Automatic Systems Synthesis : A Case Study.

Ian Page.

October 1992 (revised Nov. 1993).

Abstract

This paper describes the experience of using our hardware compilation technology to design and build hardware to interface a set of sensors to the controlling computer of a robot system. It is a common experience that producing hardware solutions for application problems is fraught with difficulties. However, the progress of hardware and software technologies, coupled with judicious use of abstract mathematics, are now beginning to enable completely different ways of implementing systems. This is particularly true when hardware components of a system are implemented by dynamically reconfigurable FPGA (Field Programmable Gate Array) chips. In the Hardware Compilation Research Group at Oxford, we are investigating the automatic derivation of complete systems, including hardware and software components, from executable specifications. We are trying to make the implementation of hardware/software systems as much like programming as possible, thus reducing emphasis on the difficult issues of hardware design. Here, we demonstrate the approach that our group is taking to designing and implementing parallel computing systems for time-critical applications by reference to one particular example.

1 Introduction

It has always been possible, and often necessary, to design special-purpose hardware solutions to information processing problems. The prime motivation for doing this is when low-cost, general-purpose solutions as typified by the microprocessor do not have the performance required by the application. The development of such hardware solutions are not for the faint-hearted however. Development costs are usually high and timescales can be very long, making this an unattractive option for many application problems.

The emergence of dynamically reconfigurable FPGA (Field Programmable Gate Array) chips is rapidly changing the nature of hardware implementations, since hardware can be constructed in milliseconds by downloading parametrisation data to such chips. If this hardware innovation is coupled with high-level tools for designing the hardware components of a system, then it is possible in some circumstances to reduce the entire hardware design and implementation to a purely software process, thus considerably speeding up the time to produce new systems.

The particular application problem that we consider here is the design and implementation of new interface hardware that was needed to link a set of angular position sensors (shaft encoders) to a controlling computer in a robot system being built for research purposes. It was required that the interface should handle all the low-level details of interaction with the physical sensors and present a high-level, command-driven interface to the control computer.

The shaft encoder device has two output signals generated from photo-sensitive detectors. Their light input is interrupted by opaque stripes on a transparent plastic disc

attached to the shaft. Two detectors and two sets of stripes in quadrature are needed so that the direction of rotation can be determined as well as its amount. Rotating the encoder disc results in two digital pulse streams which must be processed at high speed to extract the desired angular position information as well as other variables such as angular velocity and acceleration.

An existing circuit board was designed in our Engineering Department a few years ago to take the outputs from four shaft encoders and interface them to the control computer. It took six weeks of intensive effort to design, build and test this board which used commercial shaft encoder chips. It is probably fair to say that six weeks was a very short time indeed to produce this particular implementation. The new task was to design and build an interface board for another robot, but offering a higher speed of operation, higher accuracy, additional functionality, and smaller physical size. We now describe how we achieved these stated goals, as well as also achieving lower development cost, reduced development time, and increased flexibility for the new interface.

2 Specification

The first step was to ensure that we completely understood what was wanted from the hardware interface. This was achieved by talking to the engineer who wanted the interface. After extracting his requirements, we proceeeded by writing a program. The following is a top-level view of that program, suppressing detail for clarity:

```
PAR
  Encoder_handler_1
  ......
  Encoder_handler_n
  multiplexor
  command_interpreter
```

At this level, the program simply shows what modules are in the system and that they all operate in parallel. The language is essentially occam [1] although we take a few liberties here with the syntax of that language. At a more detailed level, we now look inside one of the encoder handler modules. The following is a fragment of one of these modules which converts two-bit values from the shaft encoder into a number which represents the angular position of the shaft. It is normally embedded in a loop and thus deals with an infinite sequence of two-bit encoder values:

```
SEQ
  Encoder ? Current
  IF Current ≠ Previous
    THEN
      IF Current BIT 0 ≠ Previous BIT 1
        THEN
          Angle := Angle+1
        ELSE
          Angle := Angle−1
  Previous := Current
```

This program fragment does three things in sequence. Firstly, it reads the current output value from the shaft encoder device and stores the two bits in a variable called **Current**. The sensor is always ready to output its value, so the synchronisation implied by the input command from the channel **Current** always occurs immediately.

Secondly, it determines whether the encoder has moved since it was last examined by comparing the value in **Current** with the value in the variable **Previous**. If it has, then a comparison between a bit from each value determines if the shaft has moved one unit clockwise or anticlockwise; this is recorded by incrementing or decrementing the value in the variable **Angle**.

The third action is simply to update the **Current** position to be the same as the **Previous** position so that we can repeat this program indefinitely. In fact we have to repeatedly run this program much faster than the shaft encoder could ever move, which in practice means executing the program many millions of times each second. If we do this, we can be certain that the **Angle** variable is a true representation of the physical position of the robot arm. Note that we have explicitly chosen to build a synchronous system which polls the encoder inputs, although this is certainly not the only way of solving this particular problem.

It has been necessary to leave out a large amount of detail in this presentation, but it is worth noting that a simple, but complete, version of this program was constructed within two hours of the problem first being presented. Later versions were considerably improved from this one, which is retained for this presentation due to its simplicity.

3 Formal Specifications

There are several good reasons for writing such a program. Firstly, the program is a *formal specification* of the interface. We can have much more confidence in such a specification than one written in technical English. Many of us have 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. The claim that an occam program is a formal specification of a behaviour lies in the fact that the occam language has a formal denotational semantics.

Secondly, since this specification is in a formal, mathematical, language it is open to checking, by program, for certain kinds of mistake, 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 this specification is also a program, it can be 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 hardware that we want and it is available to us very early in the design cycle. It was through just such a simulation that our engineering colleague was able to confirm that we had indeed captured precisely the behaviour that he wanted from the interface hardware.

Fourthly, the program can be transformed into other forms which might be more

useful than the original one; we cover this point in more detail below.

In fact, the only thing wrong with our program as a final implementation of the interface is that it is not fast enough. When executed on a conventional computer, this program would be perhaps a hundred times slower than necessary. There is simply no alternative but to implement our interface as special-purpose hardware, whose behaviour has been captured by our program.

To take stock, we now have a formal and 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.

4 Algebraic Transformation

A major feature of our language is that it has a well-understood semantics; that is, we know precisely what every program written in the language means. In other words, there are no programs which are anything less than completely specified and which, in principle therefore, can be completely understood. Our language is based on C.S.P.[2], 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 subset is small, it nevertheless contains all the elements necessary for a language competent to represent sequential and parallel programs, and it also supports their refinement into hardware or software implementations. As well as a denotational semantics, occam also has an algebraic semantics which has been proven congruent with the denotational semantics. This gives us a powerful base for reasoning about programs using a set of proven-correct algebraic laws. To give a flavour of these laws, we present a few of the simpler ones here. The interested reader is referred to [3] 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.

1.1 $(P\,;\ Q)\,;\ R \quad = \quad P\,;\ (Q\,;\ R)$
1.2 $\mathbf{skip}\,;\ P \quad = \quad P\,;\ \mathbf{skip} \quad = \quad P$
1.3 $\mathbf{stop}\,;\ P \quad = \quad \mathbf{stop}$

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.

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

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:

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

5 Normal Form Programs

Using these together with the other algebraic laws, we can transform our original program into a particular form, namely a *Normal Form*. The following is the normal form program derived from the encoder handling fragment above:

```
SEQ   Start,C0,C1,C2,C3,C4,Finish := 1,0,0,0,0,0,0
  WHILE  ¬Finish
    SIMUL
      Previous := Current ◁ C2 ∨ C3 ∨ C4  ▷ Previous
      Current  := Encoder ◁ (Start ∨ C0) ∧ Encoder_Rdy  ▷ Current
      Angle    := (Angle+1◁ Decr ▷ Angle−1)◁ Decr ∨ Incr ▷ Angle
      C0       := (Start ∨ C0) ∧  ¬Encoder_Rdy
      C1       := (Start ∨ C0) ∧ Encoder_Rdy
      C2       := Incr
      C3       := Decr
      C4       := C1 ∧  ¬Changed
      Finish   := C2 ∨ C3 ∨ C4
      Start    := 0
WHERE
      BitDiff  = Current BIT 0 ≠ Previous BIT 1
      Changed = Current ≠ Previous
      Incr     = (C1 ∧ Changed) ∧ BitDiff
      Decr     = (C1 ∧ Changed) ∧  ¬BitDiff
```

Some liberties have been taken with normal occam syntax to improve readability. In particular, the WHERE clause is introduced to extract common sub-expressions and aid readability; its semantics are those of textual substitution. The SIMUL keyword introduces a large simultaneous assignment statement which is better displayed on a number of text lines, rather than one. We know that this program can never be any worse than the original program, as we have derived it by applying the proven-correct algebraic laws. Some of the laws are inequalities (refinements) rather than equalities; these permit us to improve a program during transformation, for instance by making it more deterministic or weakening its preconditions.

Although this program looks much more complex than the original, it does in fact have a rather simple structure. 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 algebraic 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 [4].

Additional 'control' variables are introduced by the transformation process. The wait loop implied by the input statement is controlled by C0. The control variable for the IF statement is C1 and is true just when that statement is being executed. C2-4 are the control variables for the following three assignment statements. No optimisations have been performed on this program, although some control states clearly may be merged. Such optimisations can also be represented as source-level transformations.

6 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 much, if anything, about hardware in order to produce this implementation.

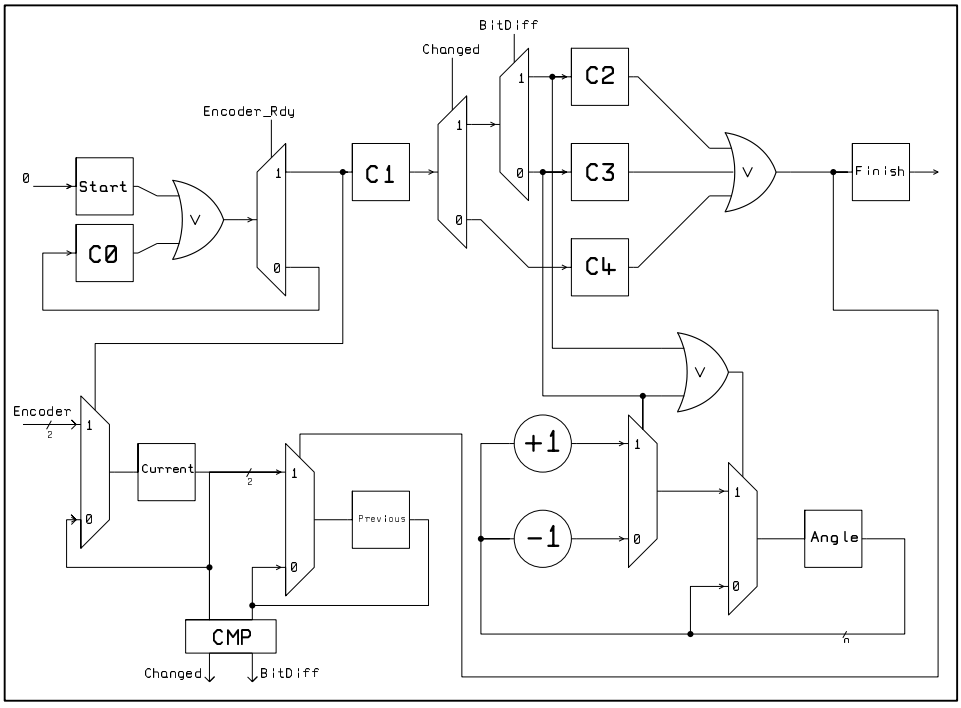

Figure 1: Equivalent Circuit Diagram

The circuit diagram (Figure 1) contains exactly the same information as the normal form program. 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 algebraic 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 calculus. 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. In general, we would transform the program so that at the top level it would be in the form:

```
CHAN in, out : PAR
  hardware_process (int, out)
  software_process (out, in)
```

where the first process is compiled into hardware using the techniques already demonstrated, and where the second process is compiled into machine code for some host processor in the conventional manner. In related work [6], we are also developing techniques where the compilation of the software component into machine code can also be proven correct.

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.

7 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.

8 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 con-

tributing 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] Inmos Limited, *Occam 2 Reference Manual*, International Series in Computer Science, Prentice-Hall, 1988.

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

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

[4] Ian Page and Wayne Luk, 'Compiling occam into FPGAs', in *FPGAs*, 271–283, Abingdon EE&CS Books, (1991).

[5] C.A.R. Hoare and I. Page, 'Hardware and software : The closing gap', *Transputer Communications*, **1234**, XX–XX, (May - perhaps 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 *FPGAs 93*, to be published by Abingdon EE&CS Books (probably), (1993).

[9] Inmos Limited, *The Transputer Development and iq systems Databook*, Inmos Limited, 1991.