith, Victoria Griffiths, David Jackson, Bryan Scattergood and David Shepherd. In addition I thank Tony Hoare and David May for the development of Occam and CSP and their general encouragement of the work, and all the members of the design teams at Inmos who have cooperated so willingly with us, and Inmos, Esprit and the Alvey programme for funding the work.

References

Barrett, G. 1987 *Formal methods applied to a floating point number system*. Oxford University Programming Research Group Technical Monograph PRG-58.

Barrett, G. 1988 The semantics and implementation of Occam, D.Phil. thesis, Oxford University.

Brown, G. M. 1991 Towards truly delay-insensitive circuit realizations of process algebras. In *Designing correct circuits* (ed. G. Jones & M. Sheeran), pp. 120–131. Springer-Verlag.

Brunvand, E. L. & Starkey, M. 1991 An integrated environment for the design and simulation of self-timed systems. In *VLSI 91* (ed. A. Halaas & P. B. Denyer), pp. 4a.2.1–4a.2.10.

Cohn, A. 1987 A proof of correctness of the Viper microprocessor: the first level, University of Cambridge Computer Laboratory Tech. Rep. no. 104.

Goldsmith, M. H. 1987 Occam transformation at Oxford. In *Programming of transputer based machines* (ed. T. Muntean). Proceedings of 7th Occam User Group Technical Meeting (14–16 September 1987, Grenoble, France). Amsterdam: IOS.

Goldsmith, M. H. & Roscoe, A. W. 1987 Transformation of Occam programs. In *The design and application of parallel digital processors*. IEE Conference Publication 298.

Hoare, C. A. R. 1985 *Communicating sequential processes*. Hemel Hempstead: Prentice-Hall International.

Hunt, W. A. 1985 *FM8501: a verified microprocessor*. University of Texas at Austin Tech. Rep. 47.

Inmos Ltd. 1988 *The transputer instruction set manual – a compiler writer's guide*. Prentice Hall.

May, D. & Shepherd, D. E. 1987 Formal verification of the IMS T800 microprocessor. In *Proceedings of Electronic Design Automation Conference* (Wembley, 13–16 July 1987).

Page, I. & Luk, W. 1991 Compiling Occam into FPGAs. In *FGPAs* (ed. W. Moore & W. Luk). Abingdon: EE & CS Books.

Roscoe, A. W. 1985 A denotational semantics for Occam. In *Seminar on concurrency* (ed. S. D. Brookes, A. W. Roscoe & G. Winskel). Springer LNCS 197.

Roscoe, A. W., Cox, A. D. B., Goldsmith, M. H. & Scattergood, J. B. 1991 Formal methods in the development of the H1 transputer. In *Transputing 91*. IOS.

Roscoe, A. W. & Hoare, C. A. R. 1988 The laws of Occam programming. *Theor. Comp. Sci.* **60**, 177–229.

Shepherd, D. E. 1988 The role of Occam in the design of the IMS T800. In *Communicating process architecture*. Prentice Hall.

Shepherd, D. E. & Wilson, G. 1989 Making chips that work. *New Scientist* 13 May 1989.

Valiant, L. G. 1990 General purpose parallel architectures. In *Handbook of theoretical computer science* (ed. J. van Leeuwen). North-Holland.