Automatic Design and Implementation of Microprocessors

Ian Page
Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, England OX1 3QD

Abstract. This paper reports recent work on the automatic design and implementation of microprocessors to suit particular applications. We use our own hardware compilation system to produce synchronous hardware implementations of parallel programs and have constructed platforms incorporating Field Programmable Gate Array and transputer components to host such implementations. Our chosen language, Handel, is essentially a subset of occam with as few extensions as necessary to address the special nature of hardware implementations. The system reported here can take a Handel program and, rather than mapping it directly to hardware, will first transform it into a custom microprocessor, expressed as another Handel program, together with a machine code program. The hardware compiler is then invoked to construct the resulting application-specific microprocessor. This approach may have benefits for applications where the kernel, or 'inner loop', is too complex to be implemented as parallel hardware, but where a speed increase beyond that possible with off-the-shelf microprocessors is desired.

1 Introduction

The Hardware Compilation Research Group at Oxford is working on a variety of techniques to compile programs into a mixture of hardware and software appropriate to particular applications. We typically implement the special-purpose hardware parts in Field Programmable Gate Arrays (FPGAs) [20] so that the production of a working hardware/software system can be reduced to an entirely software process.

A compilation system has been constructed [10] which maps programs in our Handel language, based closely on occam [12], into netlists for synchronous hardware. Expressions are always compiled into combinational logic and thus evaluate in a single clock cycle. The control circuits are such that assignment and ready-to-run communication each take one cycle, and all other language constructors add no overhead in time, giving the language a remarkably simple timing calculus.

The Handel programs in this paper are denoted in an *ad hoc* mixed occam/CSP style for clarity and compactness. In fact, Handel programs only exist in abstract syntax form, for ease of handling by automatic transformation systems; a concrete syntax for it has not yet been defined. Handel programming is carried out in the SML [17] language. This use of SML as a meta-language for representing and transforming Handel programs has turned out to be extraordinarily useful in general, and in particular for parametrisation of abstract programs.

The availability of a hardware compiler naturally results in two implementation paradigms; user programs can be compiled into (i) parallel hardware, or (ii) sequential machine code for a standard microprocessor. We typically use both of these paradigms simultaneously so that the time-critical parts of the application are mapped into fine-grained parallel hardware and the rest is implemented in software [14]. These two paradigms are essentially at two ends of a spectrum of possible implementation strategies. At one end we have high-speed, expensive, application-specific, parallel hardware implementations; at the other end we have very cost-effective, sequential, software implementations on general-purpose microprocessors.

This paper reports on our work to develop a new paradigm which lies between these two; our aim being to exploit a significantly different point in the cost/performance spectrum. The starting observation for the work was that parts of applications which were suited for microprocessor implementation could often run faster if the architecture of the microprocessor were slightly different. This might be humorously characterised by the machine-code programmer's lament "of course my program would run much faster if only this computer had a reverse bit-twiddle instruction". The possibility of designing and implementing new processors to fill such gaps has always existed, but it has rarely been possible to contemplate because of cost. Our contention is that this situation is transformed with the availability of hardware compilation systems, particularly when these are combined with FPGA implementation. The implementation of the DLX microprocessor [16] reported in [3] is an example of the large reduction in effort necessary to implement microprocessors via FPGA technology, starting from a circuit diagram in this case.

An Instruction Set Simulator (ISS) program is a common way of documenting, specifying, and simulating the behaviour of a processor. Such programs are very simple to construct compared with the design of the processor itself, and they are frequently built even before any serious work begins on the design of the processor. Simply by presenting such an ISS to our hardware compiler, the output *is* the implementation of the processor we want. Without making any claims about the viability of this method for large-scale, state-of-art processors, we have demonstrated that fast and moderately complex microprocessors can be implemented within hours on general-purpose FPGA hardware. The work by Lewis [11] on acceleration of simulation and other algorithms via application-specific microprocessors reports an impressive implementation using non-automatic design techniques.

Having built a number of simple microprocessors by hand-designing and coding the ISS programs and compiling them into hardware, it became clear that much of the work in designing application-specific processors could be automated. By taking an abstract model of a processor and then parametrising it depending on the code it will have to execute, it has proven possible to produce concrete implementations of processors which are targeted on given applications. Two recent student projects here have done much work in bringing these ideas to fruition. That work is reported in [4] and is developed further in [19].

2 The Scope for Parametrisation

It is possible to conceive of virtually every aspect of computer architecture being parametrised. To make progress, we have concentrated on some of the major aspects and intend to develop our techniques outwards from this core to incorporate further architectural aspects. At the top level of parametrisation, we believe that a single parameter should select the gross style of architecture (at least until we find convincing common ground between the styles).

The processor architecture can in principle be selected from a library of processor generators supporting a set of available parametrised processor styles, such as RISC, CISC, 1/2/3 bus, stack/register oriented, etc. Each architecture naturally needs an associated compiler for each source language. We have only constructed three models so far. One is a small, *ad hoc*, stack-based processor, another is based on a simplified model of the Inmos transputer reported by May and Keane [15], and the other is based on the Acorn ARM2 processor [1]. Each of the models is parametrised in different ways specific to that architecture. We have not yet learned enough that we can treat the parametrisation of all architectures in the same way, although we wish at least to develop a common framework for such parametrised processors.

Currently we use judgement to select one of our processor generators and apply it to the user program. In principle it might be possible to make even the choice of gross architecture automatic; an obvious method being to compile the user program into each architecture, measure its code and hardware size, speed etc. and select that architecture which best fits the selection constraints given by the user. This is almost entirely in the realm of future research at present.

Each processor generator consists of an abstract compiler and an architecture refiner. The processor generator first applies the compiler to the given user program. The compilers are constructed along standard lines, and are all quite tiny in our current systems. Working with abstract syntax for input and output helps considerably to keep these compilers small and easily managed. At this stage details such as instruction encodings and datapath widths are not fixed, so the output of the compiler is an abstract machine code program.

The abstract machine code program is then examined by the architecture refiner to obtain various measures and statistics for tailoring the processor. The refiner is an *ad hoc* SML program as we don't yet understand enough to enable us to represent all architectures and their parametrisations within a common framework. However, for the most part the processor is represented as an abstract ISS program expressed in Handel. Throughout the refinement process this ISS program is gradually made more concrete and has optimisations applied, until it is completely defined, at which point it can be submitted to the Handel compiler and converted into hardware.

As a simple example of such an abstract-to-concrete refinement, the machine code program declarations reveal the maximum bit-width of any integer variable and this is used to set the minimum datapath width of the main processor datapath. As an example of an optimisation, consider a machine code program which never uses the Stack Reverse instruction though it is available in the processor model. As there is little point in including that instruction in the repertoire of the final processor, it is removed. In this world, whenever you want to run a different program, you manufacture a different processor for it!

The following list briefly describes most of the areas of parametrisation that we have so far been able to apply with some degree of success:

1. Instructions which are not needed are deleted
2. Unnecessary resources are removed, such as an expression stack if no stack-based instructions are used, or a floating point unit if none is needed.
3. External resources such as RAMs, ROMs, and channel-based links to other external devices are added as required.
4. The bit-width of resources such as general-purpose registers, op-code and operand fields of the instruction register, and the instruction pointer register are changed to suit the size of program.
5. Expression stack depth is set from a static analysis of the code.
6. A language stack is included or excluded; its size can be determined statically if the program is non-recursive.
7. Instruction operands are shortened to fit into small instruction fields, adding sign and/or zero extension capability as necessary.
8. Instruction operands are tabulated so that long operands can be referenced by short fields in the instructions which index an in-processor lookup table.
9. Instruction op-codes are optimally assigned and instruction decoding re-organised to minimise instruction decoding delay.
10. The instruction set is extended, if requested, from user-supplied instruction definitions.
11. The processor can optionally be pipelined to achieve overlapped execution of instruction fetch and execution.
12. Bootstrap facilities are added as required. These include power-on boot from a ROM or channel, and the provision of a reboot instruction in the instruction set.

3 An Abstract Processor Example

Many styles of processor are possible, RISC, CISC, dataflow etc. So in order to make progress we must pin down some architectural details of a chosen style. We do this by informally defining the abstract instruction set of the processor, using engineering insight and experience; very much as most real-world processors have been designed.

An architecture refiner is a fairly large SML program and thus can't be shown here. We therefore choose first to show a sample of the output of a processor generator, but where no optimisation has been done. The program in Figure 1 is a simple application that inputs two integers, then calculates and outputs their sum of squares. When this program is compiled directly into hardware the Handel compiler produces a netlist with 58 latches and 3340 gates. The combinational logic is large since there are two 24-bit parallel multipliers implied by this program.

We can also give the exact same program to one of our simple processor generators, which produces the ISS program shown in Figure 2. Note that this processor has been

```
PROC main (CHAN OF INT.24 cin, cout)
  INT.24 r1, r2 :
  WHILE TRUE
    SEQ
      cin ? r1
      cin ? r2
      cout ! (r1 * r1) + (r2 * r2)
:
```

Figure 1: Simple application program.

```
PROC main (CHAN OF INT.24 cin, cout)
  VAL [16] INT.24 code IS [9437184, 2097152, ... , 11534336] :
  INITIAL INT.4 iptr IS 0 :
  INT.24 inst, areg, breg, creg :
  [4] INT.24 mem :
  WHILE TRUE
    SEQ
      inst, iptr := code [iptr], iptr + 1
      CASE inst \\ 20
        0 : areg, breg, creg := (inst <- 20) @ C 0, areg, breg --LDC
        1 : areg, breg, creg := mem [inst <- 2], areg, breg    --LDA
        2 : mem [inst <- 2], areg, breg := areg, breg, creg    --STA
        3 : areg, breg := areg + breg, creg                    --ADD
        4 : areg, breg := areg * breg, creg                    --MUL
        5 : areg, breg := breg, areg                           --REV
        6 : iptr := inst <- 4                                  --JMP
        7 : IF areg <> 0 THEN iptr := inst <- 4 ELSE SKIP      --JTR
        8 : IF areg <  0 THEN iptr := inst <- 4 ELSE SKIP      --JLT
        9 : cin  ? areg                                        --IN
        10: cout ! areg                                        --OUT
        DEFAULT : STOP
:
```

Figure 2: The resulting ISS program.

constructed specifically for purposes of presentation; there is no claim that it is a useful general-purpose processor (indeed, we would probably have failed if that were the case).

Any programmer should be able to understand the above ISS program, with the exception of the three non-occam operators introduced in Handel expressly to deal with bit-string extraction and concatenation. Although these can be related to occam shift and mask operations, they are fundamentally different as they deal with expressions of arbitrary width; in Handel, expressions of different widths are essentially of different types.

The motivation for these operators, as for the width-typing of all variables and expressions in Handel, is that that we cannot afford to pay the price of (say) 32-bit hardware registers to store a single Boolean value. This is an example of the vastly different costs of hardware and software implementations. In software, one would rarely want to pack 32 Booleans into a word because of the massively increased cost of accessing them.

The Handel bit-string manipulation operations are:

`e <- n`	delivers the least significant `n` bits from the expression `e`,
`e \\ n`	drops the least significant `n` bits from the expression `e`,
`e1 @ e2`	the bitwise concatenation of `e1` (l.s. end) and `e2`.

These operations are frequently necessary in applications and are close to the hardware mechanisms of bus restriction and merging. Note that no gates are needed to implement these operations in hardware. We also feel that these operators are more expressive than their shift and mask counterparts in standard sequential languages.

The data width specified by the source program is what has made this particular processor 24 bits wide. The nature of its input/output has caused two channels and corresponding instructions to be added to the processor. In this case, the `code` rom and the data memory, `mem`, have been constrained to be the same width. The lengths of these memories has been padded out to the next power of two up, though they could have been trimmed to exactly the size required. We have chosen to use a 'Harvard' architecture with separate instruction and data memories, although it perhaps ought to be known as a 'Babbage' architecture since he used the technique somewhat earlier!

The entire instruction set of this trivial processor has been included by suppressing the automatic removal of unnecessary instructions. The default stack depth of three has also not been optimised. Our purpose in presenting it is simply to show the *style* of processors we are dealing with as the fully optimised processor programs are much more difficult to read. The netlist for this version has 214 latches and 2699 gates and is thus smaller than the previous version due to the fact that the two multiplications sequentially share a single 24-bit combinational multiplier. It is exactly this sort of sequential resource sharing that makes microprocessors an attractive implementation strategy for certain programs.

The microprocessor version is slower of course. The original program would produce an output every three clock cycles if kept fed with data; the processor version takes 26 clock cycles. This can be seen more clearly from the following assembler-style listing of the contents of the `code` memory:

```
IN;     STA 0;  IN;    STA 1;
LDA 0;  LDA 0;  MUL;   LDA 1;  LDA 1;  MUL;  ADD;
OUT;    JMP 0
```

Its speed could be doubled if automatic pipelining were invoked, and could be further increased if conventional code optimisations were applied to the machine code program, or if a squaring instruction were added to the instruction set; all of these are possible with our generators. The size of the processor version could also be reduced if the unused resources were removed.

As yet, we have not created microprocessors automatically from other than the sequential subset of Handel. However, transformations do exist to render parallel programs into sequential ones. In fact, the Handel hardware compiler itself, together with a simple parallel assignment scheduler, is exactly such a transformation agent.

4 Implementing the Parametrisation

The first step in producing a parametrised processor is to determine the bit-widths of the major processor resources, using information gleaned from the original program and from the compiled abstract assembler program.

The width of the data memory and the associated data paths is simply chosen to be the maximum of the widths of all variables and channels in the user program. There seems to be no simple way to choose multi-word representations automatically, so we sidestep the problem by treating such data refinements as *pre*-transformations on the application program. To tackle the problem of different word sizes in the application, we simply pad out all data representations to the size of the largest. Again, pre-transformations seem the correct way to handle this problem.

After compiling and optimising the application to abstract assembler code, the depth of the expression stack can be statically determined, as there is no recursion allowed in occam/Handel. This will typically be between 0 and 3, but it depends completely on the compilation and optimisation strategies chosen; it will be much greater if a single stack is used for both expression evaluation and procedure environments. We can choose between various stack implementation options depending on the size of a stack, and on user-provided constraints. Our currently-supported options are (i) hardware LIFO, (ii) on-chip ram with stack pointer, and (ii) off-chip ram, with or without on-chip stack-top caching. We use a rather nice algorithm for compiling stack-based code for expressions which Inmos developed for their occam compilers [13].

The width of the op-code is chosen by counting the number of different instructions used in the compiled application. The width of an instruction is determined by this and by the largest operands used in the program, including short, tabular, and long operands.

These and similar arguments are sufficient to set all resource sizes in the processor. Unfortunately, the simultaneous minimisation of all these values is combinatorial in nature. We use a heuristic which consists of making an initial estimate of certain values, minimising other values in that context, refining the estimate, and repeating until no further minimisation is possible (i.e. a steepest descent search).

5 Further Parametrisations

For the processor above, we have a very simple instruction format which packs an op-code and a single short operand into one instruction word. For more complex processors, we allow double word instructions in which the following word also contains part of the operand. The short operand and the additional word are simply concatenated to form the complete operand.

We also tabulate operands in order to save instruction encoding space. This is done on a per-instruction basis. Taking the LDC instruction as an example, the abstract assembler code is examined and if all the LDC operands will fit into the short operand format, then we are finished. If there are some long operands, and all distinct operands can be indexed by a short operand, then a table of the operands is built and the LDC instruction is modified to use this table. If there are too many distinct operands to be indexed, then this instruction is left untabulated. Clearly, other optimisation strategies are possible here which might improve performance, such as allowing both tabulated and untabulated instruction forms simultaneously. Conventional ROM optimisation strategies can be applied to the resulting tables. As a final optimisation, tables that contain only a single value are replaced by the value itself.

Branch instructions are treated similarly, except that the tabulation process interacts with the jump optimisation problem. In the conventional manner, we assume that all branches can be in short form and lengthen them only where necessary. This needs a quadratic iterative algorithm to obtain minimal code size. The additional problem is that if this process tabulates a jump instruction, and the table later overflows, then all the short tabular instructions have to be lengthened, effectively de-tabulating that instruction. We have not yet investigated whether the sharing of operand tables between instructions is a worthwhile optimisation, likewise the use of hierarchical tables.

5.1 User-specified Instructions

Our processor generators can also take note of user-provided constraints in the application program. In particular, fragments of program can be included as additional instructions in the processor. For example, the user could modify the program in Figure 1 as follows:

```
cout ! $((r1 * r1) + (r2 * r2))
```

where the $ operator signifies that the following expression should be incorporated as the microcode of a new instruction. In fact all that happens is that the expression is carried across uninterpreted to the CASE statement in ISS program

```
NEW_INST : areg, ... := ((r1 * r1) + (r2 * r2)), ...
```

thus forcing the interpretation to occur in hardware at run-time.

This is a particularly neat and simple way of achieving a measure of hardware-software co-design. It is simple because the operand of the $ operator is a single expression which compiles wholly into combinational hardware, so there is no control or data-state to worry about and the interface with the rest of the implementation is trivial.

5.2 Automatic Pipelining

In practice, the Handel code for the individual instructions, equivalent to the microcode in a more conventional implementation, will often be relatively simple. In the case where all operands of these code fragments are immediately available from on-chip variables, we can symbolically execute the code to reduce it to a single parallel assignment. At this point, the structure of the processor code will be (largely) a `WHILE` loop containing two sequentially-executed parallel assignments, one for the instruction fetch, and the other for instruction execution; each of them executing in one clock cycle.

The processor throughput can now be nearly doubled by overlapping the operations of instruction fetching and execution. This is done by replicating the fetch statement, and pushing it into each arm of the `CASE` statement, using the identities:

$$\begin{array}{lcl} \mathit{WHILE\ TRUE\ DO}\ (a\,;\ b) & = & a\,;\ \mathit{WHILE\ TRUE\ DO}\ (b\,;\ a) \\ (i1 \lhd op \rhd i2)\,;\ f & = & (i1\,;\ f) \lhd op \rhd (i2\,;\ f) \end{array}$$

Here, $i1 \lhd op \rhd i2$ (= `IF` op `THEN` $i1$ `ELSE` $i2$) represents the processor's microcode. Bringing the instruction fetch fragment a out to the head of the program implements the necessary priming of the instruction pipeline.

Transformations are then made to remove the sequential compositions and turn each arm into a parallel assignment. The instructions which operate on arithmetic and logical data are typically straightforward to deal with, but there are three sources of problems with this transformation: (i) input/output instructions, (ii) branch instructions, and (iii) double word instructions, all of which are covered next.

5.3 Input/Output

Our task of implementing input/output instructions is somewhat eased as our interpretation of Handel is consistently synchronous. This means that additional laws are true of Handel which are not generally true of occam. In particular, sharing of variables between parallel processes in parallel statements is perfectly well defined as long as we provide a proof that no variable can ever be updated more than once in any single clock cycle.

If we take the example of a processor with a three-element on-chip stack, and look at the pipelined microcode for an instruction that inputs from an external channel to the stack, we find something like the following:

$$p := p + 1 \ ||\ ir := code[p+1] \ ||\ cin\,?\,a \ ||\ b := a \ ||\ c := b$$

where p is the instruction pointer, ir is the instruction register, cin is the external input channel, and a, b, c hold the expression stack.

Because of the synchronous interpretation of Handel, it is generally true that $(x := e \ ||\ y := f)$ is well-defined whenever x, y are distinct variables, even though x may be referenced in f, or y in e. In such a case the statement is equivalent to the occam parallel assignment $(x, y := e, f)$.

A direct consequence of this, together with the fact that channel communication is simply distributed assignment, means that in the instruction above there is no problem as the communication to a can only happen synchronously with the other assignments at the earliest, so a can not possibly be updated before the assignment $b := a$ executes.

However, if we turn to an output instruction that destructively outputs the top of the stack to an external channel we find that we can't immediately remove the sequential operation:

$$cout\,!\,a\,;\ (p := p + 1 \ ||\ ir := code[p+1] \ ||\ a := b \ ||\ b := c) \qquad (1)$$

There is simply no way of maintaining single cycle behaviour when the channel is not immediately willing to accept output. It is either necessary to provide a way of withdrawing an offer to communicate, or rather better is to provide a mechanism to test readiness of a channel to communicate, which is what we do in Handel. Thus, we transform the fragment to:

$$(cout\,!\,a \;\|\; ass) \lhd READY(cout) \rhd (cout\,!\,a\,;\; ass)$$

where *ass* represents all the assignments in program 1 above.

5.4 Branch Instructions

In the simple processor model here, branch instructions actually cause no problem since it is possible in Handel to evaluate an expression, use it as the address for a memory reference to on-chip RAM, and use the result of the reference, all in a single clock cycle. Thus, the microcode for each branch instruction simply evaluates the branch condition and loads the p and ir registers for either the next sequential instruction or for the branch. For example

$$\begin{array}{rl} & p, ir \;:=\; p+1, code[p+1]\,;\;\; p \;:=\; opnd(ir) \lhd a < 0 \rhd SKIP \\ \Rightarrow & p, ir \;:=\; (opnd(code[p+1]) \lhd a < 0 \rhd p+1), code[p+1]. \end{array}$$

With this architecture, the conditional branch instructions are likely to contribute the longest combinational logic paths in the final hardware, and hence set the upper bound on the clock speed. If this is unacceptable in a particular case, then a further pipeline transformation can be applied to the program to split the next-instruction fetch into two or more clock cycles. This will have the effect of reducing the worst case combinational delay, but at the expense of more complex pipeline priming/flushing code. However, this is an inevitable consequence of using pipelining in a data-dependent computation, and at least the Handel programmer gets a choice of which strategy to use.

5.5 Double Word Instructions

With double words in the architecture, it is necessary to change the code access mechanism if it is required to maintain the execution rate of one instruction per clock cycle. Clearly, the instruction fetch mechanism needs to deliver at least two words per clock. Consequently, we automatically double the width of the code store interface. It is further necessary to pad out the concrete machine code so that all double word instructions that are the target of any branch instruction must lie on a double-word boundary. This maintains single cycle execution, at the cost of slightly greater code size. A further optimisation is to modify the microcode so that the instruction pointer is increased by 2 whenever a short instruction in the first word of the two-word instruction register also detects a padding (or `NO-OP`) instruction in the second word. With this modification, no clock cycle is wasted while 'executing' the padding instructions.

6 Using the Laws of Program Transformation

At this point, we leave the description of parametrised processors and look at the basis of a method for ensuring the correctness of processor-based implementations.

The programming language we use here is based on occam, which in turn is based on Hoare's CSP [6]. As a consequence there is a rich set of laws [18] that apply to our programs and a transformational algebra which can be used to 'massage' a user program into a form more amenable to a particular implementation technology.

Using the transformational laws, we can take a user program and massage it into the form of an ISS program together with the appropriate machine code program. As the laws are correctness-preserving, we know that any ISS program designed by this method must be correct, as must the machine code program that it runs. We show a simple example of this style of transformation, to give the reader confidence that there is indeed a way to move soundly from a user program to a processor-based version of the same.

We start with the simple user program:

$$a' := b' + c'$$

The first transformation is to replace each distinct variable in the user program with a variable in a linearly addressed memory, M. This is equivalent to the memory allocation task performed by a conventional compiler. We let the integer constants a, b, c stand for the distinct indices in the array corresponding to the variables a', b', c'. Thus the transformed program is:

$$M[a] := M[b] + M[c]$$

The next transformation breaks the program into small fragments, each one corresponding to an instruction in the desired processor. This stage corresponds to code generation in a conventional compiler. In this case we assume a very simple 1-address processor with a single processor register, r.

$$r := M[b]\,;\;\; r := r + M[c]\,;\;\; M[a] := r \tag{2}$$

This transformation must of course hide the new variable r, but we will ignore such small details here. We now introduce names (i.e. small integer constants) for each of the instruction-shaped fragments. At the same time, we abstract the fragments so that they are parametrised on at most one memory variable. In this example we will call the parametrised fragments *load*, *add*, *store*. We require two additional variables *opcode* and *opnd* to hold the operation code and operand address for the current instruction. As they are not variables in the linear memory, these will eventually become processor registers. Thus, we introduce the following definition to assist us:

$$\begin{array}{rl} \mathit{Microcode} = \text{CASE } \mathit{opcode} \text{ OF} & \\ & \mathit{load} \;\;:\; r := M[\mathit{opnd}] \\ & \mathit{add} \;\;\;:\; r := r + M[\mathit{opnd}] \\ & \mathit{store} : M[\mathit{opnd}] := r \end{array} \tag{3}$$

We can now translate each fragment from Program 2 by assigning appropriate values to the *opcode* and *opnd* variables with a simultaneous assignment. Executing *Microcode* after each such assignment will then have exactly the same effect as the original fragment. Thus, we get:

$$\begin{array}{lll} \mathit{opcode}, \mathit{opnd} := \mathit{load}, b\,; & \mathit{Microcode}\,; \\ \mathit{opcode}, \mathit{opnd} := \mathit{add}, c\,; & \mathit{Microcode}\,; \\ \mathit{opcode}, \mathit{opnd} := \mathit{store}, a\,; & \mathit{Microcode}\,; \end{array}$$

We now put all the *opcode*, *opnd* instruction specifiers into a second memory array which we'll call *code*. We preload the *code* array with the necessary instructions and then execute the same program as before, but referring to the contents of the *code* array rather than the values themselves. We also take this opportunity to do a simple data refinement to pack the *opcode* and *opnd* values into a single quantity; we use round brackets to represent this packing/unpacking refinement. Thus, the preloading program can be defined as:

$$\mathit{Bootstrap} = \ \mathit{code}[0] := (\mathit{load}, b)\,;\ \mathit{code}[1] := (\mathit{add}, c)\,;\ \mathit{code}[2] := (\mathit{store}, a)\,;$$

The program can now be transformed into:

$$\begin{array}{l} \mathit{Bootstrap}\,; \\ (\mathit{opcode}, \mathit{opnd}) := \mathit{code}[0]\,;\ \mathit{Microcode}\,; \\ (\mathit{opcode}, \mathit{opnd}) := \mathit{code}[1]\,;\ \mathit{Microcode}\,; \\ (\mathit{opcode}, \mathit{opnd}) := \mathit{code}[2]\,;\ \mathit{Microcode}\,; \end{array}$$

As we are aiming at an application-specific processor, we can considerably simplify the final system by choosing a particular implementation for the first semicolon in the above program. Noting that the code memory is never altered after the first assignments to it, and by programming the code into a Read-Only Memory to hold the code, we can effectively *execute* the bootstrap program in the factory that builds the processor.

The aim behind all these transformations is to make identical each statement in the body of the sequential program. This is so that when mapped into hardware, each statement shares the same processor hardware. We are getting quite close to this ideal, as the only differences between the statements in the program above are the *code* array indices. Since we only have sequential statements here, it is particularly simple to replace the index constants with a variable which is incremented appropriately. Thus, we now introduce an additional instruction pointer variable *iptr*, and get:

$$\begin{array}{l} \mathit{Bootstrap}\,;\ \mathit{iptr} := 0\,; \\ (\mathit{opcode}, \mathit{opnd}), \mathit{iptr} := \mathit{code}[\mathit{iptr}], \mathit{iptr} + 1\,;\ \mathit{Microcode}\,; \\ (\mathit{opcode}, \mathit{opnd}), \mathit{iptr} := \mathit{code}[\mathit{iptr}], \mathit{iptr} + 1\,;\ \mathit{Microcode}\,; \\ (\mathit{opcode}, \mathit{opnd}), \mathit{iptr} := \mathit{code}[\mathit{iptr}], \mathit{iptr} + 1\,;\ \mathit{Microcode}\,; \end{array} \tag{4}$$

We have now nearly achieved our goal. It only remains to use a loop introduction theorem on this repeated pattern of statements to give us the final form of our ISS program:

$$\begin{array}{l} \mathit{Bootstrap}\,;\ \mathit{iptr} := 0\,; \\ \text{WHILE } \mathit{iptr} < 3 \\ \quad (\mathit{opcode}, \mathit{opnd}), \mathit{iptr} := \mathit{code}[\mathit{iptr}], \mathit{iptr} + 1\,;\ \mathit{Microcode} \end{array} \tag{5}$$

Program 5 is now in a form in which the processor loop can be compiled into hardware. The microcode program only appears once and is thus shared by all instructions executed on the processor. If all the transformations shown are proven correct, we would then have a constructive proof of the correctness of the microprocessor implementation.

The worked example above deliberately avoids some of the more complex aspects of this transformation, namely the transformation of programs which incorporate parallelism, conditionals and loops. However, transformations do exist for each of these and the reader is referred to associated work in [9], [5], [7], [8], and [2].

6.1 A Dynamic Bootstrap

If we are likely to use the processor for running more than one program, it may be advisable to implement the code memory as random access memory so that it can contain different programs at different times. In the same spirit as the development above, we can further transform the *Bootstrap* program, by turning it into a parallel program. We introduce a bootstrap channel b, and pass the code values over it, preceded by a count value, so that the bootstrap can terminate immediately after reading the last value. By choosing to send the values in reverse order and by identifying the bootstrap counter variable with $iptr$, we can also drop the initialisation of $iptr$ in program 4.

$$\begin{array}{ll} & (b\,!\,2;\quad b\,!\,(store, a));\quad b\,!\,(add, c);\quad b\,!\,(load, b) \\ \| & (b?iptr;\quad \text{WHILE } iptr > 0 \quad DO \quad (b?code[iptr]\,;\ iptr\ :=\ iptr - 1)) \end{array}$$

Now that we have split the bootstrap into two parallel processes, we can choose to implement each process by different mechanisms. Typically the program would be transformed to put the transmitting process in parallel with the sequential composition of the receiving process and the main ISS loop. At this point the bootstrap receiver and ISS loop can be compiled into hardware and it is left open how to implement the transmitting process. It could be an EPROM and a sequencer, or in our case it is usually a host transputer.

To put the bootstrap transmitter in parallel with the rest of the processor needs the transformation $(A||B); C = A||(B; C)$, which is clearly not true in all cases. The mathematics necessary to derive the conditions for this equation are a slight generalisation of [6], so that the termination event ($\surd$) may be in the alphabet of one component of a parallel composition without being in the alphabet of the other. Termination of a parallel construct is determined precisely by the termination of all events that have $\surd$ in their alphabets. In this case, the conditions we need are

(i) $\surd \notin \alpha A$, (ii) $\alpha A \cap \sigma C = \emptyset$, and (iii) $\alpha A \subseteq \alpha B = \alpha C$,

where σC is the set of events that can be performed in some execution by C, a subset of its alphabet. With these conditions, the parallel combination $A||B$ terminates whenever B does, interrupting the execution of A at that point. We will naturally engineer things so that the bootstrap transmitter, A, is quiescent at this point.

7 Conclusions

We have shown in general terms how an application program can be transformed into a microprocessor-based version of the same program. Such transformations can have beneficial effects when compiling into hardware any programs which could use a significant degree of sequential sharing of expensive hardware resources. We have shown that a wide range of parametrisations is possible to an abstract description of a processor so that it can be tailored for a particular application program. The whole process is fast and automatic even down to implementing the microprocessors via FPGA technology. At the quickest, we have taken a simple application program into hardware as a 10 MIPS processor-based version of the same in under five minutes.

We have also demonstrated the basis of a method that could be used to prove the correctness of a microprocessor-based version of a program. Even if not taken quite this far, the basis of the processor transformation in a set of correctness-preserving laws can give a high degree of confidence in the resulting designs.

In the future, as well as extending the range of parameters, we intend to collect together the various experiments we have made in processor synthesis into a library of abstract processor architectures. Each architecture will be paired with its appropriate compiler. If the user does not know what architectural style is appropriate, it should be possible to compile automatically into all the available architectures, perform a time/space analysis of the resulting processors and select the one which best fits a set of user-provided constraints. The development of such constraint-based compilation techniques is likely to be a challenging activity.

If further details are required, the author can be contacted at the address given or by email as Ian.Page@comlab.oxford.ac.uk. We also have a number of related publications available by anonymous ftp at ftp.comlab.ox.ac.uk in the directory Documents/techpapers/Ian.Page

8 Acknowledgements

Thanks are due to ESPRIT OMI/MAP and OMI/HORN programmes for supporting our work. My debt to Duncan Greatwood and Robin Watts for their tireless efforts on their projects in implementing and improving the concepts of parametrised processors is gratefully acknowledged. I'd also like to thank Steve Schneider for proving the bootstrap transformation. Finally, thanks are due to all the other members of the Hardware Compilation Research Group who make working here such a pleasure.

References

[1] Acorn. *RISC OS Programmers Reference Manuals.* Acorn Computers Ltd., 1989.

[2] J.P. Bowen, He Jifeng, and P.K. Pandya. An approach to verifiable compiling specification and prototyping. In P. Deransart and J. Małuszyński, editors, *Programming Language Implementation and Logic Programming (PLILP'90)*, volume 456 of *Lecture Notes in Computer Science*, pages 45–59. Springer–Verlag, 1990.

[3] Barry Fagin and Pichet Chintrakulchai. Prototyping the dlx microprocessor. In *Proc. IEEE Workshop on Rapid System Prototyping*, pages 60–67, 1992.

[4] Duncan Greatwood. Parametrizable processor generation on field programmable gate arrays. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, U.K., 1992.

[5] He Jifeng, I. Page, and J.P. Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods, Proc. IFIP WG10.2 Advanced Research Working Conference, CHARME'93*, volume 683 of *Lecture Notes in Computer Science*, pages 214–225. Springer-Verlag, 1993.

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

[7] C.A.R. Hoare. Refinement algebra proves correctness of compiling specifications. In C.C. Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, Workshops in Computer Science, pages 33–48. Springer–Verlag, 1991.

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

[9] C.A.R. Hoare and I. Page. Hardware and software : The closing gap. *Transputer Communications*, 1234:XX–XX, May - perhaps 1994.

[10] Ian Page and Wayne Luk. Compiling occam into FPGAs. In *FPGAs*, pages 271–283. Abingdon EE&CS Books, 1991.

[11] David Lewis, Marcus Ierssel, and Daniel Wong. A field programmable accelerator for compiled-code applications. In D.A. Buell and K.L. Pocek, editors, *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 60–67, 1993.

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

[13] Inmos Limited. *The T9000 Transputer Instruction Set Manual.* Inmos Limited, 1993.

[14] Wayne Luk, Vincent Lok, and Ian Page. Hardware acceleration of divide-and-conquer paradigms: a case study. In D.A. Buell and K.L. Pocek, editors, *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 192–201, 1993.

[15] David May and Catherine Keane. Compiling occam into silicon. In *Communicating Process Architecture*. Prentice Hall and Inmos, 1988.

[16] David Patterson and John Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, Ca., 1990. DLX microprocessor spec'n.

[17] Laurence Paulson. *ML for the Working Programmer.* CUP, 1991.

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

[19] Robin Watts. Applications of field programmable gate arrays. Undergraduate project thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, U.K., 1993.

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