Integer Multiplication and Division on the HP Precision Architecture

Daniel J. Magenheimer, Liz Peters, Karl Pettis, Dan Zuras

Hewlett-Packard Co.
Information Technology Group
19447 Pruneridge Ave, Cupertino CA

ABSTRACT

In recent years, many architectural design efforts have focused on maximizing performance for frequently executed, simple instructions. Although these efforts have resulted in machines with better average price/performance ratios, certain complex instructions and, thus, certain classes of programs which heavily depend on these instructions may suffer by comparison. Integer multiplication and division are one such set of complex instructions. This paper describes how a small set of primitive instructions combined with careful frequency analysis and clever programming allows the Hewlett-Packard Precision Architecture integer multiplication and division implementation to provide adequate performance at little or no hardware cost.

1. Introduction

Many recent general purpose machine architectures (e.g. [Rad82,Pat82]) have been designed around one fundamental tenet: by concentrating effort on a few frequently executed, simple instructions, average performance can be increased and at the same time hardware costs can be reduced. Many published papers [Hen82,Neu79] contain instruction distributions which are ordered by frequency. The literature largely agrees that well designed memory access instructions and low overhead branches (both conditional and unconditional) are crucial to any machine design. Arithmetic, boolean and procedure call operations are also important.

Further down the list, near the bottom, are the more complex instruction classes: floating point, decimal, large block moves and integer multiplication and division. Does the relative infrequency of these instructions imply that their implementation is unimportant? Hardly. Machine architects must avoid the tendency to either overdesign these — which results in costly additional (and largely unnecessary) hardware or increased cycle time; or to underdesign it, in which case the instructions become weak points awaiting exercise and abuse by programs and benchmarks which depend on reasonable performance for these functions. The analysis and work which allowed these tendencies to be avoided for the implementation of integer multiplication and division in the Hewlett-Packard Precision Architecture are the subject of this paper.

2. Overview

Uses of Multiplication and Division[1]. Most programs use multiplication and/or division many times, either directly or indirectly. Almost all high level languages directly support these operations with an explicit operator (e.g., '*' and '/') and almost all support constructs that implicitly require a multiplication or division to occur. For example, in C:

```
a = structureA[x][y].b;
```

requires two multiplications, namely

```
x * y * sizeof(structureA)
```

while

```
diff = structureB[x] - structureB[y];
```

requires a division for the implied operation:

```
(&structureB[y] - &structureB[x]) / sizeof(structureB)
```

[1] For the remainder of the text, the references to multiplication and division are of the integer variety.

Languages such as FORTRAN, where matrix ranks can be passed as parameters, may have large numbers of implicit multiplications by variables.

Clever compilers can reduce the number of multiplications in a program by using a technique called "strength reduction". Strength reduction is the practice of replacing multiplications by additions and additions by increments wherever possible, since they are less costly than multiplications. For example:

```
for (i=0; i<10; i=i+1)
    j = j + i*15;
```

In this simple example the multiplication by 15 can be replaced by an addition of 15, since the multiplication results form an arithmetic progression.

In many cases, primarily if the induction variable is used in both a subscript expression and a non-subscript expression, this optimization is difficult or impossible to perform. Furthermore, optimizations may be inadvertently defeated by the use of a global variable as a loop counter or by careless goto's. Since programmers are not always aware of these problems, they may unwittingly force the execution of a large number of multiplications and still expect reasonable performance.

Divisions may appear to be affected negatively by efficient optimization. Since there is rarely an opportunity for an optimizer to remove a division, the number of divisions in a program remains constant while the total number of instructions executed is reduced. As a result the percent of the time a program spends doing divisions may actually *increase!*

Frequency of Multiplication and Division. Multiplication and division are relatively infrequent instructions. The Gibson mix [Gib70] (based on the IBM 704 architecture) estimates the frequency of multiplication at 0.6 percent and that of division at 0.2 percent. Other frequency measurement studies [Huc82,Neu79] vary widely with different benchmarks giving results between 0.0 and 2.5 percent for multiplication and 0.0 to 0.5 percent for division. Clearly, the frequency does not warrant special hardware consideration in the design of a machine.

Yet these instructions are executed frequently enough that a poor hardware or software implementation could significantly decrease a machine's performance, especially on programs which heavily use the operations. So what do we do? First, let us examine the way many machines implement the multiply and divide operations.

Usual Implementation - Multiplication. In [Boo51], Booth observed that one could speed up binary multiplication by replacing strings of zeros or ones in the multiplier with a number chosen from the digit set {-1,0,1}. This reduced the number of steps by at least a factor of two. The modern version of this method, often called Booth encoding, is usually implemented by cycling through the multiplier two bits at a time and adding to the accumulating product the multiplicand times a number in the digit set {-2,-1,0,1,2}. These implementations use 16 such cycles for a full 32-bit multiply. A side effect of this method is that one bit of state analogous to a carry must be retained between each step. A correction for signed multiplies is also necessary at the end.

Usual Implementation - Division. There are a few common methods of implementing integer division [Was82]. One of the simplest is a *restoring* division algorithm. Assuming the dividend and divisor are positive, this method starts by logically shifting the divisor left. This shifted divisor is then subtracted from the dividend. If this causes the dividend to become negative, the divisor is added back. Otherwise the partial quotient is incremented. After this step, the partial quotient is shifted left and the divisor is shifted right. These subtraction and shifting steps are repeated until the final quotient is generated.

The restoring algorithm can require both an addition and a subtraction for each bit of the quotient. To reduce the number of operations required most modern computers implement a *non-restoring* division algorithm. In this method, the shifted divisor is either subtracted from, or added to, the dividend depending on whether the previous result was positive or negative. The complement of the sign of the result is shifted into the quotient. Logically these bits are +1 or -1, instead of the usual 1 or 0, but there is a simple transformation done at the end to change the quotient to the usual binary representation. This algorithm requires a single addition (or subtraction) for each quotient bit.

3. HP Precision Support

Since one of the primary goals in the development of the Precision architecture was scalability, instructions were carefully chosen so as to be easily and efficiently implemented in both discrete logic and VLSI. Further, instructions which increased the basic cycle time of the machine or significantly affected the complexity of the datapaths in either technology were critically examined and often discarded.

In particular, at one point in the evolution of the architecture both a "multiply step" and a "divide step", similar to the ones described by Jouppi (cf. [Jou81]), were included in the instruction set. The Multiply Step implemented a two-bit Booth's algorithm, but required either a "three-read-port two-write-port" register file, or a special register (the HL registers in [Jou81]) with the

associated control and datapaths *and* several additional variations of the instruction to handle pre- and post-corrections.

The Divide Step also required the special register and its pipelining affected the critical path for the basic cycle time. As a result, the Multiply Step instruction was removed from the instruction set and the Divide Step instruction was vastly simplified. But rather than replacing them directly many-for-one, and thus suffering significant performance degradation on multiplication and division, a careful analysis of the basic operations being performed and their usage was undertaken.

Operand Frequency Analysis. Several analyses have been published on the frequency and type of operands used by arithmetic instructions. Though these studies rarely focus on multiplication and division, some information on these operations is often included. Interesting observations include:

- Some 91% of multiplication operations include one compile-time constant (immediate) operand. [Neu79]
- Immediate operand values for all operations tend to be small. [Hen82,Swe82]
- Non-immediate operand values for multiplication tend to be small. [Luk86]
- "Standard" (32-bit result) multiplication occurs much more frequently than extended (64-bit result) multiplication. [Cla82]

To avoid taking these results out of context, we performed our own trace analyses (cf. [Luk86]) for independent confirmation.

In the spirit of the philosophy espoused by RISC architects [Pat85,Bir85] we should attempt to optimize the most frequent cases, even at the cost of decreasing performance of the less frequent cases. The absence of well-defined support instructions encourages us to exploit these observations. By recognizing the inherent non-uniformity and special cases of operands (and results), we may be able to increase the overall performance. This is precisely the approach used for multiplication and division in the Precision architecture which will be examined in the next few sections. But first, we need to understand the basic building blocks from which minimal support is obtained.

4. The Instructions

A byte-oriented machine may have difficulty accessing elements in an array of a datatype of 16, 32 or 64 bits without some form of an indexed load instruction. The Precision architecture has such an instruction and its implementation necessitates a "pre-shifter" datapath before one of the inputs into the ALU. This allows byte addresses to be calculated for half-word, word and double-word addresses in array indexing. This pre-shifter also serves in a second capacity: it allows for one of the operands in a normal addition to be shifted before an addition takes place. We refer to this operation generically as the *shift and add* operation. As we shall see, these simple operations prove to be quite powerful.

Shift and Add Instructions. The Precision architecture defines a set of shift and add instructions. First, the shift amount may be either one, two or three bits. Then for each of these shift amounts, there is a variation which traps if an overflow occurs and one that does not.

One might suspect that proper overflow detection requires a full 35-bit addition to be performed — an expensive proposition especially in a discrete implementation. Instead, a normal 32-bit addition is performed and overflow is detected by a circuit that compares the sign bits of the operands with the shifted out sign bits and the sign bit of the result. Although this does not allow for proper overflow detection if the operands are of different signs, this case hardly ever arises and, in practice, can easily be avoided.

Divide Step Instruction. The divide step instruction performs part of a single-bit nonrestoring divide step based on a set of input conditions and produces a set of result conditions. It calculates one bit of the quotient when a register containing the least significant word of a 64-bit partial dividend is divided by a register containing the (32-bit) divisor and leaves the partial remainder in the target register. When combined with an add with carry operation on the most significant word of the partial dividend, and both are repeated 32 times, a full division is obtained.

This operation differs from the one described by Jouppi in two important respects: First, since the most significant word of the dividend is used only in the second instruction (add with carry) of each step, the divide step instruction requires only two register reads and one register write and no additional datapaths for a special register. Second, since the determination of whether the following step does an addition or subtraction is computed late in the current step and is required early in the following step, the pipelining of this flag (which we call the *V-bit*) may increase the basic cycle time of the machine. Since the second instruction of the two instruction step does not require the V-bit, our cycle time is not affected. So although we have doubled the number of instructions to compute a division, we have reduced the cost and removed a potential bottleneck in the performance of the machine.

A more precise description of these and other Precision architecture instructions can be found in [HP86].

5. Multiplication by Constants

Multiplication by constants on the Precision architecture may be thought of as a generalization of the notion of an *addition chain*. In [Knu81], an addition chain for the number n is defined as a sequence of integers

$$1 = a_0, a_1, a_2, \ldots, a_r = n$$

that are generated by the rule

$$a_i = a_j + a_k,$$

for some j,k such that $k \le j < i$ for all i in [1,r]. If the length of the shortest chain for n is denoted by $l(n)$, Knuth shows that asymptotically,

$$l(n) \le \lambda(n) + \lambda(\lambda(n)) + O(\lambda(n)\lambda(\lambda(\lambda(n)))/\lambda(\lambda(n))^2)$$

where $\lambda(n) = \lfloor \lg(n) \rfloor$.

The Precision architecture allows us to construct a similar chain

$$a_{-1} = 0, a_0 = 1, a_1, a_2, \ldots, a_r = n$$

with instructions that implement the rules[1]

$$\begin{aligned} a_i &= a_j + a_k \\ a_i &= 2a_j + a_k \\ a_i &= 4a_j + a_k \\ a_i &= 8a_j + a_k \\ a_i &= a_j - a_k \\ a_i &= a_j << n \end{aligned}$$

for some j,k,n such that $k \le j < i$ and $n < 31$. (a_{-1} is included as the Precision architecture allows access to a register which always contains the value zero.) For example, the number 10 can be constructed by the chain

$$a_0 = 1, a_1 = 5, a_2 = 10$$

Thus, multiplication by 10 can be done in two steps by

```
r = 4s + s
r = r + r
```

All multiplication by constants may be constructed in this way.

Chain Rules. With these instructions we can generate the numbers -1, 3, 5, 9, and any power of two in a single step. Therefore, given a chain for generating n of length $l(n)$, we can generate $-n$, $3n$, $5n$, $9n$, and $2^k n$ in no more than one more step. Since the operand is always left untouched in a multiplication by constant, we can also generate $1-n$, $n-1$, $n+1$, $n+2$, $n+4$, $n+8$, $2n+1$, $4n+1$, and $8n+1$ in no more than one more step.

[1] The notation << and >> refer to left and right shifts, respectively.

These are some of the simple rules that are used to generate chains. There are more complex ones. For example, numbers of the form $(2^k-1)n$ may be generated from a chain for n in no more than two more steps by appending the sequence

```
t = n<<k
t = t - n
```

onto the end.

We have written a rule-based program that searches for chains in this manner. The chains generated by this program compare quite well to those generated by a program that exhaustively searches for all possible chains. They are generated in much less time and, for all numbers less than 10000, are of minimal length in all but 12 cases. Thus, by remembering these exceptions, minimal length chains may be generated in reasonable time for most constants.

Overflow. As has been mentioned, provision has been made in the Precision architecture for the detection of overflow on multiplication. A properly constructed chain can thus automatically detect overflow for those languages that require it. Such a chain must be monotonic and contain only add or *shift and add* instructions (as this would require the use of an instruction that does not detect overflow).

A chain is said to be monotonic if

$$a_i < a_j \text{ for all } i < j.$$

For example, one chain for multiplication by 15 is

```
a1 = a0<<4
a2 = a1 - a0
```

but this chain contains a shift by 4 and is not monotonic. A monotonic chain for multiplication by 15,

```
a1 = 2a0 + a0
a2 = 4a1 + a1
```

can, however, be made to check for overflow.

Such chains are not always as short as they might be. A similar chain for 31,

```
a1 = a0<<5
a2 = a1 - a0
```

cannot be made monotonic in two steps. A three step chain for 31, such as

```
a1 = 2a0 + a0
a2 = 4a1 + a1
a3 = 2a2 + a0
```

must be constructed if overflow is to be detected. Thus, there may be a penalty incurred for the detection of overflow that languages such as Pascal may have to pay that languages such as C may not. On the other hand, the penalty is bounded. This is

because overflow can always be detected by bounds checking the operand before using a chain that does not detect overflow.

Register Use. The number of temporary registers needed for intermediate results may also be traded off with the length of the chain. However, since a minimal length chain typically uses very few temporaries, this has not been a problem. In fact, the only numbers less than 100 that need a temporary at all in their minimal chains are 59, 87, and 94.

It is possible for a multiplication to be done in place for some constants (specifically, numbers of the form $2^i 3^j 5^k$). However, by convention, the source register for a multiplication by constant is left untouched by the multiplication. Thus, if a chain always uses the previously constructed number or the number *1* in each step, it need not use a temporary register for an intermediate result.

For example, a minimal chain for 59,

```
t = 2s + s
r = 2t + s
r = 8r + t
```

uses one temporary, while this chain

```
r = s + s
r = 8r + s
r = 2r + r
r = 8s + r
```

does not. In this case, a longer chain is not necessarily slower than a chain without temporaries. If the use of a temporary register requires that a store and load take place, a longer chain is to be preferred.

Some Results. This effort has shown some interesting results. For example, Figure 1 shows a table of the least numbers that may be generated in *r* steps for $1 \le r \le 6$.

r	least values of *n* such that *l(n)=r*
1	2,3,4,5,8,9,16,32,64,128,256,512
2	6,7,10,11,12,13,15,17,18,19,20,21
3	14,22,23,26,28,29,30,35,38,39,42
4	58,78,86,92,106,110,114,115,116
5	466,474,618,622,678,683,686,687
6	3802,4838,5326,5519,5534,5550

Figure 1.

The first *n* such that $l(n)=7$ is not yet known as execution of exhaustive searches for chains of length 7 is prohibitively time consuming. It is known that there are no such *n* less than 10000.

Even so, some conjectures can be made. It is certain that the behavior of the function *c(r)*, defined as the first *n* such that $l(n)=r$, is at least exponential. The first 6 entries suggest that is might be super exponential. If true, this would imply that the asymptotic behavior of $l(n)$ is better than $log(n)$.

6. Multiplication by Variables

Despite the simplicity of the multiply-by-constant techniques, there are still many programs with a large number of multiplications where neither operand is available at compile time. We refer to this type of multiply as a multiply-by-variable. The discussion in this section is limited to the previously-defined "standard" multiply (32-bit result). An efficient implementation of extended multiply (64-bit result) is an area of our current research.

The Basics. With no multiply step instruction to work with, it is necessary to understand the basics more thoroughly. A multiplication algorithm can be constructed using several simple operations including:

- an ADD operation,
- an arithmetic right shift operation ; and
- a bit test (or test for odd) instruction.

With this simple algorithm, the bits of the multiplier are examined one at a time, from the least significant to the most significant. For each bit, if the bit is set, the multiplicand is added to the result register. If the bit is clear, the result is left alone. In any case, the multiplicand is then doubled before the next multiplier bit is examined.

Though this simple algorithm was never considered for implementation in the Precision architecture, it is instructive both because it demonstrates several problems that must be overcome in any multiplication algorithm and because it approximates a "worst case" performance. The algorithm is shown in Figure 2.

```
tmp = mpy;
mpy = abs(mpy);
rslt = 0;
for (i = 32; i > 0; i=i-1) {
      if (mpy & 1)
            rslt = mcand + rslt;
      mpy = mpy >> 1;
      mcand = mcand + mcand;
}
if (tmp < 0)
      rslt = -rslt;
```

Figure 2.

First, notice that overflows can occur in at least two places: in the addition of the multiplicand to the result and in the addition of the multiplicand to itself. Since the semantics of most high-

level languages require that overflow be detected in multiplication, we must assume that it is possible to check for overflow in the additions. (This is, of course, always true since the same languages require overflow detection for addition.) This is of greater importance as the algorithm becomes more complex, as we must take care that all possible overflows are detected and, further, that overflow is not incorrectly reported due to intermediate calculations. The discussion also points out that, since some languages (such as C), do not report overflows on multiplies, we must provide two versions of the algorithm — one which detects overflow and one (possibly faster) which doesn't.

Second, notice the taking of the absolute value of the multiplier before the multiply occurs. A negative multiplier unnecessarily complicates the algorithm, so it is easier to take its absolute value, remember whether it was negative or not, and correct, if necessary, at the end. (These steps are ignored in the remaining examples in this section.) Note also that this is a good example of the earlier admonition regarding overflow: if the result of the multiplication is the most negative number (on a two's complement machine), either the absolute value operation, the final correction, or an intermediate calculation may report an overflow when it is possible that the result is perfectly representable.

Performance is obviously a concern too. In the Precision architecture instruction set, the algorithm in Figure 2 has a dynamic path of 167 (single cycle) instructions. A multiply which is more than two orders of magnitude slower than an add is clearly unacceptable. What can we do to speed it up?

A Simple Optimization. If we examine the distribution of possible multipliers, we note that a large quantity have an absolute value with several "leading" zeroes. Now our initial algorithm shifts the multiplier to the right one bit at a time, and, as soon as the shifted multiplier has only zeroes, the result is correct. So, we can take advantage of these facts by exiting the loop early if the shifted multiplier is zero. But do we gain anything by adding an instruction to a critical loop? Almost certainly. Our count indicates that we have increased our worst case to 192 instructions. But, if we assume that the absolute value of the set of multipliers is logarithmically distributed (a pessimistic guess further refined below), the loop will be executed on the average only half as many (16) times. The "average" multiplication now takes 103 instructions — a 40 percent improvement. We now have an algorithm with *data-dependent* performance.

Using The Power of the Shift and Add. With these basics in mind, we next turn to the Precision architecture's shift and add instructions. By examining several bits at a time from both the multiplier and multiplicand we can execute the loop fewer times and increase the algorithm's performance, as shown in Figure 3.

```
rslt = 0;
while (TRUE) {
      if (mpy & 1) rslt = mcand + rslt;
      if (mpy & 2) rslt = (mcand << 1) + rslt;
      if (mpy & 4) rslt = (mcand << 2) + rslt;
      if (mpy & 8) rslt = (mcand << 3) + rslt;
      mpy = mpy >> 4;
      if (mpy = = 0) break;
      mcand = mcand << 4;
}
```

Figure 3.

The overflow versions of the shift and add instructions allow us to detect intermediate overflows in adding the shifted multiplicand to the result. Also, two Shift Two and Add's neatly complete the left shift of the multiplicand at the end of the loop *and* check for overflows, all in two instruction cycles. Note, though, that several of the other overflow problems are yet to be solved.

The performance of this new algorithm is even more difficult to predict. First, we have reduced the worst case to 107 instructions. (The loop contains 13.) By applying the previous log-uniform distribution to four-bit "chunks", we will execute the loop an average of four times, resulting in an average instruction count of 55 instructions — *three times better* than our initial algorithm [1]. But the best is yet to come.

An Observation. Rather than examining the distribution of multipliers, we now concern ourselves with the distribution of both operands. It is rarely the case that both operands are large, say larger than 16 bits, because the result will be an overflow. If an overflow causes an abort, performance of the overflowing multiply is certainly of no concern. If the language allows overflows to be taken by exception handlers, the handler overhead (let alone the system overhead to get to and return from the handler) will overshadow the multiply's performance. And in languages with no overflow checking (e.g. C), the result after overflow is undefined. Surely, depending on multiplies which overflow is a questionable programming practice at best and, in any case, will result in a poorly performing program. Let us then no longer concern

[1]. The 4-bit right shift of the multiplicand assumes the machine implementation contains a barrel shifter for an efficient single-cycle shift. Several additional cycles are consumed otherwise.

ourselves with the performance of multiplies that result in an overflow.

We now note that, if the multiplication does not result in an overflow, at least one of the operands must be representable in less than 16 bits. Let us assume that it is the multiplier, for if it isn't, due to the commutativity of multiplication, we can simply switch the operands so that it is. With a simple test and possible swap, we have reduced the maximum number of times through the loop to four and the average to two. Our multiply is now 59 instructions, worst case, and 33 instructions on the average!

Applying Constant Multiplication. Our loop consumes 13 instructions per iteration — five are shifting the operands and loop overhead but eight are used for multiplying the multiplicand by a number between zero and 15 and adding it to the result. From the section on constant multiplication, we have seen that we can multiply any number by a positive constant less than 16 in at most three cycles (and usually two or one). By applying similar techniques, we can multiply the multiplicand by the four bits taken from the multiplier and add the result. To do this we must use a switch (case) table to isolate the 16 separate cases so that we can multiply by "constants". Our algorithm now appears in Figure 4.

```
rslt = 0;
while (TRUE) {
   switch(mpy & 15) {
      case 0: break;
      case 1: rslt = mcand + rslt; break;
      case 2: rslt = 2*mcand + rslt; break;
      case 3: rslt = 3*mcand + rslt; break;
        :
        :
      case 14: rslt = 14*mcand + rslt; break;
      case 15: rslt = 15*mcand + rslt; break;
   }
   mpy = mpy >> 4;
   if (mpy == 0) break;
   mcand = mcand << 4;
}
```

Figure 4.

A Few Additional Details. Study of the existing literature on operand distribution (supplemented by a few traces of our own) led us to believe that the lesser of the two operands was less than 16 more than half the time. We also observed that both operands were nearly always positive. Thus we optimized for:

- A single loop iteration;
- Quick exit for values of zero and one; and
- Positive operands.

Lastly, all entries in the switch statement are reduced to two instructions to reduce the algorithm's size (and the instruction cache misses suffered) and to accommodate the Precision architecture's "branch vectored" instruction.

In the final algorithm, overflow checking is completely and accurately handled and only one temporary variable is required. Many subtle techniques available in the Precision architecture instruction set have been used to remove unnecessary cycles. The result is an algorithm that is difficult to represent in a high-level language, and so it has been omitted. Figure 5 shows instruction counts for different ranges of operands.

min(|x|,|y|)	Best (including overhead)	Avg (including overhead)	Worst (including overhead)	%
0-15	10	15	23	60
16-255	20	24	34	20
256-4095	28	34	45	10
4096-46340	36	44	56	10

Figure 5.

If we assume the distribution of operands displayed in the table (and a distribution which has both operands positive about 90% of the time), we find that the algorithm performs the multiply in an average of less than 20 instructions. This is a eight-fold improvement over the original algorithm and compares favorably with Booth's algorithm implemented with a Multiply Step.

7. Division

On most machines, integer division takes significantly longer than addition or subtraction. This difference is compounded on machines such as the Precision architecture which do not have an explicit division instruction but instead perform division by calling a software routine. If many divisions are needed, this disparity in execution time can result in a bottleneck so we looked for ways to do specific division cases faster in order to improve the average performance of division.

A well known example is division by a power of 2. On a binary machine this becomes a matter of shifting. Even if some adjustments must be made to handle negative dividends, it is usually far faster to do an adjustment and then shift than to go through the general division algorithm for powers of 2. Under the Precision architecture, division by small powers of 2 can be done in one instruction when dealing with unsigned numbers and in three when dealing with signed numbers. For division by large powers of 2, this becomes one instruction for unsigned numbers and four for signed.

However, there are often cases where the division is by a known quantity that is not a power of 2. Such divisions are usually done by using the existing general purpose division algorithm or instruction. But if the divisor is known in advance, a different division method can be derived. This derived method may or may not be a worthwhile improvement depending on how fast the general division method is.

Division Problem Description. We are interested in the integer quotient of two numbers, x and y. It is assumed that y is a known constant and that x is a variable. For simplicity in the discussion that follows, assume that $x \geq 0$ and that $y > 0$. The extensions to handle negative x will be discussed later. Many programming languages such as C, Pascal, and Fortran define integer division to truncate towards 0. The derived method is designed to work in this way to yield results compatible with these programming languages. So we want to compute the function $q(x)$ such that

$$q(x) = \lfloor x / y \rfloor$$

Our technique is to find an inexpensive way to multiply x by the reciprocal of y. We choose a number z and use it to derive numbers a and b from y. The number a will be z times the reciprocal of y and the number b will be an adjustment. Then the new function q'(x) is defined:

$$q'(x) = (ax + b) / z$$

If we choose the right z, then the result of truncating $q'(x)$ to an integer will be equal to $q(x)$ for all legal values of x. In that case we can avoid the division by y in the evaluation of $q(x)$ and instead compute the truncated $q'(x)$. Note that we have replaced a division by a multiplication, an addition, and a different division. These replacement operations must be done in extended precision so that the full range of values for x can be handled. Nevertheless, for specific values of y, it will often be possible to find a, b and z such that all those operations are still significantly faster than the general division algorithm.

The Derived Method. Assume that the positive number z has been chosen (the criteria for choosing it will be discussed later). Let

$$a = \lfloor z / y \rfloor$$

Now we want to find b such that truncating $q'(x)$ will be equal to $q(x)$ over a sufficiently large range of x. In order for the two functions to be equal, if x is in the range

$$ky \leq x < (k+1)y$$

for some k, then we must have

$$k \leq q'(x) = (ax + b) / z < (k + 1)$$

If this is true for all k in some range $[0..K]$, then truncating $q'(x)$ equals $q(x)$ for x in the range $[0..(K+1)y-1]$. For 32-bit division we need $(K+1)y$ to be at least 2^{32}.

To find b, first define r to be the remainder of z when divided by y. In other words $r = z - ay$. Now $q'(x)$ is a non-decreasing function of x. Applying this to the upper bound yields

$$a((k + 1)y - 1) + b < (k + 1)z$$

which implies an upper limit for the value of b:

$$b < a + (k + 1)r$$

This is true for all k in the range $[0..K]$. In particular, it is true for $k = 0$, and this leads to an upper bound for b:

$$b < a + r$$

Similarly, using the minimum value of x to find a lower bound for b we have

$$k \leq (aky + b) / z$$

which implies that

$$kr \leq b$$

If r happens to be 0, then any b in the range $[0..(a-1)]$ will satisfy both bounds for all non-negative k. In this case we will choose $b = 0$ to eliminate the need for an addition. If r is not 0, then our choice for b implies a maximum bound for K, since $kr \leq b$ for all k. To allow K to be as large as possible, the best choice is $b = (a+r-1)$.

To summarize up to this point, we have proven that truncation of the $q'(x)$ function will yield the same result as the true division, $q(x)$, for x in the range $0 \leq x < (K+1)y$. The value of K depends on the choice of z, which also determines the numbers a and b.

Choosing z. The choice of z is constrained by a few factors. The first is that the resulting upper bound for x be sufficiently large to accommodate all values of interest. Since we are working with a 32 bit machine, we restricted x to be less than 2^{32}. The second constraint is a time constraint. The resulting multiple precision operations of multiplying x by a, adding b, and dividing by z must all be done faster than the general purpose division method.

In fact, there are an infinite number of choices for z which satisfy the first constraint. Any z which is a multiple of y, for example, will have the remainder r equal to 0. This implies that there is no upper bound on the values for x. The time constraint is more limiting, since it implies a bound on how complex the operations can be. Initially we have chosen to restrict the choice of z to a power of 2 (so that division by z and the resulting truncation would be fast). Values of z equal to a power of 2 plus or minus 1 have also been considered, since it is possible to divide by numbers of this form relatively quickly.

We also restricted ourselves to odd y, since division by an even y can be done by first dividing

the dividend by the largest power of 2 which divides y, and then dividing by the odd factor of y.

For small odd values of y, Figure 6 indicates the smallest power of 2 which satisfies the first constraint that $(K+1)y$ be at least 2^{32}. The values for a and $(K+1)y$ are in hexadecimal notation.

y	z	r	a	$(K+1)y$
3	2^{32}	1	55555555	100000002
5	2^{32}	1	33333333	100000004
7	2^{33}	1	49249249	200000006
9	2^{35}	5	E38E38E3	1999999A7
11	2^{36}	9	1745D1745	1C71C71D6
13	2^{35}	7	9D89D89D	124924938
15	2^{32}	1	11111111	10000000E
17	2^{32}	1	F0F0F0F	100000010
19	2^{36}	1	D79435E5	1000000012

Figure 6.

The intermediate result, $(ax + b)$, must be stored in a multiple precision fashion. In the cases listed, except for $y = 11$, the largest possible intermediate result will fit using two 32-bit words of precision. This result can be computed as $(x+1)a + (r\text{-}1)$ since $b = (a+r\text{-}1)$. If $r = 1$, the final addition can then be skipped.

The final observation is that the multiplication by a can be done by shifting and adding. This is particularly true when the bits forming the binary representation of a form a regular pattern. For example, if $y = 3$, then the computation of the result is shown in Figure 7. The initial add takes two instructions since the addition can carry into the higher precision word. Each shift and add pair operating on two word precision numbers takes 4 instructions, except the first pair which takes 3 instructions due to the existence of the Shift Two and Add instruction. The desired quotient is the higher precision word of the intermediate result since $z = 2^{32}$. The total sequence takes 17 instructions. This is still significantly slower than a single addition, but it is a factor of 3.5 times better than the general purpose algorithm.

```
tmp = x + 1
tmp = (tmp << 2) + tmp
tmp = (tmp << 4) + tmp
tmp = (tmp << 8) + tmp
tmp = (tmp << 16) + tmp
quo = tmp >> 32
```

Figure 7.

Negative Dividends. To handle signed division with truncation towards 0, all that is needed is to negate the incoming dividend if it is negative, do the division on the positive result, and negate the final quotient if the original dividend was negative. The testing of the dividend requires some overhead, but this can be partially offset since it will then be known that the initial adding of 1 to the dividend will not carry over into the higher precision word of the intermediate result. For example, to do signed division by 3 when the dividend turns out to be positive takes 17 instruction cycles which is exactly the same as the unsigned case. If the dividend turns out to be negative, then the signed division takes 19 instructions.

Performance. As a result of the application of this algorithm for division by small numbers on the Precision architecture, divisions using constant divisors less than twenty vary from one to 27 cycles, while divisions using variable divisors less then twenty vary from ten to 36 cycles. Both of these compare favorably with the average 80 cycles for the general-purpose divide routine.

8. Summary

Although integer multiplication and division are relatively infrequent operations, some programs may use them extensively — intentionally or otherwise. Design choices in their implementation must be made carefully to avoid slowing down or adding significant cost to the machine. We have described these design choices for the Hewlett-Packard Precision Architecture and have shown how the resultant instructions can be combined with careful frequency analysis to result in performance which meets or exceeds other methods but with significantly less cost. In particular, we have shown how, in the Precision architecture:

- Multiplications by compile-time constants can generally be performed in four or fewer (single-cycle) instructions;
- Multiplications by variables, including full overflow checking, can be obtained in an average of less than 20 (single-cycle) instructions; and
- Divisions by small numbers can be handled differently and take about 15 to 25 (single-cycle) instructions to perform, vs. about 80 for other divisions.

By examining the distribution of operands, over a large class of programs, we can conclude that, on the Precision architecture, the average multiply requires about six cycles and the average divide takes about 40.

9. Acknowledgements

The foundation for much of this work was laid by Steve Muchnick, Allen Baum, and Terrence Miller, who designed the basic instructions and explored their use in simple integer multiplications and divisions. Michael Mahon, Dave James, and Bill Bryg proposed several of the algorithmic

improvements described and Richard Campbell further refined the techniques. We are proud to present the collective work of all of these people. We especially thank Joel Birnbaum and Bill Worley, for without their vision, perseverance and leadership, the Precision architecture might never have existed.

10. References

[Bir85] Birnbaum, J.S., and Worley Jr., W.S., "Beyond RISC: High-Precision Architecture," *Hewlett-Packard Journal,* Vol. 36, No. 8, August 1985.

[Boo51] Booth, Andrew D., "A Signed Binary Multiplication Technique", *Quart. Journ. Mech. and Applied Math.,* Vol. IV Pt. 2 (1951), pp. 236-240.

[Cla82] Clark, D.W., "Measurement and Analysis of Instruction Use in the VAX 11/780," *Proc. of the 9th Symposium on Computer Architecture,* April 1982, pp. 9-17.

[Cou86] Coutant, D.S, Hammond, C.L., and Kelley, J.W., "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard Journal,* Vol. 37, No. 1, January 1986.

[Ele86] "A Simple Design May Pay Off Big for Hewlett-Packard", *Electronics,* March 3,1986, pp. 39-47

[Gib70] Gibson, J.C., "The Gibson Mix," Report TR 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. 1970.

[Hen82] Hennessy, J., et al,. "Hardware/Software Tradeoffs for Increased Performance," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Ca., March 1-3). ACM, New York, 1982, pp. 2-11.

[HP86] *Precision Architecture and Instruction Reference Manual,* Hewlett-Packard Co., HP Part Number 09740-90014, November 1986.

[Huc83] Huck, J.C., *Comparative Analysis of Computer Architectures.* Ph.D. Th., Stanford University, May 1983.

[Jou81] Jouppi, N., "MIPS II - Multiplication and Division Features," EE392C Final Reports, Stanford University, June 1981.

[Knu81] Knuth, D., The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, Addison-Wesley, 1981, pp. 444-446.

[Luk86] Lukes, J.A., "HP Precision Architecture Performance Analysis," *Hewlett- Packard Journal,* Vol. 37, No. 8, August 1986.

[Mah86] Mahon, Michael J., et al., "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal,* Vol. 37, no. 8, August 1986.

[Neu79] Neuhauser, Charles J., "Instruction Stream Monitoring of the PDP-11," Stanford University, Dept. of Electrical Engr., Computer Systems Laboratory, Tech. Note No. 156, May 1979

[Pat82] Patterson, D.A., and Sequin, C.H., "A VLSI RISC," *Computer,* Vol. 15, No. 9, Sept. 1982, pp. 8-21

[Pat84] Patterson, D.A., "RISC Watch" *Computer Architecture News,* Vol. 12, No. 1, Mar. 1984, pp. 11-19

[Pat85] Patterson, D.A., "Reduced Instruction Set Computers" *Communications of the ACM,* Vol. 12, No. 1, Jan. 1985, pp. 8-21

[Rad82] Radin, G., "The 801 Minicomputer," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Ca., March 1-3). ACM, New York, 1982, pp. 39-47.

[Shu78] Shustek, L.J., *Analysis and Performance of Computer Instruction Sets.* Ph.D. Th., Stanford University, May 1977.

[Swe82] Sweet, R.E., and Sandman, J.G., "Empirical Analysis of the Mesa Instruction Set," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Ca., March 1-3). ACM, New York, 1982, pp. 158-166.

[Was82] Waser, S. and Flynn, M.J., Introduction to Arithmetic for Digital Systems Designers, Holt, Rinehart and Winston, New York, 1982