Long arithmetic on the transputer

INMOS Technical Note 39

INMOS

72-TCH-039

Contents

1 Introduction

This note describes how to use the facilities provided in the transputer and occam to implement long arithmetic, i.e. arithmetic on arbitrarily large integers.

The transputer family naturally handles integers of the wordlength of the machine (16 bit on T2xx family, 32 bit on T4xx and T8xx families). There is also particular support from a communications point of view for bytes and messages of bytes, into which all other types can be mapped. The T4xx family has special instructions to accelerate software implementation of floating point numbers, and the T8xx family has hardware floating point facilities.

Occam supports bytes, 16-bit integers, 32-bit integers, 64-bit integers, 32-bit floating point and 64-bit floating point on all transputer types.

Floating point representation allows the expression and manipulation of very large values, but in so doing trades precision for range. Thus for absolute integer precision with very large number range, floating point is not appropriate.

Certain applications use long integers for other reasons, such as generating Cyclic Redundancy Checks, cryptography, spread-spectrum radio etc.

(Note that later members of the T4xx family, and all the T8xx family, include dedicated instructions for cyclic redundancy checking.)

The INMOS occam compilers give direct access to the transputer instructions through predefined procedures that compile into inline code, rather than a procedure call. This note demonstrates that the efficiency of these is such that the performance cannot be significantly improved by using assembly language.

2 Requirements

The need is to be able to perform arithmetic on integers of any length, not limited by the wordsize of the cpu. For simplicity, however, it is usual to implement an integer length that is a multiple of the cpu wordlength.

Clearly, arithmetic on such integers is going to be slower than on the machine's natural length, but it is a requirement that the overhead in such operations is not excessive (exact figures would be application dependent).

Certain applications have indicated needs for up to 3200 bit integers for cryptography, and 5115 bits for spread spectrum communications, so clearly the 64 bit facilities provided by occam need extension.

3 Facilities available on the transputer

The transputer instruction set has support for long arithmetic. These include instructions which perform addition and subtraction, with carries and borrows to allow extension to arbitrary length operations. These instructions are directly available in occam as predefined procedures, for which the compiler generates in-line code, with no procedure call overhead.

Both signed and unsigned versions are available when appropriate, and the procedures are listed below.

Occam predefines used for long arithmetic:

LONGADD	signed add with carry
LONGSUM	unsigned add with carry
LONGSUB	signed subtract with borrow
LONGDIFF	unsigned subtract with borrow
LONGPROD	unsigned multiply with carry-in
LONGDIV	unsigned divide
SHIFTRIGHT	double word shift right
SHIFTLEFT	double word shift left
NORMALISE	double word normalise
ASHIFTRIGHT	single word arithmetic shiftright -instruction sequence
ASHIFTLEFT	single word arithmetic shiftleft -instruction sequence

The add, sub and ashift predefines would be used for the most significant word of a long integer for signed work, using unsigned for the body of the integer.

For unsigned operands, sum, diff and shifts would be used throughout.

The next section gives the occam interface for these routines. A detailed definition of their operation, in occam, is given in section A, taken from reference [1]. Note that this is a definition; in general the compiler inserts only parameter loads and the instruction itself.

4 Interface description for the Occam Predefines

4.1 The integer arithmetic functions

LONGADD performs the addition of signed quantities with a carry in. The function is invalid if arithmetic overflow occurs.

```
INT FUNCTION LONGADD (VAL INT left, right, carry.in)
  -- Adds (signed) left word to right word with least significant
  -- bit of carry.in.
```

LONGSUM performs the addition of unsigned quantities with a carry in and a carry out. No overflow can occur.

```
INT, INT FUNCTION LONGSUM (VAL INT left, right, carry.in)
  -- Adds (unsigned) left word to right word with the least
  -- significant bit of carry.in.
  -- Returns two results, the first value is one if a carry occurs,
  -- zero otherwise, the second result is the sum.
```

LONGSUB performs the subtraction of signed quantities with a borrow in. The function is invalid if arithmetic overflow occurs.

```
INT FUNCTION LONGSUB (VAL INT left, right, borrow.in)
  -- Subtracts (signed) right word and borrow.in from left word.
```

LONGDIFF performs the subtraction of unsigned quantities with borrow in and borrow out. No overflow can occur.

```
INT, INT FUNCTION LONGDIFF (VAL INT left, right, borrow.in)
  -- Subtracts (unsigned) right word and borrow.in from left word.
  -- Returns two results, the first is one if a borrow occurs,
  -- zero otherwise, the second result is the difference.
```

LONGPROD performs the multiplication of two unsigned quantities, adding in an unsigned carry word. Produces a double length unsigned result. No overflow can occur.

```
INT, INT FUNCTION LONGPROD (VAL INT left, right, carry.in)
  -- Multiplies (unsigned) left word by right word and adds carry.in.
  -- Returns the result as two integers most significant word first.
```

LONGDIV divides an unsigned double length number by an unsigned single length number. The function produces an unsigned single length quotient and an unsigned single length remainder. An overflow will occur if the quotient is not representable as an unsigned single length number. The function becomes invalid if the divisor is equal to zero.

```
INT, INT FUNCTION LONGDIV (VAL INT dividend.hi, dividend.lo, divisor)
  -- Divides (unsigned) dividend. hi and dividend.lo by divisor.
  -- Returns two results the first is the quotient and the second
  -- is the remainder.
```

SHIFTRIGHT performs a right shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= 2*bitsperword`

```
INT, INT FUNCTION SHIFTRIGHT (VAL INT hi.in, lo.in, places)
  -- Shifts the value in hi. in and lo. in right by the given
  -- number of places. Bits shifted in are set to zero.
  -- Returns the result as two integers most significant word first.
```

SHIFTLEFT performs a left shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= 2*bitsperword`

```
INT, INT FUNCTION SHIFTLEFT (VAL INT hi.in, lo.in, places)
  -- Shifts the value in hi. in and lo. in left by the given
  -- number of places. Bits shifted in are set to zero.
  -- Returns the result as two integers most significant word first.
```

NORMALISE normalises a double length quantity. No overflow can occur.

```
INT, INT, INT FUNCTION NORMALISE (VAL INT hi.in, lo.in)
  -- Shifts the value in hi. in and lo. in left until the highest
  -- bit is set. The function returns three integer results
  -- The first returns the number of places shifted.
  -- The second and third return the result as two integers with
  -- the most significant word first;
  -- If the input value was zero, the first result is 2*bitsperword.
```

4.2 Arithmetic shifts

ASHIFTRIGHT performs an arithmetic right shift, shifting in and maintaining the sign bit. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

No overflow can occur.

N.B the result of this function is NOT the same as division by a power of two.

```
INT FUNCTION ASHIFTRIGHT (VAL INT operand, places)
  -- Shifts the value in operand right by the given number
  -- of places. The status of the high bit is maintained.
```

ASHIFTLEFT performs an arithmetic left shift, shifting out the most significant bits, and filling the least significant bits with zeroes. The function is invalid if significant bits are shifted out, i.e. if the most significant bit changes value at any point in the shift operation, signalling an overflow. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

N.B the result of this function is the same as multiplication by a power of two.

```
INT FUNCTION ASHiFTLEFT (VAL INT argument, places)
  -- Shifts the value in argument left by the given number
  -- of places. Bits shifted in are set to zero.
```

5 Methodology

This section will show the code required to provide general purpose long arithmetic. At this point the code will be written for algorithmic efficiency but occam clarity. For ultimate performance see the performance section for optimisations.

For each operation, a procedure will be demonstrated that takes as input arrays of integers being the operands, and returns an array of integers that is the result. All operands are arbitrarily sized integer arrays, so that if fed arrays of 100 words, that is the size for which the operation will be performed. To be safe, such routines should test for compatibility of the sizes of the three arrays passed, which should be identical for add and subtract, n,n and 2n for multiply and divide. The code for this is omitted from the examples below for clarity.

The first two, add and subtract, are trivial, and as the result has the same length as the operands, are shown for the operation `a := a op b`. Multiply is simple to program, but there are subtleties of the powerful transputer operations to be noted to achieve algorithmic efficiency. Divide accentuates this even further, and is more dependent on the algorithm, for which one is referred to [2]. As the result has a different length from the operands, multiply and divide are illustrated for the operation `a := b op c`.

All arrays are assumed to be stored little-endian, i.e. the element with the lowest index is the least significant. The code could equally support big- endian arrays if required. Within a word, the bits MUST be stored littleendian to match the transputer hardware.

5.1 Addition

As an example, the general addition code will be derived from first double-length, then triple-length.

The following example adds two double length unsigned quantities explicitly. occam actually supports this implicitly as type INT64.

```
PROC add.double.unsigned ([2]INT result, VAL [2]INT rightop)
  INT carry:
  SEQ
    carry , result[0] := LONGSUM (result[0],rightop[0],0)
    carry , result[1] := LONGSUM (result[1],rightop[1],carry)
:
```

To make this a signed operation, the final operation, i.e. that on the most significant word, is performed using the LONGADD predefine. This produces exactly the same result, but will raise the error flag if overflow should occur.

```
PROC add.double.signed ([2]INT result,VAL [2]INT rightop)
  INT carry:
  SEQ
    carry , result[0] := LONGSUM (result[0],rightop[0],0)
    result[1] := LONGADD (result[1],rightop[1],carry)
:
```

To extend beyond double length simply involves using the same operation again.

```
PROC add.triple.signed ([3]INT result,VAL [3]INT rightop)
  INT carry:
  SEQ
    carry , result[0] := LONGSUM (result[0],rightop[0],0)
    carry , result[1] := LONGSUM (result[1],rightop[1],carry)
    result[2] := LONGADD (result[2],rightop[2],carry)
:
```

To create the general purpose case, one uses a loop rather than in-line code, and controls the loop with the length of the array passed in, which is accessible at runtime in occam. The following code performs a long-add on compatible arrays of integers. This version operates on signed integers, clearly one could make it slightly smaller for unsigned integers by looping one more time and omitting the last statement.

```
PROC add.long.signed.int([]INT result,VAL []INT rightop)

  INT carry:
  VAL last.index IS (SIZE rightop) -1:

  SEQ
    carry := 0

    SEQ i = 0 FOR last.index
      carry,result[i] := LONGSUM (result[i],rightop[i],carry)

    result[last.index] := LONGADD (result[last.index],
                                  rightop[last.index],carry)
:
```

5.2 Subtraction

Similarly for subtraction, using the appropriate pair of predefines in exactly the same harness, with the same comments applying.

```
PROC subtract.long.signed.int([]INT result,VAL []INT rightop)

  INT borrow:
  VAL last.index IS (SIZE rightop) -1:

  SEQ
    borrow := 0

    SEQ i = 0 FOR last.index
      borrow,result[i] := LONGDIFF (result[i],rightop[i],borrow)

    result[last.index] := LONGSUB (result[last.index],
                                  rightop [last.index],borrow)
```

5.3 Multiplication

Multiplication is more complex. The algorithm is best developed by mimicking a child doing long multiplication. Imagine multiplying 12 by 34. First, one removes the sign from both operands, generating the result sign as s1 XOR s2.

Then one takes the least significant digit of the multiplier(4) and multiplies all the digits of the multiplicand by it, writing the answer down with no shift to the left. One then takes the next digit to the left(3) and multiplies all digits of the multiplicand by it, writing the result down with a one digit shift to the left. This continues until all digits have been multiplied, then all the partial results are added for a final result.

```
  12    12    12    12
x 34    34    34    34
----- ----- ----- -----
        48    48    48
             36    36 +
            ----- -----
                   408
```

To analyse this operation, consider the position we write each result. Units times units we write in the units column, tens times tens in the hundreds column, so clearly the destination column is 10 to the power (m + n), where m and n are the powers of ten of the corresponding operand columns.

To implement this directly on a computer would require a large amount of memory. A 100 word integer would require 100 rows of intermediate results, and each row would be around 100 words... i.e. ten thousand words of memory, or 40K bytes on a 32 bit machine. Also, the control and implementation of the final add would use excessive cpu time.

The solution on the transputer instruction set, mapped into occam via the predefined procedures, is to incorporate the add operation into each multiply, so that only one intermediate result is maintained, and that can clearly occupy the same space as the final result will occupy eventually. In order to achieve this, the result array must be cleared before use. Remember this code is designed for clarity... efficiency comes later. Clearly (i+j) could be evaluated once in the inner loop, and clearly leftop [i] need only be accessed once per outer loop.

```
PROC multiply.long.unsigned.int([]INT result,VAL []INT leftop,rightop)
  SEQ

    SEQ i = 0 FOR SIZE result
      result[i] := 0

    SEQ i = 0 FOR SIZE leftop

      INT carry:
      SEQ
        carry := 0
```

```
          SEQ j = 0 FOR SIZE rightop
            INT temp:
            SEQ

              temp, result[i+j] :=
                             LONGPROD(leftop[i], rightop[j], result[i+j])
              carry, result[i+j+1] := LONGSUM(result[i+j+1],temp,carry)
  :
```

The reason this comes out so simply comes from the design of the instructions. Note that the multiply operation performs an accumulation for us, as its carry input takes a full word width operand. We need a double width accumulate however, so a long-sum is used to complete the operation for the upper word of the result. Note that the carry from the long-sum is required TWO words further up the result. This is achieved by holding it until the next iteration, when the index will be one higher, and storing it one higher than the index as usual.

Note that this algorithm will actually multiply arrays of differing length, providing the length of the result array is appropriate. Note also that as the loop counts start at zero, the highest loopcontrol value being n-i, the final access of the final carry operation is to imax+jmax+1. The result array is of size 2n, but the last access is to index (n-1) + (n-1) + 1, i.e. 2n-1, the correct last address.

There is a more efficient way of dealing with signed multiply than mimicking the human. Deleting a minus sign is easy, but negating a very long twos-complement integer may not be, as it could affect the bit pattern of every word.

The solution is to perform a multiply assuming the operands are both positive, and then to correct the result by subtraction if either of them was negative, as shown below

```
  PROC multiply.long.signed.int([]INT result,VAL []INT leftop,rightop)
    SEQ
      multiply.long.unsigned.int(result,leftop,rightop)

      result.top.half IS
           [result FROM SIZE leftop FOR SIZE rightop]:
      VAL leftop.top.word IS leftop[(SIZE leftop) -1]:
      IF
        leftop.top.word < 0
          subtract.long.unsigned.int (result.top.half,rightop)
        TRUE
          SKIP
      result.top.half IS
           [result FROM SIZE rightop FOR SIZE leftop]:
      VAL rightop.top.word IS rightop[(SIZE rightop) -1]:
      IF
        rightop.top.word < 0
          subtract.long.unsigned.int (result.top.half,leftop)
        TRUE
          SKIP
  :
```

Note that the above code handles differing length arrays, but assumes that the long subtract procedure can also, which the example given thereof cannot.

5.4 Division

Division is another order of magnitude more complex than multiplication. As with multiplication, one mimics the human, extracting the sign operation first. To perform the division, a human makes a guess at a partial result, multiplies back and subtracts to achieve a remainder. If the remainder is negative, the guess was too large, so is repeated. If the remainder is greater than the divisor, the guess was too small.

The computer does exactly the same, but it can be 'helped' in making its initial guess appropriately.(Ref 2, Knuth)

```
  PROC divide.long.int([]INT result, VAL []INT leftop,rightop)
    ...  declarations
    SEQ
      ...  extract signs from operands -> s.r, left.u,right.u
      ...  normalise divisor (right.u)
      higher.word.left := 0
      SEQ i = 0 FOR SIZE leftop
        VAL i.l.rev IS (SIZE leftop)-1) - i:
        VAL i.r.rev IS (SIZE rightop)-1) - i:
        SEQ
          IF
            higher.word.left = right.u[i.r.rev]
              temp.result := MAX INT
            TRUE
              temp.result,remainder := LONGDIV ( higher.word.left,
                                 left.u[i.r.rev], right.u[i.l.rev])

          IF
            temp.result<>0
              SEQ
```

```
                multiply.long.unsigned.int( temp.vec,
                                            [temp.result] ,right.u)
                subtract.long.unsigned.int(left.u,temp.vec)

                WHILE (left.u[(SIZE left.u)-1] /\ signbit) <> 0
                  SEQ
                    temp.result := temp.result - 1
                    add.long.unsigned.int( left.u, right.u)

            TRUE
              SKIP

          result[i.l.rev] := temp.result
          higher.word.left := left.u[i.l.rev]

      ...  unnormalise wrt previous normalisation
      ...  replace sign of result.
  :
```

The trial result temp.result can be either one or two units too large. To cover this possibility, the WHILE loop tests if the current remainder is negative, and adds back the divisor as many times as necessary to remedy the situation, adjusting the estimated quotient each time.

There is also a marginally faster version in Knuth that adds an extra test to improve the guess, and thus guarantees to be accurate or one high, never two high.

6 Shift Operations

The shift operations are simple in that there is only a single primitive to be used per loop, but note should be taken that the same performance enhancements as will be demonstrated on multiplication in section 7.1, particularly opening out the loops and using abbreviations on blocks of sixteen words, should be used here, as the actual shift operation is dwarfed in CPU time by the loop control around it otherwise.

Note that the examples given here assume a shift of less than the natural wordlength of the machine. In fact for any shift of 16 bits or greater, it is more efficient to do a block move assignment to handle the byte offset, and only tidy up the remaining bit-shift using this code.

It should also be noted that the shift left and rotate left operations should be performed from the 'top' of the vector, which means reversing the loop index, whilst the shift/rotate right operations are done from the lowest index.

```
PROC rotate.long.int.left([]INT buffer,VAL INT n)
  INT highest,dump:
  VAL last IS (SIZE buffer) -1:
  SEQ
    highest := buffer[last]
    SEQ ii = 0 FOR last
      VAL i IS last - ii:
      buffer[i],dump := SHIFTLEFT(buffer[i],buffer[i-1],n)
    buffer[0],dump   := SHIFTLEFT(buffer[0],highest,n)
:
```

To make it an unsigned shift left, one simply omits the wrap around. Note that as a result, the last operation can be done with a conventional single length shift.

```
PROC shift.long.int.left([]INT buffer,VAL INT n)
  INT dump:
  VAL last IS (SIZE buffer) -1:
  SEQ
    SEQ ii = 0 FOR last
      VAL i IS last - ii:
      buffer[i],dump := SHIFTLEFT(buffer[i],buffer[i-1],n)
    buffer[0] := buffer[0] << n
:
```

For arithmetic shift, one simply handles the first (most significant) word separately. Because the arithmetic shift on the transputer is single length, it is appropriate to use this for overflow checking only, and repeat the operation with a logical shift. Thus one simple inserts the following line before the loop. (This does not apply to right shift, see later)

```
dump := ASHIFTLEFT(buffer[last],n)
```

Similarly, for right shifts, noting that we progress the other direction along the vector:

```
PROC rotate.long.int.right([]INT buffer, n)
  INT first,dump:
  VAL last IS (SIZE buffer) -1:
  SEQ
    first := buffer[0]
    SEQ i = 0 FOR last
      dump,buffer[i]  := SHIFTRIGHT(buffer[i+i],buffer[i],n)
    dump,buffer[last] := SHIFTRIGHT(first,buffer[last],n)
:
```

Again, shift and arithmetic shift involve removing the wraparound, and using ASHIFTRiGHT for the last line, respectively. Note that the arithmetic shift is done last, not first, with the right shift, as we progress up the vector from least significant to most significant.

6.1 Normalisation

Normalisation is similar to shifting, but is conditional on the data, so is split into five sections. Firstly, from the most significant end, the first non-zero word is found. That word is then normalised, using a single application of the normalise predefine, which also pulls in any bits needed from the next word down. The third operation is to shift all the remaining words left by the number of places returned by the normalise routine, and then the fourth is to move the active words, now correctly word aligned, to the top of the array. The final operation is to clear the vacated words at the bottom of the array.

```
PROC normalise.long.integer ([]INT buffer)
  INT pointer, trash :
  INT places :
  VAL len IS SIZE buffer :
  SEQ
    -- find first non-zero word
    IF
      IF i = 1 FOR len
        buffer[len MINUS i] <> 0
          pointer := len MINUS i
        TRUE
          pointer := 0

    -- normalise that word, pulling in bits as
    -- needed from next word down
    places,buffer[len],trash :=
       NORMALISE(buffer[pointer],buffer[pointer-1])

    VAL diff IS (len) MINUS (pointer)
    SEQ
      -- shift the rest of the buffer left by the
      -- same number of bits
      shift.left ([buffer FROM 0 FOR pointer],places)

      -- block move up to the top of the buffer
      SEQ i = 0 FOR (pointer PLUS 1)
        VAL ii IS (pointer) MINUS (i) :
        buffer[ii PLUS diff] := buffer[ii]
      -- fill the vacated words with zeros
      SEQ ii = 0 FOR (diff MINUS 1)
        buffer [ii] := 0
:
```

7 Performance

7.1 Optimisation, using multiplication as an example

Addition and subtraction are sufficiently simple, and sufficiently directly built on the transputer instructions that little algorithmic optimisation can be done. However, as the arithmetic operations are so fast, they suffer greatly from the loop control overhead, so benefit greatly from opening out the loops.

However multiplication can be considerably optimised, in three steps. The first is to take invariant expressions outside loops, saving both indexing and arithmetic. The second is to set up abbreviations (or pointers) to frequently accessed arrays.

The third, and most beneficial, is to open out the inner loop by some factor, ideally sixteen. This both saves loop control, but combined with the abbreviations means that many array accesses are reduced to constant indices, which are very fast on the transputer. This does have the restriction that the arrays must be a multiple of the opening factor, rather than truly variable size. The second method is a stepping stone to the third, and the performance benefit of it is not seen until the loop is opened.

The following sections show the inner two loops suitably modified:

Simple Code

```
  SEQ i = 0 FOR SIZE leftop

    SEQ
      carry := 0
      VAL leftop.i IS leftop[i]:

      SEQ j = 0 FOR SIZE rightop -- usually the same size

        VAL ij  IS i  + j :
        VAL ij1 IS ij + 1 :
        SEQ
          temp, result[ij] :=
                     LONGPROD(leftop.i, rightop[j], result[ij])
        carry, result[ij1] := LONGSUM(result[ij1],temp,carry)
```

Using Array Abbreviations and opened loops

```
SEQ i = 0 FOR SIZE leftop

  SEQ
    carry := 0
    VAL leftop.i IS leftop[i]:

    SEQ j = 0 FOR (SIZE rightop)/opening.factor

      VAL ij  IS i  + j :
      VAL ij1 IS ij + 1 :
      VAL rightop.j IS [rightop FROM (J TIMES opening.factor )
                        FOR opening.factor]:
      result.ij IS [result FROM i+(j TIMES opening.factor)
                        FOR opening.factor + 1]:
      SEQ

        VAL k IS 0:
        SEQ
          temp, result.ij[k] :=
                      LONGPROD(leftop.i,rightop.j[k],result.ij[k])

          carry, result.ij[k+1] :=
                      LONGSUM (result.ij[k+1],temp, carry)

        VAL k IS 1:
        SEQ
          .
          .
        .
        .
        VAL k IS opening.factor-1:
        SEQ
          .
          .
```

Points to note here are that now all access to the arrays are of the form k or k + constant. As k itself is a constant, this will be folded at compile time, so all array accesses in the inner loop have constant indices and are thus very fast, using instructions designed for that operation.

7.2 Performance Figures

The following table gives the performance of each of the algorithms mentioned above, and for the optimised versions with open loops, and finally for the same operation in assembly language. These numbers are all for a 20MHz 32 bit transputer.

Performance in microseconds for 3200 bit integer

OPERATION	**SIMPLE**	**OPTIMISED OCCAM**	**ASSEMBLER**
ADD/SUBTRACT	305	164	164
MULTIPLY	57700	43500	43485
DIVIDE	36-72ms	-	-
SHIFT/ROTATE 1 bit	300	167	153
SHIFT/ROTATE 8 bit	336	202	187
SHIFT/ROTATE 15 bit	372	236	211
SHIFT/ROTATE 8*N	41	41	41
NORMALISE	384	-	-

Note that assembler coding gains very little, and also the spectacular performance of the shift 8*N version, due to the block move hardware in the transputer.

Divide spends most of its time multiplying out the estimated results, so has little scope for optimising other than in the multiply. Divide and Normalise have data-dependent execution times. The figures given are mid-range, i.e. (max.time+min.time)/2.

8 Conclusions

The long arithmetic facilities in occam allow very efficient implementation of arbitrary length integers, meaning that there is no benefit in using assembler.

The underlying instructions of the transputer are directly accessible as in-line procedures in occam, and are themselves sophisticated primitives allowing maximum performance in such computationally intense applications.

The assistance of Roger Shepherd (INMOS, Architecture Group) is gratefully acknowledged, particularly for help with the divide algorithms, and of Andy Hamilton and John Carey, INMOS Central Applications, for benchmarking and verification work.

A The Occam Predefined Procedures

A.1 Definition of terms

For the purpose of explanation imagine a new type INTEGER, and the associated conversion. This imaginary type is capable of representing the complete set of integers and is presumed to be represented as an infinite bit two's complement number. With this one exception the following are occam descriptions of the various arithmetic functions.

```
-- constants used in the following description
VAL bitsperword IS machine. wordsize(INTEGER) :
VAL range       IS storeable. values(INTEGER) :
                -- range = 2^bitsperword
VAL maxint      IS INTEGER (MOSTPOS INT) :
                -- maxint = (range/2 - 1)
VAL minint      IS INTEGER (MOSTNEG INT) :
                -- minint = -(range/2)
-- INTEGER literals
VAL one         IS 1(INTEGER) :
VAL two         IS 2(INTEGER) :
-- mask
VAL wordmask    IS range - one :
```

In occam, values are considered to be signed. However, in these functions the concern is with other interpretations. In the construction of multiple length arithmetic the need is to interpret words as containing both signed and unsigned integers. In the following the new INTEGER type is used to manipulate these values, and other values which may require more than a single word to store.

The sign conversion of a value is defined in the functions unsign and sign. These are used in the description following but they are NOT functions themselves.

A.2 The integer arithmetic functions

LONGADD performs the addition of signed quantities with a carry in. The function is invalid if arithmetic overflow occurs.

The action of the function is defined as follows:

```
INT FUNCTION LONGADD (VAL INT left, right, carry.in)
  -- Adds (signed) left word to right word with least significant
```

```
    -- bit of carry.in.

    INTEGER sum.i, carry.i, left.i, right.i :
    VALOF
      SEQ
        carry.i := INTEGER (carry.in /\ 1)
        left. i := INTEGER left
        right.i := INTEGER right
        sum.i   := (left.i + right.i) + carry.i
      -- overflow may occur in the following conversion
      -- resulting in an invalid process
      RESULT INT sum.i
  :
```

LONGSUM performs the addition of unsigned quantities with a carry in and a carry out. No overflow can occur.

The action of the function is defined as follows:

```
  INT, INT FUNCTION LONGSUM (VAL INT left, right, carry.in)
    -- Adds (unsigned) left word to right word with the least
    -- significant bit of carry.in.
    -- Returns two results, the first value is one if a carry occurs,
    -- zero otherwise, the second result is the sum.

    INT carry.out :
    INTEGER sum.i, left.i, right.i :
    VALOF
      SEQ
        left.i  := unsign (left)
        right.i := unsign (right)
        sum.i   := (left.i + right.i) + INTEGER (carry.in /\ 1)
        IF                    -- assign carry
          sum.i >= range
            SEQ
              sum.i := sum.i - range
              carry.out := 1
          TRUE
            carry.out := 0
      RESULT carry.out, sign (sum.i)
  :
```

LONGSUB performs the subtraction of signed quantities with a borrow in. The function is invalid if arithmetic overflow occurs.

The action of the function is defined as follows:

```
  INT FUNCTION LONGSUB (VAL INT left, right, borrow.in)
```

```
    -- Subtracts (signed) right word from left word and subtracts
    -- borrow.in from the result.

    INTEGER diff.i, borrow.i, left.i, right.i :
    VALOF
      SEQ
        borrow.i := INTEGER (borrow.in /\ 1)
        left.i   := INTEGER left
        right.i  := INTEGER right
        diff.i   := (left.i - right.i) - borrow.i
      -- overflow may occur in the following conversion
      -- resulting in an invalid process
      RESULT INT diff.i
  :
```

LONGDIFF performs the subtraction of unsigned quantities with borrow in and borrow out. No overflow can occur.

The action of the function is defined as follows:

```
  INT, INT FUNCTION LONGDIFF (VAL INT left, right, borrow.in)
    -- Subtracts (unsigned) right word from left word and subtracts
    -- borrow.in from the result.
    -- Returns two results, the first is one if a borrow occurs,
    -- zero otherwise, the second result is the difference.

    INTEGER diff.i, left.i, right.i :
    VALOF
      SEQ
        left.i  := unsign (left)
        right.i := unsign (right)
        diff.i  :=  (left.i - right.i) - INTEGER (borrow.in /\ 1)
        IF                    -- assign borrow
          diff.i < 0
            SEQ
              diff.i := diff.i + range
              borrow.out := 1
          TRUE
            borrow.out := 0
      RESULT borrow.out, sign (diff.i)
  :
```

LONGPROD performs the multiplication of two unsigned quantities, adding in an unsigned carry word. Produces a double length unsigned result. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGPROD (VAL INT left, right, carry.in)
  -- Multiplies (unsigned) left word by right word and adds carry.in.
  -- Returns the result as two integers most significant word first.

  INTEGER prod.i, prod.lo.i, prod.hi.i, left.i, right.i, carry.i :
  VALOF
    SEQ
      carry.i   := unsign (carry.in)
      left.i    := unsign (left)
      right.i   := unsign (right)
      prod.i    := (left.i * right.i) + carry.i
      prod.lo.i := prod.i REM range
      prod.hi.i := prod.i / range
    RESULT sign (prod.hi.i), sign (prod.lo.i)
:
```

LONGDIV divides an unsigned double length number by an unsigned single length number. The function produces an unsigned single length quotient and an unsigned single length remainder. An overflow will occur if the quotient is not representable as an unsigned single length number. The function becomes invalid if the divisor is equal to zero.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGDIV (VAL INT dividend.hi, dividend.lo, divisor)
  -- Divides (unsigned) dividend. hi and dividend.lo by divisor.
  -- Returns two results the first is the quotient and the second
  -- is the remainder.

  INTEGER divisor.i, dividend.i, hi, lo, quot.i, rem.i :
  VALOF
    SEQ
      hi         := unsign (dividend.hi)
      lo         := unsign (dividend.lo)
      divisor.i  := unsign (divisor)
      dividend.i := (hi * range) + lo
      quot.i     := dividend.i / divisor.i
      rem.i      := dividend.i REM divisor.i
    -- overflow may occur in the following conversion of quot.i
    -- resulting in an invalid process
    RESULT sign (quot.i), sign (rem.i)
:
```

SHIFTRIGHT performs a right shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= 2*bitsperword`

The action of the function is defined as follows:

```
INT, INT FUNCTION SHIFTRIGHT (VAL INT hi.in, lo.in, places)
  -- Shifts the value in hi. in and lo. in right by the given
  -- number of places. Bits shifted in are set to zero.
  -- Returns the result as two integers most significant word first.

  INT hi.out, lo.out:
  VALOF
    IF
      (places < 0) OR (places > (two*bitsperword))
        SEQ
          hi.out := 0
          lo.out := 0
      TRUE
        INTEGER operand, result, hi, lo :
        SEQ
          hi      := unsign (hi.in)
          lo      := unsign (lo.in)
          operand := (hi << bitsperword) + lo
          result  := operand >> places
          lo      := result /\ wordmask
          hi      := result >> bitsperword
          hi.out  := sign (hi)
          lo.out  := sign (lo)
    RESULT hi.out, lo.out
:
```

SHIFTLEFT performs a left shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= 2*bitsperword`

The action of the function is defined as follows:

```
INT, INT FUNCTION SHIFTLEFT (VAL INT hi.in, lo.in, places)
  -- Shifts the value in hi. in and lo. in left by the given
  -- number of places. Bits shifted in are set to zero.
  -- Returns the result as two integers most significant word first.

  VALOF
    IF
      (places < 0) OR (places > (two*bitsperword))
        SEQ
          hi.out := 0
          lo.out := 0
      TRUE
```

```
        INTEGER operand, result, hi, lo :
        SEQ
          hi      := unsign (hi.in)
          lo      := unsign (lo.in)
          operand := (hi << bitsperword) + lo
          result  := operand << places
          lo      := result /\ wordmask
          hi      := result >> bitsperword
          hi.out  := sign (hi)
          lo.out  := sign (lo)
    RESULT hi.out, lo.out
:
```

NORMALISE normalises a double length quantity. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT, INT FUNCTION NORMALISE (VAL INT hi.in, lo.in)
  -- Shifts the value in hi. in and lo. in left until the highest
  -- bit is set. The function returns three integer results
  -- The first returns the number of places shifted.
  -- The second and third return the result as two integers with
  -- the most significant word first;
  -- If the input value was zero, the first result is 2*bitsperword.

  INT places, hi.out, lo.out :
  VALOF
    IF
      (hi.in = 0) AND (lo.in = 0)
        places := INT (two*bitsperword)
      TRUE
        VAL msb IS one << ((two*bitsperword) - one) :
        INTEGER operand, hi, lo :
        SEQ
          lo      := unsign (lo.in)
          hi      := unsign (hi.in)
          operand := (hi << bitsperword) + lo
          places  := 0
          WHILE (operand /\ msb) = 0
            SEQ
              operand := operand << one
              places  := places + 1
          hi      := operand / range
          lo      := operand REM range
          hi.out  := sign (hi)
          lo.out  := sign (lo)
      RESULT places, hi.out, lo.out
:
```

A.3 Arithmetic shifts

ASHIFTRIGHT performs an arithmetic right shift, shifting in and maintaining the sign bit. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

No overflow can occur.

N.B the result of this function is NOT the same as division by a power of two.

e.g. -1/2 = 0
ASHIFTRIGHT (-1,1) = -1

The action of the function is defined as follows:

```
INT FUNCTION ASHIFTRIGHT (VAL INT operand, places) IS
            INT( INTEGER (operand) >> places )
  -- Shifts the value in operand right by the given number
  -- of places. The status of the high bit is maintained.
:
```

ASHIFTLEFT performs an arithmetic left shift. The function is invalid if significant bits are shifted out, signalling an overflow. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

N.B the result of this function is the same as multiplication by a power of two. The action of the function is defined as follows:

```
INT FUNCTION ASHIFTLEFT (VAL INT argument, places)
  -- Shifts the value in argument left by the given number
  -- of places. Bits shifted in are set to zero.

  INTEGER result.i :
  VALOF
    result.i := INTEGER (argument) << places
    -- overflow may occur in the following conversion
    -- resulting in an invalid process
    RESULT INT result.i
:
```

A.4 Word rotation

ROTATERIGHT rotates a word right. Bits shifted out of the word on the right, re-enter the word on the left. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

No overflow can occur.

The action of the function is defined as follows:

```
INT FUNCTION ROTATERIGHT (VAL INT argument, places)
  -- Rotates the value in argument by the given number of places.

  INTEGER high, low, argument.i :
  VALOF
    SEQ
      argument.i := unsign(argument)
      argument.i := (argument.i * range) >> places
      high       := argument.i / range
      low        := argument.i REM range
    RESULT INT (high \/ low)
:
```

ROTATELEFT rotates a word left. Bits shifted out of the word on the left, re-enter the word on the right. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

The action of the function is defined as follows:

```
INT FUNCTION ROTATELEFT (VAL INT argument, places)
  -- Rotates the value in argument by the given number of places.

  INTEGER high, low, argument.i :
  VALOF
    SEQ
      argument.i := unsign(argument)
      argument.i := argument.i << places
      high       := argument.i / range
      low        := argument.i REM range
    RESULT INT(high \/ low)
:
```

References

[1] occam 2 Reference Manual, INMOS Limited, Prentice Hall 1988, ISBN 0-13-629312-3 (Particularly Appendix L,pp105-113.)

[2] Seminumerical Algorithms, The Art of Computer Programming Vol 2, Donald Knuth, Addison-Wesley 1969,1981, ISBN 0-201-03822-6(v.2) (Particularly Section 4.3)