Compiling occam into Field-Programmable Gate Arrays

Ian Page and Wayne Luk

Programming Research Group, Oxford University Computing Laboratory,
11 Keble Road, Oxford England OX1 3QD

Abstract

We describe a compiler which maps programs expressed in a subset of occam into netlist descriptions of parallel hardware. Using Field-Programmable Gate Arrays to implement such netlists, problem-specific hardware can be generated entirely by a software process. Inner loops of time-consuming programs can be implemented as hardware and the less intensively-used parts of the program can be mapped into machine code by a conventional compiler. Software investment is protected since the same program can run entirely in software, entirely in hardware, or in a mixture of both. A single program can thus result in many implementations across a potentially wide cost-performance range. The compilation system has been used to generate inner-loops, hardware interfaces to real-world devices, systolic arrays, and complete microprocessors. In the near future we hope to have a proven version of the compiler, enabling us automatically to generate provably correct hardware implementations, including microprocessors, from higher-level specifications.

INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are readily available, multi-sourced components which have a proven place in development, small-scale production, and even large-scale production runs. One style of field-programmable gate array uses static RAM to dynamically configure both the function of the logic cells and their interconnections. Current examples of large-scale FPGAs in this style are the *Xilinx* 3000 and 4000 series (Xilinx, 1991 and 1990), the *Electrically Reconfigurable Array* (Plessey 1990) and the *Logic Cell Array* (Algotronix 1990) devices. Vendor-provided CAD software creates device configuration data from user-provided circuit schematics, Boolean equations, state-machine descriptions, or other higher level descriptions.

Recently, dynamically reconfigurable FPGAs with equivalent complexity of 10,000 to 20,000 gates have become available, and denser devices are being developed. Due to the relatively large scale of these devices and their programmability, it is now possible to compile non-trivial algorithms directly into reusable hardware. By combining FPGAs with a traditional microprocessor, it is possible to build systems that are compiled from a high-level notation

directly into a mixture of hardware and software in order to exploit the best features of each domain effectively. The user's software investment can be protected since the same program can run entirely in software, entirely in hardware (subject to available resources), or in a mixture of both. A single program might result in many implementations across a potentially wide cost-performance range.

Many application programs demand more processing power than can be supplied by conventional processing resources. Many of these programs have a kernel which is responsible for a significant fraction of the total execution time of the program. We have a particular interest in vision algorithms where these conditions certainly hold and we are increasingly looking towards application-specific hardware to provide the necessary computational resource for real-time robot vision. Other application areas exhibiting similar characteristics abound. While a complete application program is probably too complex to be implemented as special-purpose hardware, it will often be feasible to implement the kernel of the application in hardware with the remainder of the program being implemented in machine code for one or more conventional computers. Examples of possible uses are legion, with computer graphics, image processing and vision, data compression, database search and maintenance, and cryptography being some obvious application areas. Another area of interest to us is interfacing to real-world devices which need intelligent control with real-time requirements beyond the capability of conventional processors.

The work reported here describes a compilation system which takes specifications of algorithms into specifications of synchronous hardware in netlist form. Related work on delay-insensitive implementations of CSP/occam, can be found in van Berkel and Saeijs (1988), and Brown (1991). Our netlists contain only simple gates and D-type flip-flops. The netlist is processed with Xilinx software tools (XNF2LCA, APR, and XACT) to produce configuration data for Xilinx FPGA chips. The compiler maintains a refinement relation between the semantics of the algorithm specification and of the hardware such that the observable behaviour of the algorithm is identical whether it is implemented in hardware or by conventional software.

The occam language (Jones 1987) is an ideal candidate for the source language for such a compiler due to its simplicity, static compilation properties, minimal run-time demands, and its expressive power for parallelism. Above all, its well-defined semantics, being based in Hoare's CSP (1985), make it highly appropriate for checking the equivalence between widely varying implementations and amenable to formal proof techniques. We are making extensive use of these properties in our attempts to prove correct a version of the compiler described here.

Due to the array architecture of these chips, they are especially appropriate for systolic implementations of algorithms. We design such systolic arrays using a declarative language with a set of correctness-preserving transformations. A companion paper (Luk and Page 1991) describes our approach to the specification, transformation and implementation of systolic algorithms.

A prototype hardware host has been built which has a Xilinx XC3090-100 FPGA coupled with two banks of 32k x 8 static RAM, and a Transputer Link Adapter which allows the system to be coupled to a transputer array. We are currently designing the successor to this prototype which has a T800 transputer, a large bank of dynamic RAM, a large Xilinx chip, a frequency synthesizer, and two banks of static RAM on a daughter board designed to the TRAM standard (Inmos 1991). We hope to replicate this module on standard TRAM host boards in order to exploit coarse-grained algorithmic parallelism, as well as fine-grained parallelism within the Xilinx chip (and of course within the transputer itself).

ALGORITHM DESCRIPTION LANGUAGE

For the reasons given above, a simple subset of occam1, with a few minor extensions appropriate to its current use, was chosen for the input language. As the current compiler is a prototype written in SML (Harper 1986), it has proved convenient to supply the input programs in the form of an abstract syntax. This has obviated the need to build a parser for the language, but at the cost of writing programs in a slightly verbose form. A pretty-printer for the abstract syntax produces listings in conventional occam form. As part of the process of demonstrating a proof of correctness, we have produced a small version of this compiler in Prolog so that the implementation and proof are very close to each other.

The input language of the compiler currently includes integers and channels as the basic data types, expression evaluation, multiple assignment and communication as the basic processes, and SEQ, PAR, IF, WHILE, ALT as the constructors. It has been found useful, though not necessary, to make a few minor extensions to the occam language. The modifications allow variable precision integers and channels, and offer a slightly richer set of logical operators in the expression sub-language. We take it as evidence of the descriptive power of occam that so little has been modified, despite the fact that we are generating hardware rather than machine code. In time, even these extensions could be withdrawn as more powerful optimisation capabilities are built into the compiler.

COMPILATION STRATEGY : GENERAL SCHEME

The compilation is top-down, syntax-driven, and consequently is very fast. The declarations are first usage-checked against the program body and used to generate netlist entries for the registers corresponding to user variables. Channels generate no static hardware; the circuitry for a particular channel is generated from the input and output statements which quote it. A minor amount of hardware is generated to provide the global START signal for the hardware and to deal with the global STOP signal. Compilation then proceeds by recursive descent over the body of the program, adding hardware to the netlist according to the circuits given in the next section. A final phase optimises the netlist and adds I/O buffers before outputting .XNF files, which are currently our interface between the compiler and vendor software for programming the FPGAs.

In the following sub-sections we review the general characteristics of the control circuits generated by the compiler and the strategy for dealing with expressions.

Clocking Scheme

For reasons of simplicity (of the compiler and proof) and compact designs, the compiler generates completely synchronous circuits. There is a single global clock for the target hardware; a clock period is terminated by the rising edge of the global clock, at which time *every* flip-flop in the system is triggered. This model is well supported by the Xilinx chips which have dedicated low-skew clock lines. However, much of the compiler will remain unchanged when we eventually re-engineer it to produce self-timed circuits.

The synchronous nature of the hardware, together with the fact that every opportunity for parallel implementation is taken, means that precise guarantees can be given for the real-time response of the resulting implementations. There is a simple calculus of dura-

tions of these programs, which is particularly simple in the case of WHILE-free programs. The essence of this calculus is that SKIP, assignment and communication (where each process is ready) each take precisely one clock cycle, and all other constructs take precisely zero time. At the moment, we rely on vendor software to predict maximum delays in our circuits after placement and routing and use this information to set the clock speed, or alternatively, use it as a guide in transforming the program into a faster version using only the laws of occam. In the future, we hope to have the compiler do more of this work itself.

An unexpected side-effect of this synchronous implementation technique, is that there are a number of additional laws for these implementations which are not valid for all implementations of occam. It may be possible to make use of these in deriving more optimal implementations of programs using formal transformation tools such as the Oxford Occam Transformation System (Goldsmith 1987).

Control Strategy

As well as generating all necessary datapaths, the compiler generates fully distributed control hardware for the program. Each statement in the program results in the generation of hardware which controls the activation of the corresponding parts of the datapath (i.e. the *execution* of the statement). There is usually one flip-flop corresponding to each primitive statement in the user program which is set only when the statement is being executed. This is basically the 'one-hot' approach to control state encoding and has been demonstrated to be space-efficient in particular on Xilinx FPGAs (Knapp 1990 and Schlag *et al* 1990). It is perfectly possible to adopt alternative encoding schemes which may be preferable in other circumstances. We have developed provably equivalent control state encoding schemes which will later be integrated into the compiler as optional optimisations.

The control hardware for each statement compiled has a single control input signal (called START) which triggers execution of the statement and a single control output (called FINISH) which signals that statement execution is complete. For a single execution of any statement, these control signals are each high for exactly one clock period (terminated by the rising edge of the global clock).

The external environment must guarantee that the initial control pulse to start program execution is just one clock period long and must not attempt to initiate further executions of the program until the corresponding FINISH signal has been generated. Given this initial *environment guarantee* to the program, the control circuits generated by the compiler maintain this guarantee for every (nested) statement in the program.

Expressions

The language has only integers as a basic type, so the basic values are integer constants and values of variables. The sub-language of expressions has a selection of binary and unary operators over variable-width integers, currently including addition, subtraction, multiplication, magnitude extraction, bitwise And/Or/Xor/Not operations, shifting, field extraction and concatenation. New operators are usually prototyped in the declara-

tive language mentioned earlier and then interfaced to the compiler. We are working on a comprehensive integration of the two approaches in which a higher-order declarative sub-language replaces the current expression language in an occam-like framework.

At present, the compiler generates fully parallel hardware for expression evaluation, with some hardware unit typically being generated for every operator in an expression. Figure 1 shows a simple expression involving three operands (derived from registers, say) from which a result is calculated by two hardware units. All expressions are represented as trees of this form and expression evaluation hardware mirrors this structure. Only untimed, combinational logic is present in expression hardware.

Since expressions are evaluated constantly, the environment must guarantee a sufficient period of stability of the input operands before the result of an expression circuit is used. We make the simplifying assumption that all expressions are evaluated within a single clock period. There are hooks present in the compiler which make it straightforward to have individual expressions evaluate in any integral number of clock periods, but currently the compiler does not have code to predict evaluation timings, so this feature is dormant. In any case, this could only be done partially in advance of placement and routing.

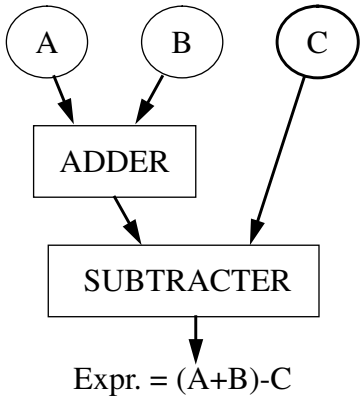

Figure 1 Expression Hardware

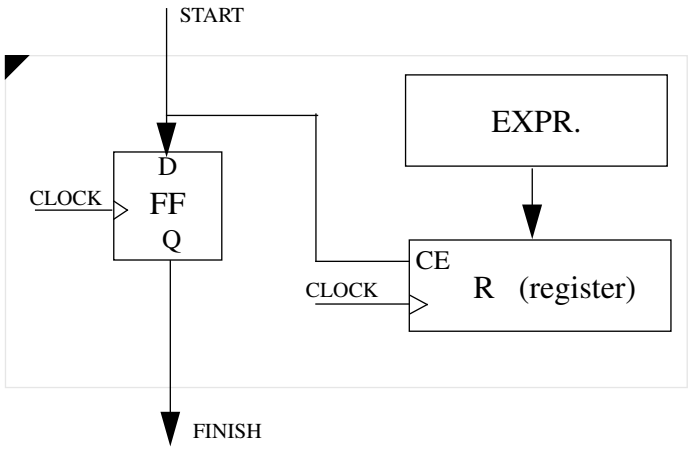

Figure 2 Assignment Control Hardware

Much of the combinational hardware associated with expression evaluation (in particular) is removed by a peephole optimisation step before producing the final netlist. These optimisations consist of removing gates with no connections to their output terminals, removing gates with constant output, and removing constant, or duplicated, inputs to gates. As an example, an expression which adds 1 to a variable will generate a chain of full-adders, which will then be reduced by the optimisation rules to a chain of half-adders. There are many other opportunities for optimisation of these circuits and some of these are the subject of further work.

COMPILATION OF STATEMENTS

In this section we show how the various statements of the input language are mapped into control hardware. Further details of the datapaths generated are not given, as they are straightforward and are essentially covered in the notes given above and in the following sub-section.

Assignment Statements

The hardware generated by the statement R := EXPR is shown in Figure 2. The black triangle in the top left hand corner of the box denotes an implementation of the control circuitry for a statement so that it can be distinguished readily from other hardware units.

Since expressions evaluate within one clock period, the control circuitry should cause the destination register to load the expression value at the end of the clock period in which the START pulse is present. It must also provide a control output pulse co-incident with the following clock period. All destination registers have a synchronous clock and a clock enable input. This means that the incoming control pulse itself can be used to drive the REGISTER_LOAD (i.e. clock enable) input of the destination register since it will be high on the rising edge at the end of the START clock period. Termination of the statement is signalled to any subsequent statements in the enclosing program by a FINISH pulse, which is a delayed version of the input pulse generated by a D-type flip-flop.

In most programs, there will be multiple assignments which target the same destination register. In this case, multiplexors are generated so that each REGISTER_LOAD signal associated with a destination register enables just one of the source expressions onto the register input bus. Simple sum-of-products multiplexors are used here. The language semantics guarantee that any register can be the target of only one assignment (or communication) at any one time, so no checks to deal with this kind of conflict are needed in hardware. All REGISTER_LOAD signals for a particular register are ORed together, and the resulting signal used as the clock enable signal for the register.

SKIP and STOP Statements

The implementation of SKIP is a D-type flip-flop providing a delay of one clock period, rather like an assignment statement without a destination register. This implementation was chosen so that arbitrary delays (in multiples of the clock period) can easily be programmed. It can be optimised to a piece of wire when the delay is known not to be required; e.g. when explicit temporal constraints on execution have already been met.

The implementation of STOP receives a START pulse, and never generates a FINISH pulse. Instead, it sets a global flip-flop to indicate the stop condition

SEQ Statements

The simplest composite statement is SEQ, the implementation of which is shown in Figure 3. The START pulse triggers the first statement in the sequence. On its termination, the first statement will trigger the second statement, and so on to the termination of the

whole SEQ construct, when the FINISH pulse is generated. If the environment of the complete SEQ construct respects the rules for control pulses, then clearly the construct provides a well-behaved environment for each statement in the sequence. This can be formally proven, as can similar assertions for the other statements.

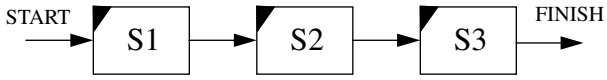

Figure 3 SEQ (S1, S2, S3)

IF and WHILE Statements

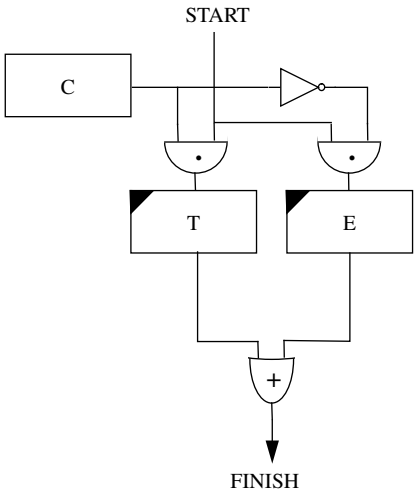

Figure 4 IF C THEN T ELSE E

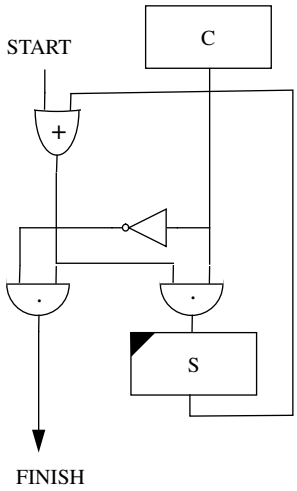

Figure 5 WHILE C DO S

Conditional statements are provided by an IF-THEN-ELSE construct, whose hardware structure is shown in Figure 4. The incoming START pulse is steered to one of the two controlled statements, depending on the evaluated Boolean expression, C. More complex conditional statements, such as an occam-style IF are built from compositions of the basic construct.

The implementation of the WHILE construct is shown in Figure 5. where the control pulse (either recirculated, or the initial one) is directed either back to the controlled statement or to the FINISH of the construct depending on the current value of the Boolean expression.

A CASE statement has also been implemented but is not shown here. At the hardware level, the evaluated expression (in $\log_n$ bits) drives a 1-of-n decoder with each output triggering one of the dependent statements in the CASE expression. The hardware optimisation phase removes redundant hardware if the set of CASE constants is not compact.

PAR Statements

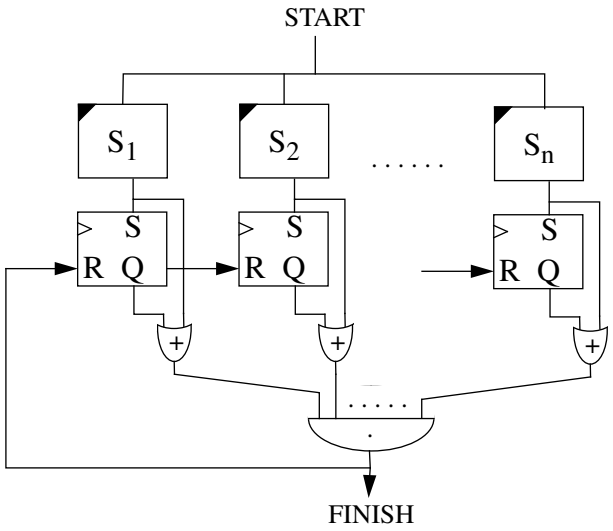

Figure 6 Implementation of PAR

The implementation of the PAR construct is shown in Figure 6. The incoming control pulse activates all statements in parallel and each statement sets a (reset dominant) SR flip-flop when it has completed. When all statements have completed, the outgoing control pulse is generated and the flip-flops are all reset. The OR gates are an optimisation so that the last statement to complete can trigger the output directly without waiting for a further clock period. There is thus no time overhead for initiating or terminating a parallel construct.

Implementation of Channels

The implementation of control hardware for channel communication is shown in Figure 7. An arbitrary number of statements may input to, or output from, a channel; all such statements are implemented with the circuit shown. The REG_LOAD signal is only relevant for channel input statements where it is used to enable loading of the destination reg-

ister, exactly as for assignment (indeed, communication in occam can be regarded simply as distributed assignment).

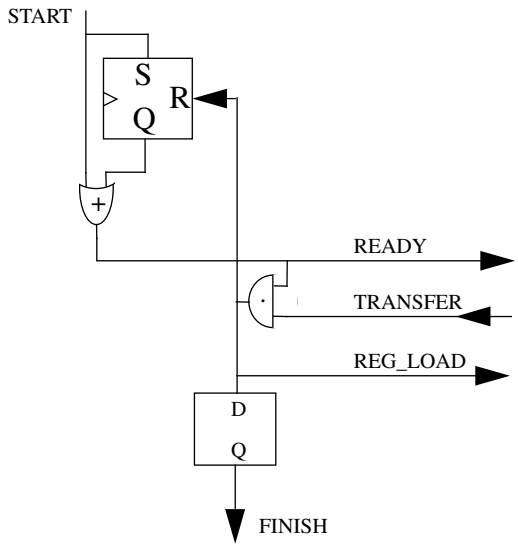

Figure 7 Implementation of Channels

There is one pair of START, FINISH signals for each input or output statement which quotes a particular channel, as shown in Figure 7. The START signal sets a (reset dominant) SR flip-flop to remember that communication is pending for a particular statement. An OR gate performs the same optimisation as was seen on the flip-flops in the implementation of PAR.

The READY signals for all the input statements involving a particular channel are ORed together, giving a signal which is asserted whenever *any* statement is ready to input from the channel. A similar OR circuit determines when any statement is ready to output to the channel. These two signals are ANDed to form the TRANSFER signal shown in the figure. This TRANSFER signal is asserted when precisely one input and one output statement are ready to communicate over the channel. The language semantics ensure that there can only ever be at most one input and one output statement trying to communicate over any channel at any one time, so no checks are necessary in the hardware to deal with this type of conflict. The TRANSFER signal also resets the request flip-flops that initiated the transfer, and the D-type flip-flop creates a one clock delay exactly as for assignment. The channel datapath circuitry between sending and receiving statements is generated by the mechanism already set up for assignment.

The compiler also implements a fixed-priority ALT which accepts input from the first channel to offer it. Lack of space precludes the presentation of its details however.

APPLICATION EXAMPLES

We briefly present two widely differing implementations of a simple algorithm. One is a direct implementation of an algorithm to generate Fibonacci numbers, the other is a (very simple) 'application specific' microprocessor running code which achieves the same

end. Although not shown here, we can formally prove the equivalence of such implementations.

```
INT_7 R1, R2 :
SEQ
    R1, R2 := 0, 1
    WHILE TRUE
        R1, R2 := R2, (R1 + R2)
```

The direct Fibonacci algorithm above is straightforward, and the compiler produces a circuit with 18 D-type flip-flops and 52 primitive gates and inverters from it. The circuit produces a new value (modulo 2^7) every clock cycle.

As a simple example of a custom microprocessor for an algorithm, the program below is an interpreter for a processor with 7-bit words and a repertoire of 8 instructions. The instruction format has a 3-bit opcode at the most significant end, and a 4-bit operand. Some liberties have been taken with occam syntax to reduce the size of the program on the page.

```
INT_4           IPTR :
INT_7           INST, AREG :
[16] INT_7      MEM :
WHILE (INST DROP 4) <> STOP
    SEQ
        INST, IPTR := MEM [IPTR], IPTR + 1          -- Fetch+Increment
        CASE (INST DROP 4)                          -- CASE (opcode)
            0 :  SKIP                               -- SKIP
            1 :  AREG := INST TAKE 4                -- LDC
            2 :  AREG := MEM [INST TAKE 4]          -- LDA
            3 :  MEM [INST TAKE 4] := AREG          -- STA
            4 :  AREG := AREG + MEM [INST TAKE 4]   -- ADDA
            5 :  IPTR := INST TAKE 4                -- JMP
            6 :  IF                                 -- JLT
                     AREG < 0
                         IPTR := INST TAKE 4
                     TRUE
                         SKIP
            7 :  SKIP                               -- STOP
```

The TAKE and DROP operators either take or drop *n* least significant bits from a value. They are simply shorthand equivalents of certain shift and mask operations, and are used here for extracting the operand and opcode.

The compiler implements this microprocessor using 143 D-type flip-flops and 313 primitive gates. These figures ignore the circuitry to bootstrap machine code into the on-chip memory, MEM, but include the on-chip memory itself. The processor is not pipelined (though it easily could have been) and so takes two clock cycles for each instruction fetched and executed; one for the instruction fetch and instruction pointer increment, the other for the CASE statement (as it happens every variant takes exactly one cycle).

The listing below shows a Fibonacci program that can be loaded into the microprocessor. It generates a new result every 8 instructions. Clearly, this program and the microprocessor are illustrative, and not meant to be in any sense optimal. The point of showing the microprocessor implementation is to highlight that (i) algorithms can be compiled either directly into hardware or into an interpreted form, or a mixture of both, (ii) the two forms can be made provably equivalent, (iii) the compiler itself might be the best judge of which implementation style to choose for (parts of) an algorithm, and (iv) eventu-

ally, the compiler might be smart enough to design optimal architectures for the processor(s) from the details of the particular application program itself.

Address	Contents		
0	LDC	0	
1	STA	R1	Initialise R1 (previous Fib.)
2	LDC	1	
3	STA	R2	Latest Fib. in Accumulator at this point
4	ADDA	R1	Calculate next Fib.
5	STA	X	Re-arrange register contents
6	LDA	R2	
7	STA	R1	
8	LDA	X	
9	JMP	3	Loop
13	VAR	R1	
14	VAR	R2	
15	VAR	X	

In order to prove the equivalence of the two programs, the bootstrap code (consisting of explicit assignments of the integer constants corresponding to the machine code instructions, into the memory locations indicated) would be prepended to the microprocessor definition above. Application of the laws of occam are then sufficient to demonstrate that the second program refines the first.

PROOF OF CORRECTNESS

We are currently working on a proof of correctness of a compiler based on the principles outlined in this paper (Page and He, 1991). Using only the algebraic semantics of occam, we show equivalences between user programs and a stylized normal form program, also written in occam. It is further shown that a variety of interpretations of these normal form programs is possible. One of the interpretations leads to an isomorphism between user programs and the style of hardware circuits described in this paper.

CONCLUSIONS

This project addresses a number of important issues in the development of computing systems. It addresses the problem faced by designers with applications which demand more, and in particular more specialized, processing than can be offered by conventional processor systems (even parallel ones). Algorithms which require non-standard operations (such as bit-reversal, field extraction/parsing, sorting/searching, complex matching) may fit into an FPGA and offer considerable speed-up when compared with machine code implementations.

It addresses the problem of what to do (usefully!) with the vast number of transistors that are becoming available, very cheaply, on silicon. Programmable gate array technology can be used side-by side with conventional microprocessors to produce uncommitted co-processors as outlined in this paper. A more exciting possibility is to use some of the microprocessor's real estate for programmable gate array(s). The microprocessor would thus gain from (i) a faster interface to the gate array(s), (ii) extra flexibility in application,

and (iii) an extended life of the microprocessor design itself since some upgrades could be handled as software updates in user systems.

It offers a degree of late-design and post-delivery flexibility for the reconfiguration of systems as new user demands arise or as new algorithms become available. In particular, interfaces to complex devices can rapidly be implemented, and since other people's devices are notoriously ill-specified there is a large degree of flexibility for coping with change in those sub-systems over which the designer has no control.

It addresses the need of system developers to prototype systems extremely rapidly. If run-time reconfiguration is not a requirement and large volumes are anticipated, an identical circuit, or one derived from it if the design method could not adequately capture the designer's implementation intentions, can be produced in mask programmable gate array form.

Finally, and perhaps most importantly, a provable correct hardware compiler offers a clear way forward for the production of complete systems, including both hardware and software, which are themselves provably correct. Where applicable, this approach has significant advantages over attempts to prove correct *ad hoc* hardware designs.

ACKNOWLEDGEMENTS

We gratefully acknowledge the help and assistance of Jifeng He who has contributed enormously to the efforts to prove correct a version of this compiler, to Prof. C.A.R. Hoare who has been tireless in his support of this new research initiative, to Adrian Lawrence who has undertaken all the detailed design work of the TRAM module, to Bernard Sufrin whose help with ML has been invaluable, and to Jonathan Bowen who has undertaken the Prolog specification work. These valued colleagues are all members of the Programming Research Group, except for Adrian Lawrence who is a member of the Computing Service at Oxford.

REFERENCES

Algotronix Ltd., *CAL1024 Datasheet*, Edinburgh EH9 3JL, UK, 1990.

van Berkel, C. H. and Saeijs R.W.J.J., ''Compilation of communicating processes into delay-insensitive circuits'', in *Proc. ICCD'88*, Rye Brook, New York, October 1988.

Bertin, P. et. al., "Introduction to programmable active memories", in *Systolic Array Processors*, J. McCanny et. al., Eds., Prentice-Hall International, 1989, pp. 301-309.

Brown, G. M., ''Towards truly delay-insensitive circuit realisation of process algebras'', in *Designing Correct Circuits*, G. Jones and M. Sheeran, Eds., Springer-Verlag, 1991, pp. 120--131.

Goldsmith, M.H., "Occam transformation at Oxford", in *Proc. 7th Occam User Group Technical Meeting,* Muntean, Ed., IOS B.V., 1987.

Harper, P. et. al., *Standard ML*, Report ECS-LFCS-86-2, Laboratory for Foundation of Computer Science, University of Edinburgh, 1986.

Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

Inmos Ltd., *The Transputer and IQ Systems Databook*, 1991

Jones, G., *Programming in occam*, Prentice-Hall International, 1987.

Knapp, S., “Accelerate FPGA Macros with One Hot Approach”, *Electronic Design*, Sept 13, 1990.

Leeser, M. et. al., “The BEDROC high level synthesis system”, in *ASIC’91*, IEEE, September 1991, to appear.

Luk, W. and Page, I., “Parametrising designs for Field-Programmable Gate Arrays”, *this volume*.

May, D., “Compiling occam into silicon”, in *Developments in Concurrency and Communication*, C. A. R. Hoare, Ed., Addison-Wesley, 1990, pp. 87-129.

Page, I. and He J. “Provably Correct Hardware Compilation”, PRG internal report, 1991.

Plessey Semiconductors Ltd., *ERA60100 Electrically Reconfigurable Array Data Sheet*, Swindon SN2 2QW, UK., 1990

Roscoe, A. W. and Hoare, C. A. R., “The laws of occam programming”, *Theoretical Computer Science*, vol. 60, pp. 177-229, 1988.

Schlag D.F., Chan P.K., Kong J., An Empirical Study of the Performance of Multilevel Logic Minimization Tools for a Field-Programmable Gate Array Technology, University of California at Santa Cruz, Computer Engineering Departmental *Report UCSC-CRL-90-60*, 1990

Shepherd, D. and Wilson, G., “Making chips that work”, in *New Scientist*, 13May1989,-pp.61-64.

Xilinx, “Technical data book”, *XC4000 series Logic Cell Array Family*, Xilinx Inc., San Jose, Ca. 95124, 1990

Xilinx Inc., *The Programmable Gate Array Data Book*, 1991.

Xilinx Inc., *XACT Reference Manual*, San Jose, Ca. 95124, 1990.