Automatic Design and Construction of Hardware Systems. (for New Electronics, 6 Dec. '94) Ian Page. October 1994.

1 Introduction

It has always been possible, and sometimes necessary, to create special-purpose hardware solutions to some computing problems. The prime motivation for doing this is when low-cost, general-purpose solutions, typically using microprocessors, do not have the performance required by the application. However the development of such hardware solutions is not for the faint-hearted. Development costs are usually high and timescales can be very long, making this an unattractive option where the need for high performance does not make it unavoidable.

However, the emergence of Field Programmable Gate Array (FPGA) chips is rapidly changing the nature of hardware implementations, since hardware can be constructed in milliseconds by loading parametrisation data into FPGAs. If this hardware innovation is coupled with high-level tools for designing the hardware configurations, then it is possible to reduce the entire hardware design to a purely software process. Additionally, where it is possible to standardise on a platform using a fixed number of FPGA and other VLSI components, it is possible to render the entire hardware design *and* implementation to a purely software process.

Over the last four years my Hardware Compilation Research Group in the Computing Laboratory at Oxford University has been working to that end, and we have developed a range of software tools and hardware platforms which allows us to create new hardware-based solutions, sometimes in only a few hours.

2 The Benefits Of Being More Precise About Design

The first step in our method is to produce an 'executable specification' for the system that we want. Such a specification looks just like a program, so you might well wonder what is the difference between a program and an executable specification. Confusingly, the answer to this question is simultaneously "not much at all", and "almost everything", depending on your point of view. The executable specification looks exactly like a program and can be compiled and executed in just the same way, so it really is a program. However, the crucial difference between an executable specification and the sort of program we all encounter every day is the language it is written in.

If you have a program written in the C language, say, there is no document in existence anywhere in the world which tells you what your program actually means; the only way of finding that out in general is to compile the program, run it, and see what happens. Unfortunately, what happens with one compiler need not be the same as what happens with a different compiler. The different compiler writers may have chosen, say, to represent characters in either 8 or 16 bits with a consequent change in how the program runs. This is much more than a simple porting issue. It may indeed be quite simple to change a few lines of the program in order to get it to run on another platform, but the fact that *any change at all* had to be made means that the language itself is not fully defined and thus can not be used for the precise definition of anything else.

A language for writing executable specifications is completely different, though superficially similar. Such a language actually *starts* with the defining document that was conspicuous by its absence in the previous case. Basically of lot of smart mathematicians and computer scientists work together over rather a long time to produce this defining document which gives a "formal semantics" for the language. The language itself will probably end up looking like a rather restricted programming language, perhaps missing some common programming constructs (the **goto** statement would be among the first to be removed for instance). With any luck, it is precisely the programming constructs which it is hard to be precise about which get left out.

If you are writing a safety-critical program, then you will want to be absolutely sure about what your program can be expected to do in operation, so you will probably be willing to trade some expressive power in your programming language, for the one massive advantage that the new language can give you, which is the existence of a complete and rigorous definition of the language which can be appealed to in the event of any dispute or question. There are not many such languages in the world at present, but their numbers are growing.

My group has chosen to use the framework provided by the CSP (Communicating Sequential Processes) model which was proposed by Prof. C.A.R. Hoare at Oxford and has been worked on for over fifteen years since then, both here and elsewhere. A major result of all this work is that there now exists not only a formal semantics for this language, but also a set of rules for transforming programs. Lest this seem too dry and uninteresting, and just before give up and move on to the next article, let me say that it is precisely this which makes it possible to **remove the distinction between hardware and software**, in the hope that this will keep you reading!

2.1 On The Advantages Of Not Writing A Program.

There are several good reasons for writing an executable specification rather than a program. Firstly, and most obviously it is a formal specification of the desired system. We can have much more confidence in such a specification than one written in technical English. We have probably all suffered from specifications which were incomplete, ambiguous, contradictory, or were interpreted by an implementor in a way not envisaged by the author. Moving to an easily understood formal specification can help eliminate such problems.

Secondly, since this specification is in a mathematically-based language it is open to checking, by program, for certain kinds of mistake. This is something that is not even remotely possible with specifications written in English. The sorts of things that can be checked for are that there are no missing parts of the specification, that everything that is in the specification is actually relevant, or that the various parts of the specification agree precisely on their interfaces with each other.

Thirdly, since an executable specification can also be regarded as a program, it can be compiled and executed on a computer and we can check that it actually behaves as we expect. So we have, almost for free, a simulation of the system that we want and it is available to us very early in the design cycle. If this program were to execute quickly enough on cheap enough hardware then we might already have the implementation that we seek. Here we are concerned with the situation when this is not the case and there is simply no alternative but to implement the system as special-purpose hardware, whose behaviour has been captured by the specification.

Fourthly, the program can be transformed into other forms which might be more useful than the original one. The transformation rules that come as part of a good specification language allow us to treat the specification/program as an object and to massage it into any number of different forms. If we follow the rules, we are guaranteed that we will not and up with a program that operates in any way worse way than the original one.

To take stock, we have a formal executable specification of the desired behaviour and what we have to produce is a description of some hardware that implements that behaviour. That description will be a circuit design consisting of many logic gates wired together. Normally, we'd give our specification to a trained engineer who would design and implement the hardware. At this point, our approach has already given us a cost-effective way of designing new digital systems, with a higher degree of confidence in the results than traditional methods offer, due to the presence of a formal specification which serves as a mutually-comprehensible reference document and interface between the designer and the implementor.

However, instead of turning the specification over to a human circuit designer, the most important advantage of our approach, *program transformation*, now comes into play.

3 Turning Programs into Hardware

Our chosen language is based on C.S.P.[1], and is essentially a subset of the occam language. These languages have been the subject of intensive research at Oxford and elsewhere for over fifteen years. The results of these efforts have directly contributed to the success of the work reported here.

Whilst our language is small, it nevertheless contains all the elements necessary to represent both sequential and parallel programs. Because of the massive amount of prior work on this language framework, it also has a set of transformation rules which are guaranteed to be correct (in other words it has a transformational algebra). These transformation rules support the refinement of user programs into hardware or software or mixed implementations. Here, refinement is a specific term which means that we use transformation rules in such a way that we can be guaranteed not to make the program worse-behaved by transforming it; however, we are allowed to make it better-behaved than the specification if we choose.

To give a flavour of these laws, we present a few of the simpler ones here. The interested reader is referred to [2] which justifies the basic laws defining these programs. The laws use a different syntax to the programs to aid readability.

Law 1 : Sequence.
These are some of the laws obeyed by the sequential composition operator (denoted by an infix semicolon), where P, Q, and R are arbitrary processes.

$$\begin{array}{lllll} 1.1 & (P\,;\ Q)\,;\ R & = & P\,;\ (Q\,;\ R) & \\ 1.2 & \mathbf{skip}\,;\ P & = & P\,;\ \mathbf{skip} & = \quad P \\ 1.3 & \mathbf{stop}\,;\ P & = & \mathbf{stop} & \end{array}$$

The process **skip** 'does nothing', and **stop** is the process which represents a broken program. The laws basically say that sequential composition is associative, that it has **skip** as unit, and **stop** as left zero.

Law 2 : Parallel
Among many others, the parallel operator $||$ satisfies the following laws, where ch is an internal channel. The process $ch\,?\,x$ is willing to accept a value transmitted over channel ch and place it in variable x. The process $ch\,!\,e$ is willing to transmit the value of the expression e over channel c.

$$\begin{array}{llll} 2.1 & (ch\,?\,x\,;\ P)\ ||\ (y := e\,;\ Q) & = & y := e\,;\ ((ch\,?\,x\,;\ P)\ ||\ Q) \\ 2.2 & (ch\,?\,x\,;\ P)\ ||\ (ch\,!\,e\,;\ R) & = & x := e\,;\ (P\ ||\ R) \end{array}$$

The first law shows that an assignment can be pushed forward through the parallel operator (because the variable can't be used in the other arm of the parallel construct). The second law very succinctly specifies what communication actually means in a parallel system, by equating communication with assignment.

Law 3 : Conditional.
The conditional process $P \lhd b \rhd Q$ (read as P IF b ELSE Q) has the following basic laws:

3.1	$(P \lhd b \rhd Q)\,;\ R$	$=$	$(P\,;\ R) \lhd b \rhd (Q\,;\ R)$
3.2	$(P \lhd b \rhd Q)$	$=$	$Q \lhd \neg b \rhd P$
3.3	$P \lhd \mathbf{true} \rhd Q$	$=$	P

4 Normal Form Programs

Using these together with the other laws, we can transform our original program into a particular form, namely a *Normal Form Program*:

```
all_variables := 0;
WHILE NOT Finished (some_variables) DO
    all_variables := next_state (all_variables);
```

We know that this program can never be any worse than the original program, as we have derived it by applying the proven-correct laws.

This program will in general appear much more complex than the original. That is because all of the program control structures (If, While etc.) of the original program will have been folded up into the usually very complex `next_state` function. Our normal form programs consist of a single parallel assignment embedded in a loop. Such normal form programs capture all the parallelism explicitly available in the original program, and they can be generated rapidly by a program which applies the transformations. The normal form program is much harder to understand than the original program, but then it is not intended for human consumption. These normal form transformations are presented as a set of direct mappings from language constructors to circuit diagrams in [3].

5 Hardware Interpretation

The justification for our particular normal form, is that it can be interpreted directly as a hardware circuit. Everything on the left hand side of the simultaneous assignment can be interpreted as hardware storage devices (synchronously clocked flip-flops in our case), and everything on the right hand side can be interpreted as a set of logic gates. This is possible since all the state of the system appears on the left-hand side and none of the expressions on the right-hand side require any sequential computing steps. The expressions are all simple logical and arithmetic combinations of variable values and constants. The simultaneous assignment has the entire state of the system on its left side, and the next-state function on its right; the system thus represents a single finite-state automaton. In everything but superficial appearance, this program *is* the circuit diagram of the hardware that we seek. Note that the user has not had to know anything about hardware in order to produce this implementation.

What we have in effect done is to move smoothly from the world of computer programs to the world of electronic circuits, taking with us a guarantee that we have not introduced any errors by crossing this boundary. If we wanted to justify this step completely, we would model each of the circuit elements with an occam program, combine the separate programs with the parallel combinator according to the circuit diagram, and then show that this program also refines the original. See [5] for further details of this proof step.

The normal form program, when interpreted as hardware, proceeds with the parallel assignment being executed once each clock cycle. The number of clock cycles for the program to complete represents the sequentiality of the program and the complexity of the assignment statement represents its parallelism. Using the laws, we could further trade off the sequential and parallel aspects against each other to achieve different cost/performance constraints for the hardware. In practice we would do this by transforming the source program and re-compiling since our current compiler guarantees to preserve the parallel and sequential elements.

In fact our system has deliberately been engineered so as to have a very simple timing behaviour. In essence, assignment and communication between ready processes each take one clock cycle, and all other language constructors have no overhead in time whatsoever. This means that we have a high degree of control over real-time behaviour which is attractive for many time-critical applications such as the one presented here.

Since we have encompassed software and hardware in the same theoretical framework, it is now possible to compile complete *systems*, with both hardware and software components and to do it from a single source program. We can use this *hardware/software co-design* strategy to change the implementation of a system by trading off the hardware and software components against each other to achieve desired cost and performance measures. As a further advantage, development of the normally problematical hardware/software interface becomes risk-free, since it is derived automatically.

6 Hardware Implementations

We use Field-Programmable Gate Arrays from Xilinx[7] which enable us to implement hardware almost instantly. Each of these chips contains circuitry which can rapidly be parametrised so that it behaves like an arbitrary circuit. The hardware design shown above is first expanded into a *netlist*, by simple macro-expansion of the multi-bit data-objects and the operators, into a collection of gates and flip-flops. This netlist is then given to vendor software which converts it into a set of bits for parametrising the particular FPGA chip chosen. The FPGA chips we use are based on static ram technology, so that parametrising them is simply a matter of writing data into the on-chip static ram. This means that we can generate new circuits quickly and as often as we like by a purely software route.

The new shaft encoder interface board that was constructed with our hardware compilation technology consists of little more than the FPGA chip itself, a chip to communicate directly with the control computer, and a few non-digital components necessary for each shaft encoder. It took only a few hours to design and hand wire this board.

An additional result of using FPGAs is that the physical construction of the hardware can be started (and maybe even completed) before the full specification of the system is available. We can defer a great many design decisions and achieve a high-degree of product flexibility, even to the extent of post-delivery reconfiguration. In our case, the same encoder interface board has been re-used a number of times with significantly different interface circuits without resoldering a single wire.

7 Summary

In summary, we have shown how we use parallel programs as specifications of systems and automatically generate the hardware and software components. In straightforward cases, we can go from problem specification to a working system in days, or even hours. At the fastest, we have used our parametrised processor software [8] to design a new microprocessor for a specific application completely automatically, to compile the processor into hardware, place and route it, and have a working 10MIPs processor, all in under 10 minutes.

We have indicated how hardware compilation technology allows us easily to re-engineer systems to meet a variety of cost/performance constraints, and how it significantly reduces risk in a notoriously difficult area of systems design.

We are investigating a number of application areas and we have so far found this approach has useful benefits in most of them. In particular, we are looking at methods of providing fast access to large databases using hardware search engines. We have demonstrated some basic graphics and image processing algorithms implemented in hardware. We have also implemented algorithms in data compression, computational chemistry, neural networks, and text processing.

We are also investigating hardware/software co-design in a more general-purpose context. To support this work, we have designed and built a system consisting of a 32-bit microprocessor (an Inmos T805 transputer) and a dynamically-reconfigurable FPGA (a Xilinx 3195). Each of these has its own local memory and can share the transputer bus for fast communications. The transputer can also download the FPGA and control a 100MHz frequency synthesiser to generate an arbitrary clock signal suited to the particular configuration in the FPGA. This system, known as HARP1, is implemented as a size-6 Inmos-standard TRAM module[9]. Thus, it can be readily integrated with other copies of itself, and with other commercial computing and input/output modules, and with various host platforms. In this way we can exploit coarse-grained parallelism across a number of TRAM modules as well as medium-grained (pseudo) parallelism in the transputer, and very fine-grained (true) parallelism in the FPGA hardware.

We are using the HARP1 system as the host for most of our experiments in hardware/software co-design as it is sufficiently general-purpose to host a wide variety of applications. In the future we hope to address some more specific application areas by designing further microprocessor/FPGA boards; in particular, we hope to investigate some of the processing and interfacing problems in the area of multi-media systems.

In summary, we believe that this type of technology has great possibilities for contributing significant improvements in productivity and cost-effectiveness in many areas of computing and engineering. By the use of programming and compilation tools we hope to contribute to the production of effective, reliable, and flexible hardware/software systems from a single source program.

References

[1] C.A.R. Hoare, *Communicating Sequential Processes*, International Series in Computer Science, Prentice-Hall, 1985.

[2] A.W. Roscoe and C.A.R. Hoare, 'Laws of occam programming', *Theoretical Computer Science*, **60**, 177–229, (1988).

[3] Ian Page and Wayne Luk, 'Compiling occam into fpgas', In Moore and Luk [4], 271–283.

[4] *FPGAs*, eds., Will Moore and Wayne Luk, Abingdon EE&CS books, 1991.

[5] C.A.R. Hoare and Ian Page, 'Hardware and software : The closing gap', *Transputer Communications*, **2(2)**, 69–90, (June 1994).

[6] C.A.R. Hoare and He Jifeng, 'Refinement algebra proves correctness of a compiler', in *Programming and Mathematical Method: International Summer School directed by F.L. Bauer, M. Broy, E.W. Dijkstra, C.A.R. Hoare*, ed., M. Broy, volume 88 of *NATO ASI Series F: Computer and Systems Sciences*, 245–269, Springer–Verlag, (1992).

[7] Xilinx, San Jose, CA 95124, *The Programmable Gate Array Data Book (1993)*.

[8] Ian Page, 'Parametrised processor generation', in *More FPGAs*, eds., Will Moore and Wayne Luk, Abingdon EE&CS books, (1994).

[9] Inmos Limited, *The Transputer Development and IQ Systems Databook*, Inmos Limited, 1991.