Occam in the specification and verification of microprocessors

A.W. Roscoe
Oxford University Computing Laboratory, 11 Keble Road, Oxford, and
Formal Systems (Europe) Ltd, Unit 7, The S.T.E.P. Centre,
Osney Mead, Oxford

February 17, 2004

Abstract

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 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 case studies, radically different techniques were required for them. Based on this and other evidence, 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

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 work, though the greatest part of this effort has been devoted towards the complete verification of necessarily simple processors (see [3, 7], for example). While we fully endorse the philosophy that eventually all microprocessors should be verified to the standard which that work has aspired to, it is for the moment true that the complexity of modern state-of-the-art processors such as the 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 the author has 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 employing 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 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 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 a INMOS (in 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 would like to 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 co-operated so willingly with us, and INMOS, ESPRIT and the Alvey programme for funding the work.

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 employed 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 nondeterminism 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, for example [?], 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.

The semantics of occam are rather like the product of those of CSP, 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 [10]. 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 employed 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: nondeterminism, 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 is case-study provided by the design of the Floating Point Unit (FPU) of the IMS T800 Transputer. This has been well reported elsewhere, for example [2, 9, 13] 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 stylised 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 [4, 5]. This is a tool built around the algebraic semantics of occam [?], a complete set of laws which, by their nature preserve the semantics of programs. It can be programmed with high-level strategies such as normalisation, 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 [14]. 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 and IMS T9000 range up to 30MHz and 50MHz 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 modularise 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.

Our work on the T9000 was previously reported in [11], and this paper expands a little on that and brings it more up to date.

Following the philosophy set out above, the areas where we believed formal methods would be best applied were ones where the new chip does things in different and more complex ways than previous ones – not ones where it was simply faster or had more of something. The following major areas of increased sophistication can be readily identified in the T9000.

4.1 Communications support

The introduction of more-or-less unlimited connectivity, to override the T-series 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 make the T9000 very close to an ideal component for a general-purpose parallel computer in the sense of Valliant [?].

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 T-series Transputers would, if only they had enough links.

The formal methods work on this is described in section **??**.

4.2 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.

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, 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 interdependancies are established as early as possible in the design process.

4.3 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 next stage.

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 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 co-operation of the design team within INMOS.

5.1 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 [8]) 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 nondeterminism 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 production of this simulator has been an integral part of the design process at Inmos.

Though a number of side issues have arisen, the central thrust of our effort has been to establish the congruence of these two simulators. Modulo the level of abstraction which still separates the pipeline simulator from the silicon, this achieves our objective.

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 all have the property that they are 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 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 summarised below.

5.2 High-level progress

At the highest level we have constructed an abstraction in CSP [6] 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 typical group relative to the input and output of groups and fedback stack information.

We will describe processes $Empty(S)$ (representing the stage when empty) and $Full(S_{in}, S_{out})$ (with a group in) where the parameters are

- S is 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).

$$\begin{array}{l} Empty(S) \ = \\ in?\langle S_{in}, S_{out}\rangle \to Full(S_{in} \cup S, S_{out}) \\ \Box(\boxed{}\{FbIn.X?d \to Empty(S \cup \{(X,d)\}) \mid X \in SR \setminus dom(S)\}) \\ \Box(\boxed{}\{FbOut.X!d \to Empty(S \setminus \{(X,d)\}) \mid X \in BBH \wedge (X,d) \in S\}) \end{array}$$

$$Full(S_{in}, S_{out}) = Stack \Box (Groups \lhd\!\!| dom(S) = SR |\!\!\rhd (STOP \lhd Groups))$$

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 $Stack$)

$$Groups \ = out!\langle S_{in}, S_{out}\rangle \to Empty(\emptyset)$$

$$\begin{array}{l} Stack \ = \\ (\boxed{}\{FbIn.X?d \to Full(S_{in} \cup \{(X,d)\}, S_{out}) \mid X \in SR \setminus dom(S_{in})\}) \\ \rhd(\boxed{}\{FbOut.X!\hat{d} \to Full(F, S_{in}, S_{out} \cup \{(X,d)\}) \mid X \in BBH \setminus dom(S_{out})\}) \end{array}$$

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. 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 $FbOut.X.!\hat{d}$ means that the value of the output stack value chosen is, from the point of view of this abstraction, nondeterministic. 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.

CSP tools constructed for the verification of the VCP (see below) 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.

5.3 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 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 involves 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 regime 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.

5.4 Individual groups

The final part of our factorisation 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.

5.4.1 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 reseting 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. (Although the first of these would also be an eventual corollary of that described in Section 5.4.2 below, it is much quicker and easier to write software to perform this task in isolation. It was in fact performed separately first, in order to provide information earlier.) The second task is performed by the following process.

- We analyse the equivalence classes of instructions which are treated in the same way by the Decoder. A representative is picked from each, shortest (in terms of bytes) if the class contains members with different lengths.

- The Decoder simulator is then tested to see which sequences of these representative instructions are recognised by the logic. A consequence of the notion of equivalence, which is in turn derived from analysis of the structure of the Decoder, is that any other sequence of instructions may be formed into a group if and only if it is no more than eight bytes long and the corresponding sequence of representative instructions is an allowed group.

- There are two written specifications of the Grouper: a table and a BNF syntax. Both of these are given in terms of 'resources' of which a single instruction might use one or more. These are compared using two programs. The first (by automatically converting the BNF into a network of occam processes which compute the members of the various classes in a syntax) generates all productions of a finitary BNF. The second converts tables such as the one describing the grouping into another finitary BNF. The productions of the original BNF and the one derived from the table are generated and compared. Clearly they should be the same.

- A comparison is then performed between the sets of instruction groups of length less than or equal to eight bytes fitting the specifications and those allowed by the Decoder. Here again the combinatorial explosion can be reduced by judicious use of equivalence arguments. Care needs to be exercised in doing so in order to ensure that the equivalence classes of instructions used are fine enough so that they respect both the Decoder equivalence discussed above and equivalence of resources as viewed by the specifications. Various programs have been written for this phase of the effort.

5.4.2 The actions of groups

As a group passes along the pipeline, data items are generated and modified at relatively few places. Assuming correctness of the various inter-group mechanisms discussed above, the outputs from a group (which are, amongst others, its fed-back stack, and the address and data associated with any write it performs) are all computed from relatively few possible inputs in a bounded number of simple operations. The complexity of checking the activity of individual groups comes from the interactions of the control signals, which decide both which of these actions to carry out on data items and also which of a myriad possible switching operations are carried out on the items as they pass along. Nevertheless (with a few well-understood exceptions) the actions of the pipeline element, both switching and substantive operations, have the property that they depend (deterministically) only on control signals set by the Decoder for the group, but not on the data values being handled in the group nor on other data. It is also very helpful that the functionality of all groups is such that they never read information which they have, themselves, written to memory.

Therefore a proof that the pipeline works for a particular group can consist of showing (allowing for the exceptions alluded to above) that the control signals it sets cause the pipeline to do the desired things to its 'input values' (which include any operands present in primary instructions in the group) in producing its 'output values'. We have chosen to do this by developing transformations which allow us to detect, symbolically, what the outputs of the group are as a function of the inputs. This process is described in Section 6.

5.4.3 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 to 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.

6 Sybolic Execution

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

- The introduction of constants at various places (*ldc*, *mint* and floating-point zeros).
- Local address calculation ($W{+}4n$) and loads ($@(W{+}4n)$) in the Local Cache stage.

- Address calculation (linear combinations such as $A{+}4B{+}4n$, from *wsub* and *ldnlp*, etc) in the Address stage.
- Indirection (e.g., $@A$) in the Read stage.
- 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. In summary this is assigned or input whenever the original is, but is not used to decide control flow. With a few well-understood exceptions, which can be dealt with, the only things which tell a stage what to do with a group's data are its control signals: the data for which we are computing the symbolic values thus does not generally affect the decisions of which operations are to be carried out on it or how it is to be switched around. This is very important in justifying this verification strategy.

Typically, we are only directly interested in the symbolic representations of a small proportion of the variables of a given program. However, in order to be able to calculate these we need to keep ghosts of all variables which can contribute (in some sense) to their values. Also, because of the ways in which we handle procedure arguments and communication (discussed below), it can be difficult if one variable of a given type has a ghost and another does not. For these reasons, we ghost all or none of the identifiers of a given type (with mechanisms to handle interface with the 'outside world' reasonably gracefully). Where we ghost variables of type T, we also ghost arrays of T. This will lead to more ghosted objects being generated and switched than might otherwise be the case, though the execution overheads of even this at-first-sight over-extensive ghosting have proven surprisingly light in practice. The type of the ghost variable itself should be appropriate for carrying the symbolic information. In our case it is generally a string (array of bytes). The following sets out the basics of how the ghost assignments, inputs and outputs are formed, and how procedures are modified.

- Where one ghosted identifier (or component of a ghosted array) is assigned to another, the corresponding ghost assignment is carried out. Subscripts of ghost arrays are the real, not the ghost ones. Similar considerations apply to outputs. Thus, if the arrays a and b and the scalars c and d all have ghosts ($\widehat{a}$, $\widehat{b}$, $\widehat{c}$ and $\widehat{d}$) then the assignments $a := b$, $a[i] := c$, $c := b[4]$ and $c := d$ are doubled up with the respective ghost assignments $\widehat{a} := \widehat{b}$, $\widehat{a}[i] := \widehat{c}$, $\widehat{c} := \widehat{b}[4]$ and $\widehat{c} := \widehat{d}$.

  The rationale behind this is that it allows us to track the values as they are switched around the pipeline.

- Where something other than simple switching is carried out in an assignment to a ghosted variable, we need to store in its ghost a symbolic record of the new object that has been stored in it. This would apply to the assignment of a constant, an identifier with no ghost, or an expression involving operators. Early versions of the ghosting tool did not attempt to do this automatically: any operation was turned into a labelled 'bottom' which, when it propagated to the outputs, indicated the need to edit it into a symbolic operation. However the large size and frequent versions of the pipeline simulator made it necessary to automate this procedure, so there is now the facility to introduce the symbolic operations automatically. The type of symbol with respect to which this automated procedure works are strings with functions represented in prefix (sometimes called Polish) form.

- A ghost assignment should always be placed *before* the one it is shadowing because the real assignment might affect some values (e.g., array indices) which are needed on either side of the ghost one. Since the real assignment does not depend on ghost variables, the put-ghost-first version is safe.

- The parameter lists of procedures (both at declaration and calls) are extended to include ghost versions of any parameters of a type we are ghosting.

- Any channel protocols involving such types have to extended by ghost types. Furthermore, if the types we are ghosting include INT (say), then any channel with an unnamed protocol involving the type INT, such as a CHAN OF INT, CHAN OF INT::[]BYTE or CHAN OF [56][3]INT has to have a new protocol defined for it (which carries the old information plus the ghosted information). This has to be done in a way such that the name chosen for the protocol is determined purely by the semantics of the channel.

- Inputs and outputs are extended to conform to the modified protocol. The precise places in the input/output item list where the new values are to be placed are again determined by consideration of the possible undesired changes to index variables caused by the 'real' inputs. Particularly delicate considerations surround the ghosting of a channel with a protocol including a ghosted counted-array component: the reader might like to work out the contortions necessary to allow the ghosted version of an input such as

  $$c\ ?\ a[0];\ a[a[0]]::a$$

  to work correctly when a is an array of a (scalar) type to be ghosted.

- Because functions in occam are free of side-effects, and in order not to prejudice the use of the original functions in the expressions where they were used, functions are ghosted in a rather different way to procedures. There is no need to ghost a function whose return type is non-ghosted, since by assumption no unghosted value is affected by the values of ghosts. Where its type is ghosted, we leave the old declaration alone and declare a second function which returns the ghost value. The parameters to the ghost function are both the original parameters of the function and the ghosts of any with appropriate type. At any point in the program where we compute an expression which is the value of a ghosted function (such as the assignment of a call of the function to a ghosted variable), the corresponding value of the ghost function is used in the ghosted version of the expression.

  One problem which arises from this is that the type used for ghosting may not be one which can legally (at least in current implementations) be used as the return type of a function. In other words, the result of applying the above strategy will produce non-compilable occam. We get around this difficulty by using a separate transformation which we term *defunctionalising*, which eliminates such function calls in favour of procedure calls. Note that an expression such as $f(g(x), g(y))$ involving several such functions may have to be transformed into a sequence of procedure calls with local variables being used to hold intermediate results. Function calls in abbreviations typically require the introduction of extra sequential construction, and care is also needed to avoid the (possibly error-causing) calculation of function calls which occur in the second argument of an AND or OR, when the overall result is determined by the first argument.

This complexity over functions is necessary in our application for two reasons. The first is because the automated symbolisation of functions generates cascaded string-type functions from even a simple expression such as $x + (y * 3)$, which becomes

$$sym\_plus(\widehat{x}, sym\_times(\widehat{y}, sym\_const(3)))$$

The second is that there are a large number of places in the pipeline simulator where functions are used to do switching of data. These functions are often composed in a single assignment, modelling the cascading of gates in the silicon, in a way which makes the discussion in the last paragraph highly relevant.

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.

6.1 Analysing the pipeline

It is important to remember that this technique 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 it is ghosted.

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.) 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 the group are both 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 typical groups are given below. The format is prefix as described above, and that used by the ghosted simulator, 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 letters `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.

Some sample instruction specifications:

```
add                       ldnl
A = +BA                   A = @+<<i2A
B = C                     B = B
C = u                     C = C
I = +I1                   I = +I1
W = W                     W = W

stnl                      mint
A = C                     A = (#80000000)
B = u                     B = A
C = u                     C = B
I = +I1                   I = +I2
W = W                     W = W
WriteAddr = +<<i2A
WriteData = B
```

Two sample results of groups

```
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
```

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 nondeterminism 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.

7 Communications support and the VCP

We have already described how the combination of the VCP and the IMS C104 crossbar provide enhanced communications support in the T9000 family, and how the proof obligation essentially comes down to showing that the effect of using these is essentially the same as using transputers with a separate physical link for each logical channel.

Some of the proof obligations (criteria for ensuring deadlock-freedom in the wormhole routing and interval labelling schemes, for example) are quite straightforward. Others, such as the interactions between the VCP and the scheduler and CPU pipeline components of the T9000, require more subtle analysis. Some work has been done here, but by far the largest piece of work that has been, and is being, carried out on the VCP is state-machine analysis of the multiplexing and related mechanisms.

This is a natural continuation of previous formal work on the transputer model of communication concurrency, namely [1]. It has been done by Geoff Barrett and Victoria Griffiths at INMOS, with the help of tools designed and built by Barrett, David Jackson and the author. The notation used has been a version of CSP rather than occam, since at the level of abstraction of this work the state information is irrelevant or ignored.

There are essentially three levels to this work: a CSP description close to the actual implementation, a CSP description intended to represent the intended behaviour of the VCP, and abstract properties such as deadlock-freedom and simple safety specifications. The first two of these can be encoded as the parallel compositions of a number of simple state-machines, with internal communications hidden. The required condition then is that the implementation VCP should refine the specification one. The abstract properties, such as deadlock-freedom and showing that a disabled channel does not communicate, can also be brought within this refinement setting by encoding a typical specification as the most nondeterministic process satisfying it. Typically this can be thought if as a state machine with very few states. We can then prove that the specification VCP meets them by proving refinement – on the assumption that the implementation VCP refines the specification one, we can use transitivity to deduce them of the implementation as well.

This being a real system with a complex specification, any failure of refinement is as likely to be in the specification level as in the implementation one. This gives additional reasons for wanting to prove the more abstract properties of the specification, and also makes it very desirable to have good diagnostics in tools used to verify refinements.

The parallel/hiding composition of a finite number of finite-state CSP processes is still finite-state, though the number of states typically rises exponentially with the number of processes. It is thus in principle possible to check all the refinements required automatically. The number of channels dealt with by the VCP gives rise to state-spaces well beyond reach, so in practice the work has been done with a number (typically three or four) which avoids this problem but which is believed (for formal or heuristic reasons) to be sufficient to display any undesirable behaviour.

The form of refinement used is the usual one for CSP: failures/divergence, though with an element of special treatment of divergence. Our tools turn a CSP description into one of two types of state machine depending on whether it is at the specification or implementation end of the proposed refinement. The specification end is *normalised*: the state-machine has nodes which are labelled with the set of maximal refusals allowed, and a unique arc for each action (all visible actions) which the node can perform. Given a designated initial node, there is only one possible node possible after any given trace (sequence of visible actions) in the system. The normal form is also compacted to ensure that the machine has no indistinguishable nodes.

The implementation end is left in an easier-to-compute form, with unstructured nodes allowing both invisible (τ) actions and multiple arcs with the same label. The construction of the normalised form is easiest to visualise in terms of using this simpler form as an intermediary – this is sometimes the method used, though on other occasions other techniques are more efficient.

The normalisation procedure is comparatively expensive, indeed potentially exponential in the size of the un-normalised state machine since the basic computation is based on sets of these states (the sets of states in the un-normalised machine which it is possible to reach on some trace). It has been practical in our cases because the actual growth is not nearly as bad as this suggests, and because the specification end of the refinement almost invariably has many fewer states than the implementation end. The abstract specifications at the highest level almost invariably have trivial numbers (single figures), and we have in practice had little difficulty in normalising descriptions which give un-normalised state-machines with a few thousand states.

The checking of refinement then proceeds by examining the cartesian product of the specification and implementation state-machines. It is necessary to determine which pairs of states can be reached in the two machines on the same state. For each such matching it is required that the set of initial actions in the implementation should be a subset of those for the specification, and that if the implementation can refuse a set of actions (by being stable, for example) then this is a subset of one of the specification's maximal refusals.

This exploration then proceeds straightforwardly and linearly in the number of combinations found (in practice this rarely exceeds the number of states in the implementation by a significant factor).

What we are doing here is, of course, a form of model-checking [**?**, **?**]. The normalised specification is playing the same role in our system as the specification state-machines used there.

Like all model-checking techniques which explore a large finite state space, our algorithms are essentially bound by the size of the machine's physical memory, since the memory access patterns are typically very random and recourse to virtual memory has a crippling effect on performance. (A typical factor here is 100.) The technology we have employed for conserving this resource has not yet had to be particularly sophisticated, though this will doubtless be the subject of further work. Typically we can now deal with of the order of 100,000 states on a largish workstation, and would expect to be able to increase this to 1,000,000 without too much difficulty.

An understandable and reasonably small example, with obvious similarities to the VCP, is provided by buffer multiplexing in CSP. A one place buffer in CSP can be written

$$COPY = in?x \rightarrow out!x \rightarrow COPY$$

which, forgetting about the details of data values, becomes

$$B = in \rightarrow out \rightarrow COPY$$

N of these operating in parallel and independently can be written

$$Spec_N = (1.B \| 2.B \| \ldots \| N.B)$$

If these are all to operate between a pair of processors but we do not have the resources to allocate each of them a separate channel, we can multiplex them using a pair of channels, one each way. One carries the data (were there any) and an indication of which channel it belongs to. The other takes back acknowledgements. The complete multiplexed network consists of a pair of processes for each buffer and four fan in/fan out processes for messages and acknowledgements. We define these processes as follows:

$$\begin{array}{rcll} S_i & = & i.in \rightarrow a.i \rightarrow d.i \rightarrow S_i & (i \in \{1, \ldots, N\}) \\ R_i & = & b.i \rightarrow i.out \rightarrow c.i \rightarrow R_i & (i \in \{1, \ldots, N\}) \\ TM & = & a?i \rightarrow data.i \rightarrow TM & \\ RM & = & data?i \rightarrow b.i \rightarrow RM & \\ TA & = & c?i \rightarrow ack.i \rightarrow TA & \\ RA & = & ack?i \rightarrow d.i \rightarrow RA & \end{array}$$

The complete network (structured to see the two ends of the system and with the hiding shown at the most logical places, though it can all be moved to the outermost level by the laws of CSP) is then

$$((S_1 \| \ldots \| S_N \| TM \| RA) \setminus \{a, d\} \| (RM \| TA \| R_1 \| \ldots R_N) \setminus \{c, d\}) \setminus \{data, ack\}$$

The state space of this 'implementation' system Imp_N is clearly much larger than the 'specification' one given earlier. What we would typically prove is that (for some fixed N) the implementation refines the specification (i.e., $Spec_N \sqsubseteq Imp_N$). In fact, in this case, they are equivalent, but this would not come out without either running the system in reverse (which is more expensive for reasons discussed earlier) or examining the specification to demonstrate that it is deterministic.

8 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

- 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
- 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 dealt with the great complexity of the problems tackled, particularly with respect to the T9000 pipeline.

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 standards 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 in 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.

References

[1] Barrett, G., *The Semantics and Implementation of occam*, Phd Thesis, Oxford Universtity, 1988.

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

[3] Cohn, A., A proof of correctness of the Viper microprocessor: the first level, University of Cambridge Computer Laboratory Technical Report No. 104, 1987.

[4] Goldsmith, M.H., "occam Transformation at Oxford", in Muntean *ed.*, *Programming of Transputer Based Machines*, Proceedings of 7^{th} occam User Group Technical Meeting, (14–16 September 1987, Grenoble, France), IOS B.V., Amsterdam, 1987.

[5] Goldsmith, M.H. & A.W. Roscoe, "Transformation of occam Programs" in *The Design and Application of Parallel Digital Processors*, IEE Conference Publication 298, 1988

[6] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice/Hall International, Hemel Hempstead, 1985.

[7] Hunt, W.A., *FM8501: A verified microprocessor*, Unversity of Texas at Austin Technical Report 47, 1985.

[8] INMOS, *The transputer instruction set manual — a compiler writer's guide*, Prentice Hall, 1988.

[9] May, D., & D.E. Shepherd, "Formal verification of the IMS T800 microprocessor", in Proceedings of Electronic Design Automation Conference (Wembley, 13–16 July 1987), 1987.

[10] Roscoe, A.W., "A denotational semantics for occam", in Proceedings of the Pittsburgh seminar on concurrency, LNCS 197, 1985.

[11] Roscoe, A.W,, A.D.B. Cox, M.H. Goldsmith & J.B. Scattergood, "Formal Methods in the development of the H1 transputer", in Proceedings of Transputing 91, IOS 1991.

[12] Roscoe, A.W., & C.A.R. Hoare, "The laws of occam programming", Theoretical Computer Science ????

[13] Shepherd, D.E., "The role of occam in the design of the IMS T800" in *Communicating Process Architecture*, Prentice Hall, 1988.

[14] Shepherd, D.E. & G. Wilson, "Making chips that work" in *New Scientist*, 13 May 1989.