Integer Division by Small Constants

Technical Report No 113

Gordon B Steven

Submitted to 10th IEEE Symposium on Computer Arithmetic

November 1990

Integer Division by Small Constants

Abstract
Integer division is considered within the context of the development of iHARP, an integrated circuit version of HARP, the HAtfield RISC Processor architecture. The paper demonstrates that execution times for division by small constants can be significantly reduced. Two cases are considered, division by powers of two and division by other small constants. In each case specialised instruction primitives are introduced to reduce the execution time.

If an arithmetic shift right instruction is used to implement division by a power of two, an incorrect result is produced for negative dividends. The first iHARP primitive corrects the shift result and allows division by powers of two to be performed in a single machine cycle. The second iHARP primitive allows division by other small constants to be implemented efficiently as a finite series of multiplication steps. Execution time is reduced both by decreasing the number of operations involved and by exploiting the parallel nature of the iHARP architecture.

Key words
Integer division RISC VLIW

Gordon Steven

November 1990

1. Introduction

This paper considers the division of integers by constants as a separate design problem. The aim is to reduce the execution time of division in the many cases where the divisor is known at compile time. Unlike the general problem of integer division, this topic has been neglected in the literature with the notable exception of a recent paper from the HP Precision Architecture group [Magen87].

The work described was undertaken during the course of the development of iHARP, a VLIW (Very Long Instruction Word) processor currently being developed at Hatfield Polytechnic [Steven89][Adams90a]. In each processor cycle iHARP fetches four integer instructions from the instruction cache and executes them in parallel in four distinct pipelines. An iHARP processor is therefore capable of performing four ALU operations in parallel. The ALU configuration for each pipeline is shown in Fig1. The shift unit in front of the ALU is capable of shifting one of the ALU operands a small number of bit positions left or right. Other distinctive features of iHARP include the conditional execution of all instructions [Adams90b] and ORed indexing [Steven88].

The paper first outlines the implementation of general-purpose division on iHARP. The use of arithmetic shift right instructions to implement division by powers of two is then reviewed, and the additional hardware provided by iHARP to allow divisions by powers of two to be executed in a single cycle is described.

An algorithm for division by three is then developed and its hardware implementation on iHARP described. A subsequent section describes how this algorithm can be generalised to other constants. Finally comparative execution times for division by various small constants are provided. The paper demonstrates that execution times for division by small constants can be significantly reduced. The improvement comes from two sources. Firstly, the number of operations is reduced by converting the division into a multiplication. Secondly, the algorithm employed is able to exploit the multiple parallel pipelines provided by iHARP.

2. General-Purpose Division on iHARP

General-purpose division on iHARP uses a non-restoring division algorithm [Gosling80]. The quotient is produced one bit at a time using a single length add or subtract in conjunction with a double length shift.

The algorithm is implemented using a DSTEP instruction primitive which provides the required double length shift and add/subtract step. The shift operation uses the shift unit placed immediately before the ALU in each pipeline (Fig1). An unsigned, 32-bit division operation is executed in 33 cycles. In the case of signed division the execution time increases to 36 machine cycles.

Each DSTEP instruction is executed by two adjacent pipelines acting in concert. The first pipeline performs the upper half of the double length shift and the add/subtract step while the second pipeline performs the lower half of the shift. A divide routine therefore requires the use of two pipelines but leaves the two remaining pipelines free to perform other operations including a second division. Four additional 32-bit shift registers would have been required to implement a DSTEP instruction in each of the four iHARP pipelines. These registers would have increased the machine state and would have required saving on context switches. This overhead is avoided in the solution adopted.

In addition iHARP provides special-purpose instructions to reduce the execution time of divisions where the divisor is known at compile time. These primitives provide the main subject of this paper.

3. Division by Powers of Two

It is well known that an arithmetic shift right only corresponds to division by a power of two if the operand is positive [Steele79]. Consider for example "-1 DIV 2". An arithmetic shift right of one will produce the answer minus one, whereas most programming languages expect the answer zero. The problem is that an arithmetic shift right rounds towards minus infinity while most languages follow FORTRAN and require integer division to round towards zero. The following algorithm must therefore be used to divide a signed integer, Ri, by N where N = 2 ** k:

```
IF Ri < 0 THEN Ri := Ri + (N-1);
Ri := Ri (ASR#k)
```

On iHARP this algorithm can be implemented using the following three sequential instructions:

```
1)  LTS B1,Ri,#0            /* B1:= Ri < #0 */
2)  TB1 ADD Ri,Ri,#N-1      /* IF B1=TRUE THEN Ri := Ri+N-1 */
3)  MOV Rj,Ri(ASR#k)        /* Rj := Ri shifted right k bits arithmetically */
```

Alternatively by executing instructions in two parallel pipelines, the execution time can be reduced to two cycles:

```
1)  LTS B1,Ri,#0;          ADD Rj,Ri,#N-1          /* speculatively adjust dividend */
2)  FB1 MOV Rj,Ri(ASR#k);  TB1 MOV Rj,Rj(ASR#k)    /* use B1 to select correct result */
```

Neither method is ideal, the first not only takes three instructions but destroys the original operand, while the second requires four short instructions. An alternative algorithm performs the required correction after the shift operation.

```
1)  MOV Rj,Ri(ASR#k)  /* carry flag set if Ri < 0 and a one shifted out during shift */
2)  ADDC Rj,Rj,#0     /* correct rounding */
```

In the first instruction the carry flag is set if the shift operand is negative and if any bit discarded during the shift operation is one. An add with carry instruction then uses the carry flag to correct the rounding in the case of negative dividends. On iHARP these two operations are combined in a single primitive. The shift unit in front of the ALU performs the arithmetic shift, while the ALU performs the correction required for negative operands.

```
1)  ADDC Rj,Ri(ASR#k),#0      /* Carry In = 1 if Ri < 0 and at least one "1" is discarded during
                                 the shift  operation */
```

As a result divisions by powers of two are executed in a single processor cycle.

4. Division by Three

This section introduces the iHARP approach to division by constants which are not powers of two. Consider a 32-bit positive number being divided by three. This division is equivalent to multiplication by 0.0101etc where the "01" is repeated indefinitely. The start of the series of additions required is graphically represented below. Each letter represents a single digit in the 32-bit dividend. All unrepresented bit positions are zero.

```
dddddddddddddddddddddddddddddd.dd
  dddddddddddddddddddddddddddd.dddd
    dddddddddddddddddddddddddd.dddddd
      dddddddddddddddddddddddd.dddddddd
        dddddddddddddddddddddd.dddddddddd
          dddddddddddddddddddd.dddddddddddd
            dddddddddddddddddd.dddddddddddddd
              dddddddddddddddd.dddddddddddddddd
                dddddddddddddd.dddddddddddddddddd
                  dddddddddddd.dddddddddddddddddddd
                    dddddddddd.dddddddddddddddddddddd
                      dddddddd.dddddddddddddddddddddddd
                        dddddd.dddddddddddddddddddddddddd
                          dddd.dddddddddddddddddddddddddddd
                            dd.dddddddddddddddddddddddddddddd
                              .dddddddddddddddddddddddddddddddd
                              .00dddddddddddddddddddddddddddddddd
                              .0000dddddddddddddddddddddddddddddddd
                              .000000dddddddddddddddddddddddddddddddd
```

The summation represented can be divided into two parts:

- The addition of the 15 integer parts.
- The addition of an infinite series of fractional parts.

The integer portion of the calculation can be implemented in a straightforward manner as a series of shifts and adds. The fractional portion is more problematic. Clearly an infinite series of additions is out of the question. On the other hand an exact result is required which can be correctly rounded towards zero.

Consider the first two bit positions after the binary point. Sixteen pairs of binary digits are summed. Now consider the next two bit positions. The same 16 pairs of digits are summed. Furthermore this same summation is repeated indefinitely. More formally, consider the 32-bit dividend as 16, base four digits, q_i. The fractional portion of the quotient, R, can be computed using the following series:

$$R = 1/4 \sum_{i=0}^{i=15} q_i + 1/16 \sum_{i=0}^{i=15} q_i + 1/64 \sum_{i=0}^{i=15} q_i + ...$$

Since the above is a standard geometric progression, R is given by:

$$R = \sum_{i=0}^{i=15} q_i / 3$$

At first sight we have simply returned to our original problem. However, since each bit represents only a single base four digit, the above calculation can be performed by modifying a two-bit adder to act as a single digit modulo 3 adder. Furthermore, the modulo 3 additions can be combined with the additions which compute the integer portion of the division.

Division by three can be implemented using the hardware shown in Fig1. A standard 32-bit ALU is used to compute Rdividend/3. The quotient is accumulated in the right-hand ALU input register, while appropriately shifted portions of the dividend are added from the left-hand input register. The following 17 steps are required:

```
1)     ADDC Rquot,Rdividend,#1;          MOV Rdividend,Rdividend(LSR#2);
           ( Carry In <-- C1 )
2-15)  ADDC Rquot,Rdividend,Rquot;       MOV Rdividend,Rdividend(LSR#2);
           ( Carry In <-- C1 )
16)    ADDC Rquot,Rdividend,Rquot;
           ( Carry In <-- C1 )
17)    MOV Rquot,Rquot(LSR#2);           /* remove fractional part */
```

The integer portion of the addition is carried out in the top 30 bits of the ALU in a straightforward manner. During the first step the partial quotient is set to the 30 most significant bits of the dividend. Since each step also performs a 2-bit logical shift right on the dividend, each subsequent addition instruction adds in a new right shifted dividend as required.

The fractional part of the computation is performed in parallel using the two least significant bits of the ALU. Thus a binary point notionally exists between bit two and one (zero being the least significant bit). The two least significant bits of the ALU are used to perform division by three. Modulo 3 division is achieved by maintaining a bias of one on these two bits throughout the computation. As a result of this bias whenever a total of three or more is accumulated, a carry will be passed from bit one into the integer portion of the calculation. These carries represent the integer portion of the required division by three and must be accumulated in the final total. In the code sequence an initial bias is added to the partial quotient in the first instruction. This bias is then reasserted whenever an ALU addition generates a carry from bit one. Reassertion is achieved by feeding C1, the carry from bit one, directly into the least significant bit of the ALU as Carry In. This middle-around carry mechanism ensures that the total in the two least significant ALU bits is always biased and results in modulo 3 addition being performed.

After 16 instructions, the correct truncated quotient is held in the 30 most significant bits of the quotient register. A further step is then required to right justify the answer. As long as the dividend is positive, a logical, not an arithmetic shift, is required.

Thirty-two distinct operations are involved. Alternatively, if the shift and add operations are combined in a single instruction, a total of 17 instructions is required. Since 33 shift and add instructions are required to implement the general-purpose non-restoring division algorithm, the number of cycles and operations required has been effectively halved.

In iHARP the required middle-around carry mechanism is incorporated in a DMSTEP (Division/Multiply Step) instruction primitive with the following syntax:

```
DMSTEP(i) Rdst,Rsrc1(ASR#n),Rsrc2
```

As well as adding the right-shifted register operand to the second register operand, DMSTEP ensures that the ith ALU carry bit is fed into the least significant ALU bit position as Carry In.

The execution time is further reduced by distributing the computation over all four pipelines. In the following code the partial quotient is accumulated in three pipelines while the fourth pipeline is used to compute a new right-shifted dividend for later use.

```
1)   DMSTEP(1) Rquot1,Rdividend,#1;
     DMSTEP(1) Rquot2,Rdividend(ASR#2),#0;
     DMSTEP(1) Rquot3,Rdividend(ASR#4),#0;       MOV Rdividend,Rdividend(ASR#6)
2-5) DMSTEP(1) Rquot1,Rdividend,Rquot1;
     DMSTEP(1) Rquot2,Rdividend(ASR#2),Rquot2;
     DMSTEP(1) Rquot3,Rdividend(ASR#4),Rquot3;   MOV Rdividend,Rdividend(ASR#6)
6)   DMSTEP(1) Rquot1,Rdividend,Rquot1;          /* final multiply cycle */
     DMSTEP(1) Rquot2,Rquot2,Rquot3;             /* combine sub-totals */
7)   DMSTEP(1) Rquot1,Rquot1,Rquot2;
8)   MOV Rquot,Rquot1(ASR#2);                    /* extract integer result */
```

As a result the execution time is reduced to only eight cycles, four times faster than the general-purpose algorithm.

5. Generalisation to Other Constants

The algorithm developed for division by three can be generalised to cater for division by other constants. Division by five, for example, is equivalent to multiplication by the recurring fraction 0.0011. The fractional portion of the calculation now requires a succession of modulo 15 additions to be performed. The four least significant bits of the ALU can be used to perform these additions providing the middle-around carry mechanism is extended to obtain Carry In from the carry generated from bit position three.

In general, for all K, where K is an odd integer, it can be shown that 1/K is a recurring fraction of the form 0.RRRetc, where R is of bit length less than K. In principle the division mechanism outlined can therefore be extended to any constant by extending the DMSTEP instruction so that it selects the required Carry In from a suitable range of middle-around carries. Execution times for various small constants are given in Table 1.

This paper assumes that single length ALUs are used throughout. For many constants single length working makes it necessary to separate the fractional and integer portions of the calculation for at least part of the division process. This restriction accounts for the increased number of operations required for some divisors in Table 1. Since on iHARP these additional steps map into parallel operations, their impact on the final execution time is minimal. However, in a single ALU machine it would be possible to reduce the number of shift/reduce steps by extending the length of the ALU.

The above ideas can be extended to cater for signed dividends by conditionally negating the dividend at the beginning and the result at the end. The execution time would then be extended by three cycles. iHARP uses the more elegant alternative of extending the algorithm to cater for signed dividends. This extension requires the use of arithmetic shifts throughout (as already assumed above) and a rounding correction for negative results. This method requires only one or two additional cycles to cope with negative dividends.

The DMSTEP instruction requires a carry from the middle of the ALU carry path to be fed back into the ALU as Carry In, the carry into the least significant bit. Clearly this mechanism must not be allowed to extend the CPU cycle time. Implementation is simplified if it is noted that a middle-around carry can never propagate beyond the point at which it was generated. Therefore, in an ALU using a simple ripple carry mechanism, the carry path is unlikely to be extended. If the ALU uses a carry propagate adder with full carry lookahead, the middle-around carries must be handled with more care. In this case our studies indicate that middle-around carries generated within the first 12 bits of a 32-bit ALU can be successfully fed back into the ALU without increasing the addition time. However, middle-around carries from higher bit positions must be saved in the carry flag and re-used in a subsequent DMSTEP cycle. Saving the carry for later re-use avoids any timing restraints but typically increases division execution times by one cycle.

6. Conclusions

The neglected topic of division by small constants has been examined in the context of the iHARP processor development. It has been shown that an arithmetic shift can be used to implement division by powers of two in a single cycle by providing a small amount of additional logic to adjust the incorrect result normally generated for negative numbers. A Divide/Multiply Step primitive has also been introduced to implement division by constants which are not powers of two. Using this primitive, division execution times on iHARP have been reduced to about a third of the time taken by a general-purpose division routine.

Acknowledgements

The author would like to thank Rod Adams, Paul Findlay and Fleur Williams for their helpful comments on this paper. He also gratefully acknowledges the support of the rest of the HARP team, in particular Sue Gray, Gordon Green and Liang Wang from Computer Science and Brian Johnson and Simon Trainis from Electrical Engineering. He would also like to thank Dr. S. L. Stott, Professor L.C.W. Dixon and J. A. Davis for their support.

The HARP project is supported by SERC Research Grant GR/F88018.

References

[Magen87] Magenheimer,J.M., Peters,L., Pettis,K. and Zuras,D. "Integer Multiplication and Division on the HP Precision Architecture", ASPLOS II, Palo Alto, October 1987.

[Steven89] Steven,G.B., Gray,S.M. and Adams,R.G. "HARP: A Parallel Pipelined RISC Processor", Microprocessors and Microsystems, Vol.13, No.9 (November 1989), pp579-587.

[Adams90a] Adams,R.G., Gray,S.M. and Steven,G.B. "Utilising Low Level Parallelism in General Purpose Code: The HARP Project", accepted for publication by Microprocessing and Microprogramming.

[Adams90b] Adams,R.G. and Steven,G.B. "A Parallel Pipelined Processor with Conditional Instruction Execution", Submitted for publication to Computer Architecture News.

[Steven89] Steven,G.B. "A Novel Effective Address Calculation Mechanism for RISC Microprocessors", SIGARCH, September 1988, pp150-6.

[Gosling80] Gosling,J.B. "Design of Arithmetic Units for Digital Computers", Macmillan, 1980.

[Steele77] Steele,G.L., "Arithmetic Shifting Considered Harmful", ACM SIGPLAN Notices, November 1977, pp61-68.

Fig 1 Hardware to Implement Division by Constants

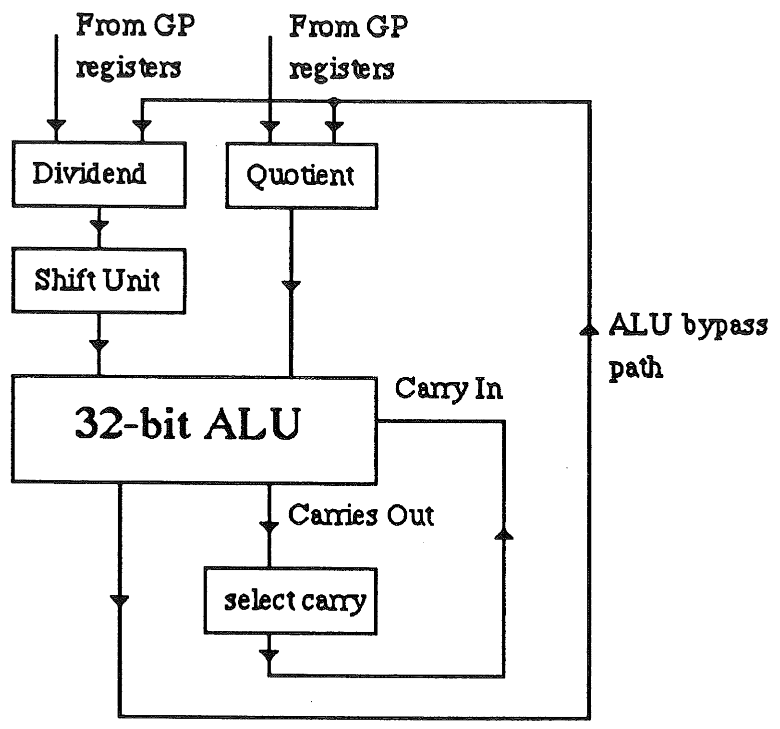

Table 1 Division by Small Constants

Divisor	Middle Around Carry	Shift/Add Operations	iHARP Cycles
variable	n/a	33	33
3	1	17	8
5	3	19	9
6	1	17	8
7	2	12	7
9	5	24	9
10	3	19	9
11	10	28	10
12	1	17	8
14	2	12	7
15	3	9	6

HATFIELD POLYTECHNIC
College Lane
Hatfield Herts
AL10 9AB

Integer Division by Small Constants

Technical Report No 113

Gordon B Steven

Submitted to 10th IEEE Symposium on Computer Arithmetic

November 1990

Integer Division by Small Constants

Abstract
Integer division is considered within the context of the development of iHARP, an integrated circuit version of HARP, the HAtfield RISC Processor architecture. The paper demonstrates that execution times for division by small constants can be significantly reduced. Two cases are considered, division by powers of two and division by other small constants. In each case specialised instruction primitives are introduced to reduce the execution time.

If an arithmetic shift right instruction is used to implement division by a power of two, an incorrect result is produced for negative dividends. The first iHARP primitive corrects the shift result and allows division by powers of two to be performed in a single machine cycle. The second iHARP primitive allows division by other small constants to be implemented efficiently as a finite series of multiplication steps. Execution time is reduced both by decreasing the number of operations involved and by exploiting the parallel nature of the iHARP architecture.

Key words
Integer division RISC VLIW

1. Introduction

This paper considers the division of integers by constants as a separate design problem. The aim is to reduce the execution time of division in the many cases where the divisor is known at compile time. Unlike the general problem of integer division, this topic has been neglected in the literature with the notable exception of a recent paper from the HP Precision Architecture group [Magen87].

The work described was undertaken during the course of the development of iHARP, a VLIW (Very Long Instruction Word) processor currently being developed at Hatfield Polytechnic [Steven89][Adams90a]. In each processor cycle iHARP fetches four integer instructions from the instruction cache and executes them in parallel in four distinct pipelines. An iHARP processor is therefore capable of performing four ALU operations in parallel. The ALU configuration for each pipeline is shown in Fig1. The shift unit in front of the ALU is capable of shifting one of the ALU operands a small number of bit positions left or right. Other distinctive features of iHARP include the conditional execution of all instructions [Adams90b] and ORed indexing [Steven88].

The paper first outlines the implementation of general-purpose division on iHARP. The use of arithmetic shift right instructions to implement division by powers of two is then reviewed, and the additional hardware provided by iHARP to allow divisions by powers of two to be executed in a single cycle is described.

An algorithm for division by three is then developed and its hardware implementation on iHARP described. A subsequent section describes how this algorithm can be generalised to other constants. Finally comparative execution times for division by various small constants are provided. The paper demonstrates that execution times for division by small constants can be significantly reduced. The improvement comes from two sources. Firstly, the number of operations is reduced by converting the division into a multiplication. Secondly, the algorithm employed is able to exploit the multiple parallel pipelines provided by iHARP.

2. General-Purpose Division on iHARP

General-purpose division on iHARP uses a non-restoring division algorithm [Gosling80]. The quotient is produced one bit at a time using a single length add or subtract in conjunction with a double length shift.

The algorithm is implemented using a DSTEP instruction primitive which provides the required double length shift and add/subtract step. The shift operation uses the shift unit placed immediately before the ALU in each pipeline (Fig1). An unsigned, 32-bit division operation is executed in 33 cycles. In the case of signed division the execution time increases to 36 machine cycles.

Each DSTEP instruction is executed by two adjacent pipelines acting in concert. The first pipeline performs the upper half of the double length shift and the add/subtract step while the second pipeline performs the lower half of the shift. A divide routine therefore requires the use of two pipelines but leaves the two remaining pipelines free to perform other operations including a second division. Four additional 32-bit shift registers would have been required to implement a DSTEP instruction in each of the four iHARP pipelines. These registers would have increased the machine state and would have required saving on context switches. This overhead is avoided in the solution adopted.

In addition iHARP provides special-purpose instructions to reduce the execution time of divisions where the divisor is known at compile time. These primitives provide the main subject of this paper.

3. Division by Powers of Two

It is well known that an arithmetic shift right only corresponds to division by a power of two if the operand is positive [Steele79]. Consider for example "-1 DIV 2". An arithmetic shift right of one will produce the answer minus one, whereas most programming languages expect the answer zero. The problem is that an arithmetic shift right rounds towards minus infinity while most languages follow FORTRAN and require integer division to round towards zero. The following algorithm must therefore be used to divide a signed integer, Ri, by N where N = 2 ** k:

```
IF Ri < 0 THEN Ri := Ri + (N-1);
Ri := Ri (ASR#k)
```

On iHARP this algorithm can be implemented using the following three sequential instructions:

```
1)  LTS B1,Ri,#0            /* B1:= Ri < #0 */
2)  TB1 ADD Ri,Ri,#N-1      /* IF B1=TRUE THEN Ri := Ri+N-1 */
3)  MOV Rj,Ri(ASR#k)        /* Rj := Ri shifted right k bits arithmetically */
```

Alternatively by executing instructions in two parallel pipelines, the execution time can be reduced to two cycles:

```
1)  LTS B1,Ri,#0;            ADD Rj,Ri,#N-1         /* speculatively adjust dividend */
2)  FB1 MOV Rj,Ri(ASR#k);    TB1 MOV Rj,Rj(ASR#k)   /* use B1 to select correct result */
```

Neither method is ideal, the first not only takes three instructions but destroys the original operand, while the second requires four short instructions. An alternative algorithm performs the required correction after the shift operation.

```
1)  MOV Rj,Ri(ASR#k)   /* carry flag set if Ri < 0 and a one shifted out during shift */
2)  ADDC Rj,Rj,#0      /* correct rounding */
```

In the first instruction the carry flag is set if the shift operand is negative and if any bit discarded during the shift operation is one. An add with carry instruction then uses the carry flag to correct the rounding in the case of negative dividends. On iHARP these two operations are combined in a single primitive. The shift unit in front of the ALU performs the arithmetic shift, while the ALU performs the correction required for negative operands.

```
1)  ADDC Rj,Ri(ASR#k),#0     /* Carry In = 1 if Ri < 0 and at least one "1" is discarded during
                                the shift  operation */
```

As a result divisions by powers of two are executed in a single processor cycle.

4. Division by Three

This section introduces the iHARP approach to division by constants which are not powers of two. Consider a 32-bit positive number being divided by three. This division is equivalent to multiplication by 0.0101etc where the "01" is repeated indefinitely. The start of the series of additions required is graphically represented below. Each letter represents a single digit in the 32-bit dividend. All unrepresented bit positions are zero.

```
dddddddddddddddddddddddddddddd.dd
  dddddddddddddddddddddddddddd.dddd
    dddddddddddddddddddddddddd.dddddd
      dddddddddddddddddddddddd.dddddddd
        dddddddddddddddddddddd.dddddddddd
          dddddddddddddddddddd.dddddddddddd
            dddddddddddddddddd.dddddddddddddd
              dddddddddddddddd.dddddddddddddddd
                dddddddddddddd.dddddddddddddddddd
                  dddddddddddd.dddddddddddddddddddd
                    dddddddddd.dddddddddddddddddddddd
                      dddddddd.dddddddddddddddddddddddd
                        dddddd.dddddddddddddddddddddddddd
                          dddd.dddddddddddddddddddddddddddd
                            dd.dddddddddddddddddddddddddddddd
                              .dddddddddddddddddddddddddddddddd
                              .00dddddddddddddddddddddddddddddddd
                              .0000dddddddddddddddddddddddddddddddd
                              .000000dddddddddddddddddddddddddddddddd
```

The summation represented can be divided into two parts:

- The addition of the 15 integer parts.
- The addition of an infinite series of fractional parts.

The integer portion of the calculation can be implemented in a straightforward manner as a series of shifts and adds. The fractional portion is more problematic. Clearly an infinite series of additions is out of the question. On the other hand an exact result is required which can be correctly rounded towards zero.

Consider the first two bit positions after the binary point. Sixteen pairs of binary digits are summed. Now consider the next two bit positions. The same 16 pairs of digits are summed. Furthermore this same summation is repeated indefinitely. More formally, consider the 32-bit dividend as 16, base four digits, q_i. The fractional portion of the quotient, R, can be computed using the following series:

$$R = 1/4 \sum_{i=0}^{i=15} q_i + 1/16 \sum_{i=0}^{i=15} q_i + 1/64 \sum_{i=0}^{i=15} q_i + ...$$

Since the above is a standard geometric progression, R is given by:

$$R = \sum_{i=0}^{i=15} q_i / 3$$

At first sight we have simply returned to our original problem. However, since each bit represents only a single base four digit, the above calculation can be performed by modifying a two-bit adder to act as a single digit modulo 3 adder. Furthermore, the modulo 3 additions can be combined with the additions which compute the integer portion of the division.

Division by three can be implemented using the hardware shown in Fig1. A standard 32-bit ALU is used to compute Rdividend/3. The quotient is accumulated in the right-hand ALU input register, while appropriately shifted portions of the dividend are added from the left-hand input register. The following 17 steps are required:

```
1)     ADDC Rquot,Rdividend,#1;         MOV Rdividend,Rdividend(LSR#2);
           ( Carry In <-- C1 )
2-15)  ADDC Rquot,Rdividend,Rquot;      MOV Rdividend,Rdividend(LSR#2);
           ( Carry In <-- C1 )
16)    ADDC Rquot,Rdividend,Rquot;
           ( Carry In <-- C1 )
17)    MOV Rquot,Rquot(LSR#2);          /* remove fractional part */
```

The integer portion of the addition is carried out in the top 30 bits of the ALU in a straightforward manner. During the first step the partial quotient is set to the 30 most significant bits of the dividend. Since each step also performs a 2-bit logical shift right on the dividend, each subsequent addition instruction adds in a new right shifted dividend as required.

The fractional part of the computation is performed in parallel using the two least significant bits of the ALU. Thus a binary point notionally exists between bit two and one (zero being the least significant bit). The two least significant bits of the ALU are used to perform division by three. Modulo 3 division is achieved by maintaining a bias of one on these two bits throughout the computation. As a result of this bias whenever a total of three or more is accumulated, a carry will be passed from bit one into the integer portion of the calculation. These carries represent the integer portion of the required division by three and must be accumulated in the final total. In the code sequence an initial bias is added to the partial quotient in the first instruction. This bias is then reasserted whenever an ALU addition generates a carry from bit one. Reassertion is achieved by feeding C1, the carry from bit one, directly into the least significant bit of the ALU as Carry In. This middle-around carry mechanism ensures that the total in the two least significant ALU bits is always biased and results in modulo 3 addition being performed.

After 16 instructions, the correct truncated quotient is held in the 30 most significant bits of the quotient register. A further step is then required to right justify the answer. As long as the dividend is positive, a logical, not an arithmetic shift, is required.

Thirty-two distinct operations are involved. Alternatively, if the shift and add operations are combined in a single instruction, a total of 17 instructions is required. Since 33 shift and add instructions are required to implement the general-purpose non-restoring division algorithm, the number of cycles and operations required has been effectively halved.

In iHARP the required middle-around carry mechanism is incorporated in a DMSTEP (Division/Multiply Step) instruction primitive with the following syntax:

```
DMSTEP(i) Rdst,Rsrc1(ASR#n),Rsrc2
```

As well as adding the right-shifted register operand to the second register operand, DMSTEP ensures that the ith ALU carry bit is fed into the least significant ALU bit position as Carry In.

The execution time is further reduced by distributing the computation over all four pipelines. In the following code the partial quotient is accumulated in three pipelines while the fourth pipeline is used to compute a new right-shifted dividend for later use.

```
1)   DMSTEP(1) Rquot1,Rdividend,#1;
     DMSTEP(1) Rquot2,Rdividend(ASR#2),#0;
     DMSTEP(1) Rquot3,Rdividend(ASR#4),#0;          MOV Rdividend,Rdividend(ASR#6)
2-5) DMSTEP(1) Rquot1,Rdividend,Rquot1;
     DMSTEP(1) Rquot2,Rdividend(ASR#2),Rquot2;
     DMSTEP(1) Rquot3,Rdividend(ASR#4),Rquot3;     MOV Rdividend,Rdividend(ASR#6)
6)   DMSTEP(1) Rquot1,Rdividend,Rquot1;            /* final multiply cycle */
     DMSTEP(1) Rquot2,Rquot2,Rquot3;               /* combine sub-totals */
7)   DMSTEP(1) Rquot1,Rquot1,Rquot2;
8)   MOV Rquot,Rquot1(ASR#2);                      /* extract integer result */
```

As a result the execution time is reduced to only eight cycles, four times faster than the general-purpose algorithm.

5. Generalisation to Other Constants

The algorithm developed for division by three can be generalised to cater for division by other constants. Division by five, for example, is equivalent to multiplication by the recurring fraction 0.0011. The fractional portion of the calculation now requires a succession of modulo 15 additions to be performed. The four least significant bits of the ALU can be used to perform these additions providing the middle-around carry mechanism is extended to obtain Carry In from the carry generated from bit position three.

In general, for all K, where K is an odd integer, it can be shown that 1/K is a recurring fraction of the form 0.RRRetc, where R is of bit length less than K. In principle the division mechanism outlined can therefore be extended to any constant by extending the DMSTEP instruction so that it selects the required Carry In from a suitable range of middle-around carries. Execution times for various small constants are given in Table 1.

This paper assumes that single length ALUs are used throughout. For many constants single length working makes it necessary to separate the fractional and integer portions of the calculation for at least part of the division process. This restriction accounts for the increased number of operations required for some divisors in Table 1. Since on iHARP these additional steps map into parallel operations, their impact on the final execution time is minimal. However, in a single ALU machine it would be possible to reduce the number of shift/reduce steps by extending the length of the ALU.

The above ideas can be extended to cater for signed dividends by conditionally negating the dividend at the beginning and the result at the end. The execution time would then be extended by three cycles. iHARP uses the more elegant alternative of extending the algorithm to cater for signed dividends. This extension requires the use of arithmetic shifts throughout (as already assumed above) and a rounding correction for negative results. This method requires only one or two additional cycles to cope with negative dividends.

The DMSTEP instruction requires a carry from the middle of the ALU carry path to be fed back into the ALU as Carry In, the carry into the least significant bit. Clearly this mechanism must not be allowed to extend the CPU cycle time. Implementation is simplified if it is noted that a middle-around carry can never propagate beyond the point at which it was generated. Therefore, in an ALU using a simple ripple carry mechanism, the carry path is unlikely to be extended. If the ALU uses a carry propagate adder with full carry lookahead, the middle-around carries must be handled with more care. In this case our studies indicate that middle-around carries generated within the first 12 bits of a 32-bit ALU can be successfully fed back into the ALU without increasing the addition time. However, middle-around carries from higher bit positions must be saved in the carry flag and re-used in a subsequent DMSTEP cycle. Saving the carry for later re-use avoids any timing restraints but typically increases division execution times by one cycle.

6. Conclusions

The neglected topic of division by small constants has been examined in the context of the iHARP processor development. It has been shown that an arithmetic shift can be used to implement division by powers of two in a single cycle by providing a small amount of additional logic to adjust the incorrect result normally generated for negative numbers. A Divide/Multiply Step primitive has also been introduced to implement division by constants which are not powers of two. Using this primitive, division execution times on iHARP have been reduced to about a third of the time taken by a general-purpose division routine.

Acknowledgements

The author would like to thank Rod Adams, Paul Findlay and Fleur Williams for their helpful comments on this paper. He also gratefully acknowledges the support of the rest of the HARP team, in particular Sue Gray, Gordon Green and Liang Wang from Computer Science and Brian Johnson and Simon Trainis from Electrical Engineering. He would also like to thank Dr. S. L. Stott, Professor L.C.W. Dixon and J. A. Davis for their support.

The HARP project is supported by SERC Research Grant GR/F88018.

References

[Magen87] Magenheimer,J.M., Peters,L., Pettis,K. and Zuras,D. "Integer Multiplication and Division on the HP Precision Architecture", ASPLOS II, Palo Alto, October 1987.

[Steven89] Steven,G.B., Gray,S.M. and Adams,R.G. "HARP: A Parallel Pipelined RISC Processor", Microprocessors and Microsystems, Vol.13, No.9 (November 1989), pp579-587.

[Adams90a] Adams,R.G., Gray,S.M. and Steven,G.B. "Utilising Low Level Parallelism in General Purpose Code: The HARP Project", accepted for publication by Microprocessing and Microprogramming.

[Adams90b] Adams,R.G. and Steven,G.B. "A Parallel Pipelined Processor with Conditional Instruction Execution", Submitted for publication to Computer Architecture News.

[Steven89] Steven,G.B. "A Novel Effective Address Calculation Mechanism for RISC Microprocessors", SIGARCH, September 1988, pp150-6.

[Gosling80] Gosling,J.B. "Design of Arithmetic Units for Digital Computers", Macmillan, 1980.

[Steele77] Steele,G.L., "Arithmetic Shifting Considered Harmful", ACM SIGPLAN Notices, November 1977, pp61-68.

Fig 1 Hardware to Implement Division by Constants

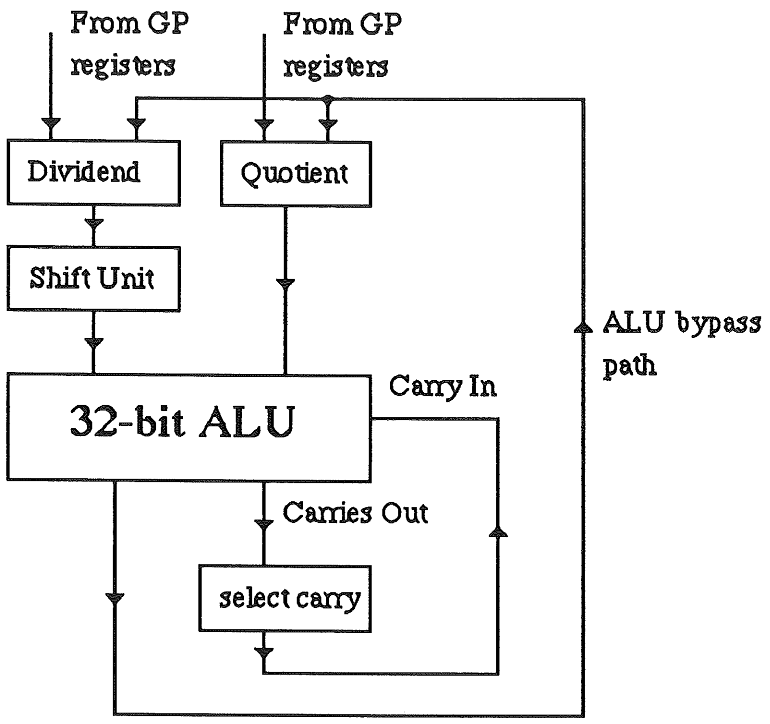

Table 1 Division by Small Constants

Divisor	Middle Around Carry	Shift/Add Operations	iHARP Cycles
variable	n/a	33	33
3	1	17	8
5	3	19	9
6	1	17	8
7	2	12	7
9	5	24	9
10	3	19	9
11	10	28	10
12	1	17	8
14	2	12	7
15	3	9	6

HATFIELD POLYTECHNIC
College Lane
Hatfield Herts
AL10 9AB