The role of occam in the design of the IMS T800

INMOS Technical Note 47

David Shepherd

September 1988
72-TCH-047

Contents

Presented at the 20th Workshop on Microprogramming, December 1-4, 1987, Colorado Springs, CO. To appear in ACM SIGMICRO Newsletter, 1988.

Abstract

This paper describes how 'correct' microcode can be produced through the use of mathematical logic and formal design methods. The use of these techniques to derive correct microcode for the IMS T800 floating point transputer from a mathematical specification is discussed. This experience on the IMS T800 has shown that this approach provides the opportunity to produce designs with a higher certainty of correctness in significantly less time as compared with 'traditional' design techniques. These techniques are currently being applied to the construction of correct specifications at the hardware description language level. This work is attempting to incorporate mathematical logic and formal design methods into the INMOS CAD system so that their use becomes the standard way of producing correct VLSI devices.

1 Introduction

Recent research has demonstrated the possibilities of producing hardware designs that have been verified as opposed to tested. Examples of this approach include the proof of correctness of a simple microcoded processor [1] and the verification of the design of various low-level hardware modules [2]. The tools that have been used in this work are LCF_LSM [3], VERITAS [4] and HOL [5].

Most people would agree that it is desirable for a manufacturer's products to meet some form of specification. This requirement becomes vital when the product is used in a life-critical situation - users must know what the behaviour of the product will be. This has resulted in the emergence of a disciplined approach to design in many engineering professions. An architect checks that a new building will not fall down, an aircraft designer does detailed calculations to ensure that the wings produce enough lift. At each step of the construction process checks are made to ensure that the components used meet their specifications in the design.

Now that computers are being used in life-critical applications, such as fly-by-wire aircraft or complex life support systems, it is vital for the underlying hardware to be correct. It is impossible to exhaustively test components as simple as a 32-bit multiplier - never mind an entire processor - so different techniques must be used to verify designs. As E.W. Dijkstra has remarked [6]

(non-exhaustive) testing can be used to show the presence of bugs
but never to show, their absence.

Starting from an agreed formal specification a correct design can be produced if the implementation is produced by a sequence of provably correct steps. This will bring the standard of computer design to the levels expected in other branches of engineering [7]. Use of verified design methods can produce savings in time and expenditure. The need to redesign part of a VLSI device may cause a 2 or 3 month delay in its launch and several such iterations can make a device obsolete before it comes to market.

This technical note details how a verified design approach was used on sections of the IMS T800 floating-point unit microcode. The formal semantics of the occam language [8, 9] and the use of program transformations are described. Then a simple example is used to show haw a high-level specification can be developed into microcode using formal design methods that guarantee the correctness of the final design.

2 Occam

The occam language [8] allows a system to be hierarchically decomposed into a collection of concurrent processes communicating via channels. This allows it to be used to represent the behaviour of a VLSI device in a very natural way - the various top-level modules can be mapped on to individual processes with their interfacing handled by channel communication. In more traditional languages the inherent parallelism of a VLSI device has to handled by explicit programming. occam has a very efficient implementation permitting fast execution of such a behavioural description to allow for simulation. Most importantly, for the purposes of this paper, occam has rich formal semantics [9] which facilitate program transformation and proof.

2.1 Occam transformations

The algebraic semantics of occam given in [9] consists of a set of laws which define the language constructs. The algebraic semantics have been shown to be consistent with the denotational semantics establishing the validity of these laws. These transformation laws enable a normal form for finite occam programs to be defined.

A transformation law can be used to transform one program into another whose observable behaviour is equivalent. Many transformation laws are 'obviously true' and are regularly used by programmers - for example sequential composition of processes is associative:

```
SEQ            SEQ
  P              SEQ
  SEQ      =       P
    Q              Q
    R            R
```

This is the law SEQ binassoc. Others are more complex and include preconditions for validity but, with a bit of effort, can be seen to be true.

If a sequence of transformations can be found to transform one program into another then the two programs are known to be equivalent. If, in addition, one of these programs is known to be a correct implementation of a specification then the correctness of the other can be inferred.

Using these techniques it is possible to demonstrate the correctness of implementations by transformation - doing this by experimental testing takes far too long for problems like floating-point arithmetic.

An example transformation

As an example consider the following program fragment:

```
SEQ
  X := A
  Y := Y + X
```

These two assignment statements can be merged into one multiple assignment statement.
First the law AS id is used to add an identity assignment to each statement:

AS id $\qquad \underline{x}, \underline{y} := \underline{e}, \underline{y} \equiv \underline{x} := \underline{e}$

giving the program:

```
SEQ
  X,Y := A,Y
  Y,X := Y + X,X
```

Next the law AS perm is applied to the second statement:

AS perm $\qquad < x_i | i = 1..n >:=< e_i | i = 1..n >$
$$\equiv$$
$< x_{\pi_i} | i = 1..n >:=< e_{\pi_i} | i = 1..n >$
for any permutation π of $\{1..n\}$

giving:

```
SEQ
  X,Y := A,Y
  X,Y := X,Y + X
```

Finally these two statements are merged by the law SEQ comb:

SEQ comb $\qquad SEQ(\underline{x} := \underline{e}, \underline{x} := \underline{f}) \equiv \underline{x} := \underline{f}[\underline{e}/\underline{x}]$

giving:

```
X,Y := A,Y + A
```

2.2 The occam transformation system

To aid the process of transforming programs a simple interactive transformation system has been implemented in the language ML [10]. A program can be parsed into this system and then manipulated by the user. All the basic laws in [9] are implemented inside the system along with some extra ones - the system is extensible and new laws (that have been proven correct) can be coded and added if required. Regularly executed sequences of transformations can be coded as ML functions giving higher-level transformations. The example transformation shown above has been coded up as the transformation law combos which itself is used in more powerful transformations. The basic transformations often have only a small localised effect but when suitably combined they can perform significant transformations which being constructed from correct component transformations are known to be correct.

The transformation system user can select which transformation laws to apply and examine the effects of these transformations. The fact that the transformation system is being used provides the verification of the equivalence between the initial program and the transformed end result - but if necessary it would be feasible to produce the list of transformations which constitute the proof.

3 Instruction development

The instruction development process consists of specifying the operation of the instruction in the Z specification language [11]. Since Z is a mathematically based language it allows precise unambiguous statements about operations to be made concisely and - if used in a sympathetic manner - clearly.

Along with the specifications of the instructions there will be a set of specifications of system constants, system state and other global features of the design. In the case of the IMS T800 floating-point unit this consists of a formal specification of the IEEE floating-point standard - such as in [12], a specification of the internal representation of floating-point numbers in registers, a specification of the floating-point unit state - i.e. the registers and flags, and definitions of various constants that are used. This corresponds to formally describing the overall architecture.

Each instruction specification is refined into a high-level occam implementation. This can involve going via a guarded command language using pre- and post-conditions as in [13]. This high-level implementation is often the sort of implementation that a competent programmer would produce from the specification but the formal derivation ensures that no mistakes are made.

The occam program is then transformed inside the transformation system into a form equivalent to the microcode assembler source. The steps in this process are motivated by the functions available in the microcode machine. This involves:

1. refining /F conditions into the conditions available on the microcode machine
2. refining the expressions so that they use the alu and bus operations available on the microcode machine
3. refining the sequential control of the program into a form that simulates the microinstruction control in the microcode machine

The various stages of simple development used as an example are shown in the next section.

4 An example instruction development

The following example demonstrates the methods that have been found to be useful in the IMS T800 design. This example takes a high-level specification in the Z specification language [11] and refines it in a sequence of steps into a microcoded implementation that will run on a microcode machine similar to the IMS T800 floating point unit. For brevity certain simplifications have been made - notably that infinities, Not-a-Numbers and denormalised numbers are ignored.

4.1 Preliminary definitions

Before any instructions are specified and implemented it is necessary to make a few preliminary definitions. There is a need to specify the format of registers, various constants and methods for interpreting data. This is a formalisation of the top level of architectural description of the device. Only the subset of definitions relevant to this example will be given.

The definition of the real format will contain the specification of the number of bits in the fractional part of a floating-point number and the exponent bias:

$$\boxed{bitsinfrac, bias : \mathbf{N}}$$

Now the floating-point register format can be specified:

Floating_Point_Register

$$\begin{array}{|l|}
\hline
frac, exp : \mathbf{N} \\
sign : \{-1, +1\} \\
\hline
(exp = 0 \land frac = 0) \\
\qquad \lor \\
(2^{bitsinfrac-1} \leq frac < 2^{bitsinfrac}) \\
\hline
\end{array}$$

This states that a Floating_Point_Registers has three fields. Two of which, frac and exp, are positive integers and the third, sign, is either -1 or +1. The predicate states that both the exponent and fraction are 0 or that frac is between $2^{bitsinfrac-1}$ and $2^{bitsinfrac}$ – this ensures that the fraction is normalised.

The valuation function on a floating-point register fv establishes the link between a Floating_Point_Register and the value it 'holds':

$$\begin{array}{|l|}
\hline
fv : Floating\_Point\_Register \mapsto \mathbf{R} \\
\hline
\forall x : Floating\_Point\_Register. \\
fv(x) = x.sign \times (x.frac \times 2^{1-bitsinfrac}) \times 2^{exp-bias} \\
\hline
\end{array}$$

Two constants are used to represent the largest and smallest integers in the integer format. As the IMS T800 uses 32-bit 2s complement integers these are specified by:

$$\begin{array}{|l|}
\hline
MinInt, MaxInt : \mathbf{Z} \\
\hline
MinInt = -2^{31} \\
MaxInt = 2^{31} - 1 \\
\hline
\end{array}$$

4.2 The instruction specification

The instruction under consideration here is a component of the real to integer conversion instruction sequence. It checks that the value of Areg lies within integer range - if it doesn't then the error flag must be set to indicate a conversion error.

The Z specification of this instruction is very simple:

Floating_Check_Integer_Range

$$\begin{array}{|l|}
\hline
Areg, Areg' : Floating\_Point\_Register \\
Error\_Flag, Error\_Flag' : bool \\
\hline
fvAreg \in \mathbf{Z} \\
Areg' = Areg \\
fvAreg \in [MinInt, MaxInt] \Rightarrow Error\_Flag' = Error\_Flag \\
fvAreg \notin [MinInt, MaxInt] \Rightarrow Error\_Flag' = true \\
\hline
\end{array}$$

The first predicate is a precondition to this operation. If 1vAreg is not an integer then the effect of this operation will be undefined. In this way the precise conditions for the correct execution of an operation are stated. This instruction is intended for use in a particular sequence of instructions and the previous instruction will have established this precondition.

It is easy to see that this specification satisfies the requirements for the instruction. Once this has been agreed to be 'correct' the development process will ensure that the final implementation will also satisfy the requirements.

4.3 Refining to procedural form

A refinement of a specification can consist of either refining a data type or decomposing the procedural form. As the major data type - reals - has already been refined into its machine representation, by using Floating_Point_Register and the abstraction function Iv, the specification can be decomposed into procedural form. The specification can be easily implemented by:

```
if
    $fv(Areg) \in [MinInt, MaxInt] \rightarrow skip$
    $fv(Areg) \notin [MinInt, MaxInt] \rightarrow Error\_Flag := true$
fi
```

Using the pre/post-condition laws in [13] this can be shown to implement the Z specification.

4.4 Refining to occam

This has produced a procedural implementation but the conditionals used in the if .. fi construct are not available in occam so they need to be refined into equivalent occam expressions.

To do this the lemmas about integer range shown below will be useful.

lemma 1 $\vdash \forall x, y : Floating\_Point\_Register.$
$(x.exp < y.exp \lor (x.frac < y.frac \land x.exp = y.exp))$
$\Leftrightarrow |fv(x)| < |fv(y)|$

lemma 2 $\vdash \forall x : Floating\_Point\_Register.$
$fv(x) = MinInt$
$\Leftrightarrow (xsign = -1 \land x.frac = MSBit \land x.exp = LargestINTExp)$

lemma 3 $\vdash MaxInt = -(MinInt + 1)$
where MSBit $= 2^{bitsinfrac-1}$
$LargestINTExp = 32 + bias$

From lemmas 1 and 2 obtain:

$\vdash \forall x : Floating\_Point\_Register.$
$x.exp < LargestINTExp$
$\Leftrightarrow |fv(x)| < |MinInt|$

The fact that $MSBit \leq x.frac$ is part of the invariant of Floating_Point_Register is used to eliminate the disjunct where $x.exp = LargestINTExp$.

Now using lemma 3 and adding an extra condition obtain:

$\vdash \forall x : Floating\_Point\_Register.$
$fv(x) \in \mathbf{Z} \Rightarrow x.exp < LargestINTExp$
$\Leftrightarrow |fv(x)| \leq |MaxInt|$

From these obtain:

$\vdash \forall x : Floating\_Point\_Register.$
$fv(x) \in \mathbf{Z} \Rightarrow fv(x) \in [MinInt, MaxInt]$
$\Leftrightarrow (x.exp < LargestINTExp \lor fv(x) = MinInt)$

4.5 High-level occam implementation

The previous section allows the high-level occam implementation below to be derived.

```
IF
  (Areg.Exp < LargestINTExp) OR
    ((Areg.Sign = 1) AND
     (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit))
    SKIP
  NOT ((Areg.Exp < LargestINTExp) OR
         ((Areg.Sign = 1) AND
          (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit)))
    ErrorFlag := TRUE
```

Using two laws IF pri and IF or-disc:

IF pri $IF(b_1P_1, ..., b_nP_n)$
$\equiv IF(b_1^*P_1, ..., b_n^*P_n)$
where $b_1^* = \not b_1 \wedge ... \wedge \not b_{i-1} \wedge b_i$

IF or-dist $IF(b_1P, b_2P, \underline{C})$
$\equiv IF(b_1 \vee b_2P, \underline{C})$

this can be simplified to the program:

```
IF
  (Areg.Exp < LargestINTExp)
    SKIP
  (Areg.Sign = 1) AND
     (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit)
    SKIP
  TRUE
    ErrorFlag := TRUE
```

which is probably the implementation of the specification that a competent programmer would produce but the 'special' case of MinInt is frequently omitted.

4.6 Transformations towards microcode

The previous sections have developed an occam program that correctly implements the specification. This can now be transformed into an equivalent form that corresponds to microcode assembler source. Full details of this process will not be given here.

Each step consists of transforming one aspect of the program towards the form used in the microcode machine. Ideally this occam program would be transformed into the final program. As the transformation system is still under development most of the laws that it contains are those that are 'general' - i.e. are correct in all environments. This does not allow the required transformation to be performed in a forwards manner. Instead at each step a proposed implementation was constructed and this was then verified by transforming it back into the current 'correct' implementation.

Refining the conditionals

The occam program given contains a three-way IF statement with the conditionals:

```
1   (Areg.Exp < LargestINTExp)
```

```
2   (Areg.Sign = 1) AND
      (Areg.Exp = LargestINTExp)
      AND (Areg.Frac = MSBit)
3   TRUE
```

The structure of the program must be transformed to take account of the conditional signals available on the microcode machine - i.e. that conditionals are available to signal that the result of an ALU operation is less than 0 or that the result of an ALU subtraction is 0 etc.

This program for implementation with refined conditionals is shown below. The various laws for IF constructs in [9] enable this to be verified:

```
IF
  (Areg.Sign = 1)
    IF
      ((Areg.Exp - LargestINTExp) < 0)
        SKIP
      NOT ((Areg.Exp - LargestINTExp) < 0)
        IF
          ((Areg.Exp - LargestINTExp) = 0)
            IF
              ((MSBit - Areg.Frac) = 0)
                SKIP
              NOT ((MSBit - Areg.Frac) = 0)
                ErrorFlag := TRUE
          NOT ((Areg.Exp - LargestINTExp) = 0)
            ErrorFlag := TRUE
  NOT (Areg.Sign = 1)
    IF
      ((Areg.Exp - LargestINTExp) < 0)
        SKIP
      NOT ((Areg.Exp - LargestINTExp) < 0)
        ErrorFlag := TRUE
```

Refining the expressions

The previous section has produced conditionals that are available in the microcode machine. The next step is to take account of how the expressions producing these conditionals are evaluated. This stage involves introducing variables to represent the various buses and conditional flags. The conditional flags appear as the IF conditionals and are evaluated in terms of the results of the ALU operations before the IF statement.

This program for implementation with refined expressions is shown below: The laws for SEQ, VAR and assignment in [9] verify this step:

```
VAR AregNegative, ExpZbus, ExpZbusNeg, ExpZbusEgZ, FracZbusEgZ :
VAR FracZbus :
SEQ
  AregNegative := (Areg.Sign = 1)
  ExpZbus := (Areg.Exp - LargestINTExp)
  ExpZbusNeg := ExpZbus < 0
  IF
    AregNegative
      IF
        ExpZbusNeg
          SKIP
        NOT ExpZbusNeg
          SEQ
            ExpZbus := (Areg.Exp - LargestINTExp)
            FracZbus := (MSBit - Areg.Frac)
            ExpZbusEgZ := ExpZbus = 0
            IF
              ExpZbusEgZ
                SEQ
                  FracZbusEgZ := FracZbus = 0
                  IF
                    FracZbusEgZ
                      SKIP
                    NOT FracZbusEgZ
                      ErrorFlag := TRUE
              NOT ExpZbusEgZ
                ErrorFlag := TRUE
    NOT AregNegative
      IF
        ExpZbusNeg
          SKIP
        NOT ExpZbusNeg
          ErrorFlag := TRUE
```

Introducing sequencing

The program now contains expressions and conditionals that can be formed in the microcode machine. However, the program does not define microwords. The final step is to mimic the microsequencing in the microcode machine by use of a variable as a microprogram counter and a WHILE loop containing an IF microinstruction selector. Each branch of the IF statement contains the 'code' for one microinstruction - i.e. it can have one fractional ALU operation, one exponential ALU operation and defines the next microinstruction to execute - possibly with one or two conditionals.

The laws for WHILE and IF allow this program to be 'unwound' back into its previous form.

4.7 Translation to microcode

The final program for low level occam implemenation from the previous transformations is:

```
VAR NextInst :
VAR AregNegative, ExpZbusNeg, ExpZbusEgZ, FracZbusEgZ :
VAR FracZbus, ExpZbus :
SEQ
  NextInst := FloatingPointCheckIntegerRange
  WHILE NextInst <> NOWHERE
    IF
      NextInst = FloatingPointCheckIntegerRange
        SEQ
          AregNegative := (Areg.Sign = 1)
          ExpZbus := (Areg.Exp - LargestINTExp)
          ExpZbusNeg := ExpZbus < 0
          IF
            AregNegative
              IF
                ExpZbusNeg
                  NextInst := NOWHERE
                NOT ExpZbusNeg
                  NextInst := CheckMinInt
            NOT AregNegative
              IF
                ExpZbusNeg
                  NextInst := NOWHERE
                NOT ExpZbusNeg
                  NextInst := OutofRange
      NextInst = OutofRange
        SEQ
          ErrorFlag := TRUE
          NextInst := NOWHERE
      ... negative case micro instructions
```

This corresponds in an almost one-to-one manner with the source format for the microcode assembler. A pattern-matching program is used to translate the stylised occam of the above program into the source for the microcode assembler. The microcode assembler then produces the definition of the microcode ROM from this source.

4.8 Microcode assembler source

Finally the microcode can be derived:

```
FloatingPointCheckIntegerRange:

  ExpConstantFromLargestINTExp
  ExpXbusFromAreg                ExpYbusFromConstant
  ExpZbusFromXbusMinusYbus
  GOTO Cond1FromAregSign -> (Cond0FromExpZbusNeg -> (NOWHERE, CheckMinInt),
                            Cond0FromExpZbusNeg -> (NOWHERE, OutofRange))

CheckMinInt:
  ExpConstantFromLargestINTExp
  ExpXbusFromAreg                ExpYbusFromConstant
  ExpZbusFromXbusMinusYbus
  FracXbusFromMSBit              FracYbusFromAreg
  FracZbusFromXbusMinusYbus
  GOTO Cond1FromExpZbusEgZ -> (CheckMinInt2, OutofRange)

CheckMinInt2:
  GOTO Cond1FromFracZbusEgZ -> (NOWHERE, OutofRange)

OutofRange:
  SetErrorFlag
  GOTO NOWHERE
```

This process has ensured that the 'program' in the microcode ROM correctly implements the initial specification. It might seem possible to do this informally in this simple case which only produces four microwords. Other instructions contain up to ninety microwords where informal development can easily introduce subtle bugs. The ability to verify an implementation using program transformations has proved invaluable.

5 Current and future work

Work on the IMS T800 has shown how correct microcode can be derived from a high-level specification. However, this has assumed that the hardware implementing the microcode machine is correct. To produce a verified processor design it will be necessary to apply the same degree of rigour to the design of the microcode machine. This necessitates refining the specifications of microfunctions into hardware description language (HDL) implementations. The INMOS CAD system already ensures that silicon layout is equivalent to its HDL specification.

This correctness of design can be achieved by defining axioms for the behaviour of low-level modules in the HDL module library if necessary down to transistor level. Larger modules and circuits can then be specified in terms of compositions of these 'axiomatic' modules. Then a logic tool, such as HOL [5], can be used to derive the behaviour of the design. Checking this against an original specification enables the correctness - or otherwise - of the design to be established.

6 Conclusions

Work at INMOS using the transformation system and a formal design strategy has been seen to be of benefit. The correctness of the microcode for the IMS T800 floating-point unit was established in far less time than would be needed by an 'adequate' amount of testing. In addition, any amount of non-exhaustive testing leaves the possibility that certain erroneous operations have not been exercised. This has enabled INMOS to produce the IMS T800 well ahead of schedule with a high degree of confidence in the correctness of the microcode - this would not have been possible by other design methods.

Work is now in progress to incorporate this formal design strategy into the other levels of the design process to maintain the correctness of a complete design. It seems clear that the CAD system will need to incorporate a theorem prover and work is progressing at INMOS to ensure that this is the case.

References

[1] Proving a computer correct, M Gordon, University of Cambridge Computer Laboratory, Technical Report 42, 1983.

[2] Specification and Verification using Higher-Order Logic, F K Hanna, N Daeche, Proceedings of the 7th International Conference on Computer Hardware Design Languages. Tokyo, 1985.

[3] LCF-LSM, M Gordon, University of Cambridge Computer Laboratory, Technical Report 41

[4] The VERITAS theorem Prover, F K Hanna, N Daeche, Electronics Laboratory, University of Kent at Canturbury, 1984 onwards.

[5] HOL: A machine orientated formulation of Higher-Order Logic, M Gordon, University of Cambridge Computer Laboratory, Technical Report 68, 1985.

[6] Dijkstra, E.W., quote taken from 7

[7] Programming is an engineering profession, C A R Hoare, Oxford University Computing PRO, Technical Monograph PRG-27, 1982.

[8] The occam Programming Manual, INMOS Ltd, Prentice Hall, 1984.

[9] The laws of occam programming, A W Roscoe, C A R Hoare. Oxford University Computing PRG, Technical Monograph PRG-53, 1986.

[10] Edinburgh LCF - chapter 2, M Gordon, R Milner, C Wadsworcn, LCNS 78, Springer Verlag,1979.

[11] The Z Handbook, B A Sufrin (editor), Oxford University Computing PRO, 1986.

[12] Formal methods applied to a floating point nurnoer system, G Barren, Oxford University Computing PRG, Technical Monograph, 1987.

[13] The science of programming, D Dries, Springer-Verlag, 1981.