Use of Formal Methods by a Silicon Manufacturer

David May
Inmos Limited

March 17, 1988

1 Introduction

A VLSI semiconductor manufacturer faces the problem of specifying very complex devices - and of ensuring that the implementation of the specification is correct. Exhaustive testing of the device may be impossible because the number of cases to be considered is too large, or impractical because of the complexity of an adequate test environment. Yet VLSI devices are employed as components of high volume products where the cost of correcting a fault is high - and in safety critical environments where a fault may endanger human life. Indeed, to a semiconductor manufacturer, each design fault in a VLSI device normally incurs substantial cost and - worse still in such a competitive market - delays introduction of the product.

In an attempt to tackle these problems, INMOS has recently started to employ formal techniques of specification, transformation and proof in the design of microcomputer products. This lecture will discuss the use of such techniques in the semiconductor industry and describe the process by which recent work at Oxford university on the formal semantics of occam was employed by INMOS in the design of the recently announced T800 floating point transputer [10].

2 VLSI design techniques

Current VLSI designs are far too large and complex to be designed by hand. For many years, computers have been used to maintain the design database and to allow simulation of the design. More recently, computers have been used to provide various forms of design checking.

For example, circuit connectivity checking was until recently performed manually by comparing the circuit diagrams with a large scale drawing of the silicon layout. These drawings became so large that designers had to crawl over them following the tracks with coloured pencils! Even for a design with only 10,000 components this is a very tedious, time-consuming and error-prone task. Automatic connectivity checking has become *essential* in VLSI design systems, and ensures that the description of a device in a hardware description language (HDL), the circuit diagram and the actual silicon layout all express the same connectivity. Similarly, automatic checking of silicon design rules - such as ensuring the two metal tracks are not too close together - is now performed automatically.

In designing a A VLSI device, it is useful to have a behavioural description of what the device does, and a hardware description of the components of the device and the way in which they are interconnected.

2.1 The INMOS CAD system

The computer aided design system developed by INMOS for hardware design has many features that enable "correct" designs to be made. The system runs interactively with a designer being able to examine portions of the silicon design on a colour display and to lay down new tracks. The system immediately checks for design rule violation and will prevent designs that contravene design rules - so assuming that the design rules reflect the capabilities of the fabrication process then the actual chip should reflect the design made in the design system. The system supports multi-user access to the same design database with record locking used to prevent two people modifying the same data.

The silicon design is specified in the INMOS hardware description language (HDL). This is a hierarchical

language which allows the design to be specified using a top down modular approach. It allows modules such as register cells and latches to be defined in terms of silicon layout. More complex modules such as registers and ALUs can then be defined in terms of the simpler modules. The hardware description language is used to express the structure of the design, but not its behaviour.

The silicon design is checked for electrical equivalence to this HDL specification. This should guarantee that the finished chip is a correct implementation of the HDL specification.

2.2 The module library

The HDL is a hierachical module based language. As designs are produced a library of modules is generated. If the same HDL specification is required in another design then the silicon design that already exists can be reused. This greatly reduces design time and the possibility of introducing errors as the module design in the library will already have been checked - and possibly will have been demonstrated in an existing product.

The floating point unit on the IMS T800 was designed as a seperate microcoded processor. This enabled major parts of the logic of the IMS T414 to be reused. In addition to the main processor on the IMS T800 (which had only minor modifications from the IMS T414), the microcode decode logic and much of the arithmetic unit logic of the floating point unit could be taken from the module library.

2.3 Behavioral Description

Behavioural description languages have been used to design sequential processors for many years. As the process of interpreting instructions in a sequential computer is (nearly) sequential, a conventional sequential programming language can be used to write the behavioural description of a processor. However, VLSI devices consist of many interacting components, and their behaviour can therefore only be expressed in a language with concurrency and communication.

The obvious difficulty is that there is no way of checking automatically that the hardware description of a device actually implements the behavioral description. This problem is the subject of several current research projects. Examples include the proof of correctness of a simple microcoded processor [3] and the verification of the design of various low level hardware modules [8]. The tools that have been used in this work are LCF_LSM [4], VERITAS [9] and HOL [5].

This work has made significant progress as a result of chosing design problems simple enough to be proved correct. A microprocessor manufacturer cannot simplify the problem in this way - to remain competitive in the world market it is necessary to design products of increasing function and performance. Only small specialised markets exist at present for a 'proven' microprocessor - though this situation may change in the future.

In future it may be possible to have associated with modules in the library sets of axioms that define their behaviour. This would be used in a proof system to formally derive the behaviour of compositions of modules. These techniques should give more knowledge of the behavior of a set of modules than simulation would and may be more efficient in time and resources.

2.4 occam

The occam language [11] 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 onto individual processes with their interfacing handled by channel communication. occam has a very efficient implementation allowing fast execution of such a behavioural description to allow for simulation. Most importantly for the purposes of this paper occam has rich formal semantics [12] which facilitates program transformation and proof.

occam was designed by INMOS as a language for describing and programming concurrent systems. It is the programming language for the INMOS transputer products. However, occam has also been used extensively within INMOS as a behavioral description language. The development of the transputer architecture was performed using an occam simulation in which the component parts of the transputer are represented as occam processes.

For the first transputer products, the implementation of this occam behavioral description in HDL was performed informally and logic simulations used (as far as possible) to check the correctness of the design.

3 The first proof

Difficulties of the informal method of testing became apparent in several areas of design. Many of the algorithms used in computer arithmetic are intricate - and cannot be exhaustively tested. Floating point arithmetic is particularly intricate and many computers have faulty implementations of basic arithmetic operations. Often these give rise to minor errors in rounding and are not noticed by users.

In the first 32 bit transputer, floating point arithemetic was supported primarily by a software package written in occam. A few special instructions were added to the processor to speed up the package. Attempts were made to validate the package and the special instructions. The normal validadtion process for floating point arithmetic is to execute a large number of test cases. In fact, the normal tests employ four different bit-patterns each of which is used in every mantissa bit position. For each mantissa values, all possible exponent values and both signs are used. In 32 bit arithmetic, this gives rise to about 50,000 values for each operand. There are 5 operations and four rounding modes, and it is therefore necessary to execute 50,000,000,000 floating point operations - this takes about 15 hours if each operation takes 1 microsecond.

In practice, it is quite impossibly to perform the tests at this speed. An increasing amount of equipment was allocated to allow testing of several cases to continue in parallel. However, a new problem now emerged. The tests were comparing a suspect implementation of arithmetic with a 'reference implementation' - a widely used VLSI device. This device had been chosen because of its relatively simple implementation. However, in view of the problems of validation, there seemed to be reason to question whether a reference implementation for floating point arithmetic exists at all. In fact, a bug was found in this device - certain operations gave rise to errors caused by 'double rounding'. Instead of rounding a result directly into its final reporesentation, it is first rounded to an internal representation and then rounded again to the final representation.

In many respects, the occam floating point package was an ideal candidate for a formal correctness proof. It is not very long, it is intricate and a great deal depends on its being correct. The idea of constructing a proof arose during a lecture given at by D. Good [2] in which it was stated that constucting a verified program in the Gypsy system took "about 5 times longer than the normal (informal) way". Within a short time, Geoff Barrett of the Oxford University programming research group was persuaded to try to construct a proof - if successful, this would demonstrate the use of formal methods in a very important practical application.

One of the immediate problems was that IEEE standard 754 is expressed in English. The first problem was, therefore, to rewrite the standard in a formal notation and for this purpose Z was used. The package was then derived by standard techniques refining the Z specification into an occam program. This work was completed in three months *and overtook the experimental validation.*

4 Validating Hardware

Whilst the validation of the software package was proceeding, work started on the design of a transputer incorporating floating point hardware. Immediately, the problem of verification arose.

A semiconductor manufacturer cannot afford to construct prototypes. They are too expensive, and take too long to design and manufacture. For this reason, the industry relies heavily on validating simulations.

Unfortunately, a simulation is very slow - at least 100 times slower than the real device. In practice, it will normally be at least 1000 times slower as the simulation is executed using technology about 4 years behind the device it is simulating. The result is that to validate the design by testing would take over 1 year.

In fact, it is normal to use a relatively low level simulator which is much slower than an ideal one. Typically, the INMOS simulator operates 1,000,000,000 times slower than real time. Although an ideal one could be written, this gives rise to the problem of maintaining consistency between several different levels of simulation as the design progresses.

Clearly, it was important to find a way of formally developing the floating point hardware, possibly starting from the already proven software package. The key to this was turned out to be the occam transformation system, at this stage still under development at Oxford Universtiy.

5 occam transformations

The algebraic semantics of occam given in [12] consists of a set of laws which define the language constructs. The algebraic semantics has 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.

5.1 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 $\quad \underline{x}, \underline{y} := \underline{e}, \underline{y} \equiv \underline{x} := \underline{e}$

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 $\quad < x_i | i = 1..n > :=< e_i | i = 1..n >$

$\equiv$

$< x_{\pi_i} | i = 1..n > :=< e_{\pi_i} | i = 1..n >$
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 $\quad SEQ(\underline{x} := \underline{e}, \underline{x} := \underline{f}) \equiv \underline{x} := \underline{f}[\underline{e}/\underline{x}]$

giving

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

5.2 The occam transformation system

To aid the process of transforming programs a simple interactive transformation system has been implemented in the language ML [6]. A program can be parsed into this system and then manipulated by the user. All the basic laws in [12] 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 *combas* 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.

6 Instruction development

The instruction development process consists of specifying the operation of the instruction in the Z specification language [13]. 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 [1], 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 [7]. 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

- refining *IF* conditions into the conditions available on the microcode machine
- refining the expressions so that they use the alu and bus operations available on the microcode machine
- 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.

7 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 [13] 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.

7.1 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.

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

Now the floating point register format can be specified.

Floating_Point_Register

$$\text{frac, exp} : \mathbf{N}$$
$$\text{sign} : \{-1, +1\}$$

$$(exp = 0 \land frac = 0) \lor (2^{bitsinfrac-1} \leq frac < 2^{bitsinfrac})$$

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".

$$fv : \text{Floating\_Point\_Register} \rightarrow \mathbf{R}$$

$$\forall x : \text{Floating\_Point\_Register}.\ fv(x) = x.sign \times (x.frac \times 2^{1-bitsinfrac}) \times 2^{exp-bias}$$

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

$$\text{MinInt, MaxInt} : \mathbf{Z}$$

$$MinInt = -2^{31}$$
$$MaxInt = 2^{31} - 1$$

7.2 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

Areg, Areg' : Floating_Point_Register
Error_Flag, Error_Flag' : **bool**

---

fv Areg $\in \mathbf{Z}$
Areg' = Areg
fv Areg $\in$ [MinInt, MaxInt] $\Rightarrow$
Error_Flag' = Error_Flag
fv Areg $\notin$ [MinInt, MaxInt] $\Rightarrow$ Error_Flag' = **true**

The first predicate is a precondition to this operation. If fv Areg 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.

7.3 Refining to procedural form

A refinement of a specification can consist of either refining a data type or by decomposing the procedural form. As the major data type – reals – has already been refined into its machine represention, by using *Floating_Point_Register* and the abstraction function fv, 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 [7] this can be shown to implement the Z specification.

7.4 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 in Figure 1 will be useful.

lemma 1 $\vdash \forall x, y$: Floating_Point_Register.
$(x.exp < y.exp \vee (x.frac < y.frac \wedge x.exp = y.exp)) \Leftrightarrow |fv(x)| < |fv(y)|$

lemma 2 $\vdash \forall x$: Floating_Point_Register.
$fv(x) = MinInt \Leftrightarrow (x.sign = -1 \wedge x.frac = MSBit \wedge x.exp = LargestINTExp)$

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

Figure 1: lemmas about integer range

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

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

From these obtain

$$\begin{array}{l}\vdash \forall x : \text{Floating\_Point\_Register}. \\ \quad fv(x) \in \mathbf{Z} \Rightarrow fv(x) \in [\text{MinInt}, \text{MaxInt}] \\ \quad\quad \Leftrightarrow (x.exp < \text{LargestINTExp} \\ \quad\quad\quad \vee fv(x) = \text{MinInt})\end{array}$$

7.5 High level occam implementation

The previous section allows the high level occam implementation in Figure 2 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
```

Figure 2: high level occam implementation

Using two laws *IF pri* and *IF or-dist*

IF pri $IF(b_1\ P_1, \ldots, b_n\ P_n)$
$\equiv IF(b_1^*\ P_1, \ldots, b_n^*\ P_n)$
where $b_i^* = \neg b_1 \wedge \ldots \wedge \neg b_{i-1} \wedge b_i$

IF or-dist $IF(b_1\ P, b_2\ P, \underline{C})$
$\equiv IF(b_1 \vee b_2\ P, \underline{C})$

this can be simplified the program in Figure 3.

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

Figure 3: simplified high level implementation

which is probably the implementation of the specification that a competant programmer would produce – but the "special" case of MinInt is frequently omitted.

7.6 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 the occam program above 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 above contains a 3 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 is shown in Figure 4. The various laws for *IF* constructs in [12] 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
```

Figure 4: implementation with refined conditionals

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 variable to represent the various buses and conitional 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 is shown in Figure 5. The laws for *SEQ*, *VAR* and assignment in [12] verify this step

Introducing sequencing

The program now contains expressions and conditionals that can be formed in the microcode machine. However the program does not define what are microwords. The final step is to mimic the microsequencing

```
VAR AregNegative, ExpZbus, ExpZbusNeg, ExpZbusEqZ, FracZbusEqZ :
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)
            ExpZbusEqZ := ExpZbus = 0
            IF
              ExpZbusEqZ
                SEQ
                  FracZbusEqZ := FracZbus = 0
                  IF
                    FracZbusEqZ
                      SKIP
                    NOT FracZbusEqZ
                      ErrorFlag := TRUE
              NOT ExpZbusEqZ
                ErrorFlag := TRUE
    NOT AregNegative
      IF
        ExpZbusNeg
          SKIP
        NOT ExpZbusNeg
          ErrorFlag := TRUE
```

Figure 5: implementation with refined expressions

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 1 or 2 conditionals.

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

7.7 Translation to microcode

The final program from the transformations above is shown in Figure 6.

```
VAR NextInst :
VAR AregNegative, ExpZbusNeg, ExpZbusEqZ, FracZbusEqZ :
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
```

Figure 6: low level occam implemenation

This corresponds in an almost 1 to 1 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.

7.8 Microcode assembler source

Finally the microcode can be derived. This is shown in Figure 7.

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

CheckMinInt:
        ExpConstantFromLargestINTExp
        ExpXbusFromAreg                 ExpYbusFromConstant
        ExpZbusFromXbusMinusYbus
        FracXbusFromMSBit               FracYbusFromAreg
        FracZbusFromXbusMinusYbus
        GOTO Cond1FromExpZbusEqZ ->     (CheckMinInt2, OutofRange)

CheckMinInt2:
        GOTO Cond1FromFracZbusEqZ ->    (NOWHERE, OutofRange)

OutofRange:
        SetErrorFlag
        GOTO NOWHERE
```

Figure 7: microcode assembler source

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 4 microwords. Other instructions contain up to 90 microwords where informal development can easily introduce subtle bugs. The ability to verify a implementation using program transformations has proved invaluable.

8 Formal models of concurrency

Work on the IMS T800 has shown how correct microcode can be derived from a high level specification. However, this has relied only on the sequential aspects of the occam language.

Other aspects of the transputer which could benefit from a formal design approach are the process scheduler and the transputer communication system. Like floating point arithmetic, intrictate algorithms are employed in the interests of efficiency, and correctness is equally important. However, the communication system is a collection of concurrently operating hardware devices operating in a concurrent external environment - it is even more difficult to validate experimentally - even if prototypes are available. This is not peculiar to the transputer; all computers have similar input-output systems.

Work has recently started to produce a formal specification of the transputer scheduler and communication system and to refine this into the implementation in hardware and microcode.

9 A view of Silicon Design

The work described above tackles only one part of the verification process; it enables correct designs at the microcode level to be produced from high level specifications. To produce a fully 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.

Various small designs are currently being examined to experiment with this topic. By defining axioms for the behaviour of low level modules in the HDL module library tools such as HOL [5] can be used to verify HDL designs. This will enable provably correct VLSI designs to be produced.

A further step will be to fully integrate these tools to enable rapid re-checking of the correctness proof. This is essential, as modifications continue to be made throughout the design process.

Future silicon designs will involve a combination of standard microcomputers and specialised designs. An integrated design system must provide tools such as

1 Proof system to manipulate Z specifications and to develop occam implementations
2 occam transformation system
3 Silicon compiler(s) to compile occam directly into HDL implementations
4 Proof system to check designs performed at the HDL level
5 Layout system to generate and check layout from HDL

Clearly, this involves an integration of programming tools and silicon design tools. It also involves a change in the skills needed by a VLSI design team. The design of the T800 floating point unit was performed by:

1	computer architect	product specification
1	system designer	microcode, logic, datapath
1	'pure' mathematician	formal specification and proof
2	electronic engineers	circuit design and layout
1	programmer	compiler and instruction set tuning
1	'applied' mathematician	scientific function library

Large and complex designs are achieved by combining simple components. The relatively small set of basic modules are designed by electronic engineers to perform logical functions. As the technology advances, the design tools - and design skills - needed are those which enable large designs to be assembled reliably and quickly from the small set of basic modules. Software specialists, system designers and mathematicians have an increasingly important role to play in this area.

10 Simulation and Proof in VLSI design

It is common practice to make extensive use of simulation in VLSI design. As the number of devices on a VLSI chip increases, the simulation tools must be increased in speed to enable the design verification to be completed in an acceptable time. However, the purpose of a simulator is to check the *behavior* of the design by examination of test cases. As the designs increase in complexity, the number of test cases to be examined in simulation grows much faster than the number of devices used in the design. Consequently, it seems unlikely that simulation tools can keep up with VLSI technology, *even if massive concurrency is employed to accelerate simulation.*

Simulation is needed in VLSI design systems to model electrical properties of silicon devices. However, this is needed only for simple modules and simple combinations of modules. Simulation is also useful to demonstrate the functional properties of a proposed device, but this can operate at a very high level.

Simulators of datapaths, logic and microcode cannot tackle current VLSI designs. In contrast, even the existing primitive transformation and proof tools can tackle these aspects of the design. They enable a design to be completely verified much more rapidly than a simulator can provide a partial verification. These tools are therefore the key to future VLSI designs; they must be developed further and made interactive by the use of parallelism.

11 Conclusion

Techniques enabling formal design of *all* aspects of a computer are now required urgently, especially in such intricate areas as computer arithmetic and input-output. Most computers are now used in embedded systems where the cost of errors is very high - easily high enough to justify substantial investment in formal design techniques.

However, work at INMOS suggests that the use of formal techniques is already *cost-effective,* enabling designs to be produced more quickly with fewer designers and with less design hardware than by conventional techniques.

Existing theory is adequate in many areas of practical importance, although further devlopment is needed to provide the theoretical foundations for work involving concurrency and timing.

Computer aided design systems require substantial further development to allow integration of the various tools. Sufficient information must be included in the design database to allow rapid automatic checking of the correctness proof.

These problems are not unique to the semiconductor industry; anyone designing computers has encountered similar problems of validating complex hardware and microcode designs. It seems strange that the computer industry has made little use of formal design techniques - resulting in bugs in areas such as arithmetic units, interrupt systems and virtual memory systems. It is to be hoped that the use of formal techniques will give rise to a new generation of computers which are, at last, free of design errors.

References

[1] Barrett, G., *"Formal methods applied to a floating point number system"*, Technical Monograph PRG-58, Oxford University Computing Laboratory, Programming Research Group, 1987

[2] Good, D.I., *"Mechanical Proofs About Computer Programs"* in *"Mathematical Logic and Programming Languages"*, Prentice-Hall International, 1985

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

[4] Gordon, M., *"LCF_LSM"*, Technical Report 41, University of Cambridge Computer Laboratory, 1983

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

[6] Gordon, M., Milner, R., Wadsworth, C., *"Edinburgh LCF"* – chapter 2, LCNS 78, Springer Verlag, 1979

[7] Gries, D., *The science of programming"*, Springer-Verlag, 1981

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

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

[10] Homewood, M., May, D., Shepherd, D., Shepherd, R., *The IMS T800 Transputer*, IEEE Micro, Volume 7 Number 5, October 1987

[11] INMOS ltd., *"The occam programming manual"*, Prentice Hall, 1984

[12] Roscoe, A.W., Hoare, C.A.R., *"The laws of occam programming"*, Technical Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, 1986

[13] Sufrin, B.A., editor. *"The Z Handbook"*, Oxford University Computing Laboratory, Programming Research Group, 1986