

Efficient Hardware Arithmetic in Common Lisp

Hardware cycles, and one Ring to find them

Alexey Dejneka*, Christophe Rhodes†

March 25, 2004

Abstract

We describe a system whereby algorithms on arithmetic rings, such as those used in computing hash values, can be supported by using hardware instructions as typically found on general-purpose processors, even if the implementation of the algorithm is in a language and context generally without access to the underlying hardware. We present examples of how Lisp environments have previously dealt with efficiency issues in the mismatch between high-level arithmetic and the ring arithmetic provided by the hardware, and as an example of its use, give timings for an implementation of the MD5 algorithm written in natural Lisp.

1 Introduction

There are many axes along which implementations of computer languages may be measured: for instance, whether the implementation is interpreted or compiled (or where on the spectrum between these extremes it is based); whether all execution proceeds in a batch fashion, or whether it provides an interactive environment for incremental development and modification. One property which could, in contrast to these examples, be viewed as a property of the language — as opposed to a property of a given implementation — is its position on the ‘low-level’–‘high-level’ axis.

Common Lisp is rightly viewed as a high-level language, by which we mean that the underlying hardware or operating system is largely irrelevant to the behaviour of the standardized operators. Exceptions to this include the behaviour of pathname-related operations such as `open` and `parse-namestring`, which will obviously vary depending on the host filesystem; for our purposes here, a more interesting exception is the presence of the `fixnum` type.

Common Lisp defines [1] the semantics of mathematical operators (such as `+`) in terms of an idealized machine; arithmetic operations on rational arguments are required to return the exact rational result, while logical operators are defined to act on integers as though the internal representation were infinite-length twos-complement.

```
(+ 2147483648 2147483648) => 4294967296
(- 4294967296 1/2)      => 8589934591/2
(/ 4294967296 6)       => 2147483648/3
(logorc2 2147483648 15) => -16
(logqv 0 (1- (ash 1 32))) => -4294967296
```

*adejneka@comail.ru

†c.s.rhodes@damtp.cam.ac.uk

The `fixnum` type (as well as the constants such as `most-positive-fixnum`) is a window into the hardware representation of integers. It is expected that for sufficiently small integers (between `most-negative-fixnum` and `most-positive-fixnum`) there is a representation that admits implementation of mathematical operators with only a few hardware primitives. Thus, given the definitions

```
(defun add-a (x y)
  (+ x y))
(defun add-b (x y)
  (declare (type fixnum x y))
  (the fixnum (+ x y)))
```

it is expected that the second could be implemented in terms of the underlying hardware addition operation, whereas the first is a completely generic addition, and must therefore cater for all numerical types in the implementation (at least integers, rationals, floats and complexes).

Implementations of Common Lisp running on twos-complement 32-bit processors (such as the majority of general-purpose computers available today) typically have a `fixnum` range of either $-2^{23} \leq x < 2^{23}$ or $-2^{29} \leq x < 2^{29}$. The reasons for these choices follow from design decisions in the rest of the Lisp implementation; a Lisp object is typically represented as some tag bits (for type information) masked into an address (where the data structure lies). The range of 2^{30} comes from giving `fixnums` a distinguished representation compared with other immediate objects (which fit into a 32-bit word) such as characters; choosing to limit the range to 2^{24} frees two tag bits for other uses, such as efficiency tricks in a garbage collector. The exposure of the `fixnum` type therefore allows arithmetic to be almost as efficient as the underlying hardware¹, but over a more limited range than the underlying hardware allows.

In addition, compare the definition of `add-b` above with

```
(defun add-c (x y)
  (declare (type fixnum x y))
  (+ x y))
```

where we see an essential difference in the semantics of Common Lisp's `+` and a typical hardware addition operation; the addition here is required to give the arithmetically correct result, whereas a hardware addition is generally implemented modulo a certain word size.

Previous attempts to address this difference, found in commercially-available Common Lisp implementations, include slight deviations from conforming behaviour (as in Allegro Common Lisp, where the definition of `add-c` in an environment with the `safety` optimization quality set to zero translates to an assumption that operations with `fixnum` arguments have `fixnum` results), or extra-standard declarations (such as Lispworks' `fixnum-safety` declaration); in each case this provides a means for exploiting the underlying hardware, at a cost of additional complexity and non-portability, while still not allowing the entire range of the hardware arithmetic to be exploited.

This is of particular concern when considering commonly-used hashing algorithms such as MD5 [2] and SHA1 [3], or encryption algorithms such as Blowfish [4], where the algorithms have been designed to admit efficient implementation by using arithmetic on the $\mathbb{Z}_{2^{32}}$ ring. For an implementation of these algorithms in Common Lisp to be even minimally efficient on a 32-bit platform, the compiler must allow access to the full range of inputs to hardware arithmetic functions.

There are partial solutions for this problem in other high-level language environments: these include the availability in some languages (such as Java or Ada) of a modular integral type

¹but note that `lognot` and similar operators must arrange to treat tag bits specially.

| | | | |
|---------------------|----------------------|-----------------------|----------------------|
| <code>+</code> | <code>logxor</code> | <code>logandc1</code> | <code>logorc2</code> |
| <code>-</code> | <code>logeqv</code> | <code>logandc2</code> | <code>logior</code> |
| <code>*</code> | <code>lognand</code> | <code>logorc1</code> | <code>logand</code> |
| <code>lognot</code> | <code>lognor</code> | | |

Table 1: Standardized Common Lisp operators which can be optimized following the simple method described.

(see also [5] for related work); provision of functions which reflect the underlying hardware directly (such as in CMUCL [6] with its functions in the `KERNEL` package); or else the implicit requirement that any computation depending on direct representation in hardware for efficiency be implemented in a different language and called by a foreign function binding. None of these solutions is totally satisfactory, though Ada’s implementation of modular types is possibly the nearest; it is, however, not suited to a dynamically typed language, as each integer object would in general have to contain modularity information in addition to its value.

2 Implementation

While arithmetic is generic² in Common Lisp, it is easy to express the request for modular arithmetic on integers

```
(defun add-d (x y)
  (declare (type (unsigned-byte 32) x y))
  (logand (+ x y) #xffffffff))
```

where the request for ring arithmetic from `logand` can be expressed equivalently with either the `ldb` or `mod` operators. A naïve compiler, however, will still not exploit the available hardware operations, because the intermediate result (the value of `(+ x y)`) can be deduced to lie between 0 and 8589934590, which is clearly not suitable for a 32-bit representation in general).

It is, however, possible to compile `add-d` to the obvious hardware operation. Consider the addition of two bitstrings, $a_{n-1} \dots a_{32} a_{31} \dots a_0$ and $b_{n-1} \dots b_{32} b_{31} \dots b_0$, followed by the extraction of the low 32 bits. Clearly, this will give the same result as masking away the high bits, adding, then masking again, which is equivalent to addition modulo 2^{32} .

Note that

```
(defun add-e (x y)
  (declare (type (unsigned-byte 32) x y))
  (the (unsigned-byte 32) (+ x y)))
```

does not have this property in general; the use of `the` in `add-e` is a promise to the compiler that the addition does not overflow the `(unsigned-byte 32)` range, and the consequences are undefined if the addition does overflow.

We have implemented this conceptual transformation in SBCL [7] by performing a rewrite of the compiler’s internal representation of the program within the scope of a `logand`: if the result of the `logand` operation is known to be within the range $0 \leq x < 2^{32}$, the arguments to `logand` can be cut to a 32-bit width. This ‘cutting’, when faced with a combination such as `(+ x y)`, replaces the function being called with a specialized version `+mod32`, and recursively cuts the

²Here ‘generic’ is used in a generic sense, not the technical sense used by CLOS.

| | | |
|----------------------------------|--|----------------------------------|
| ; 990: ADD \$NL4,\$NL4,\$NLO | | ; ABO: ADD \$NL1,\$NL1,\$NLO |
| ; 994: ADD \$NL4,\$NL5,\$NL4 | | ; AB4: ADD \$NL1,\$NL5,\$NL1 |
| ; 998: RLWINM \$NL4,\$NL4,9,0,31 | | ; AB8: RLWINM \$NL1,\$NL1,4,0,31 |
| ; 99C: ADD \$NL4,\$NL1,\$NL4 | | ; ABC: ADD \$NL1,\$NL2,\$NL1 |
| ; 9A0: AND \$NLO,\$NL4,\$NL2 | | ; AC0: XOR \$NLO,\$NL1,\$NL2 |
| ; 9A4: ANDC \$NL5,\$NL1,\$NL2 | | ; AC4: XOR \$NLO,\$NLO,\$NL3 |
| ; 9A8: OR \$NL5,\$NLO,\$NL5 | | ; AC8: ADD \$NL5,\$NL4,\$NLO |
| ; 9AC: ADD \$NL5,\$NL3,\$NL5 | | |

Table 2: Segments from the inner loop of the MD5 algorithm (a G segment, left, and an H segment, right) compiled for 32-bit PowerPC by the SBCL compiler.

arguments to this function, until the whole of the parse tree has been cut or a call to an unknown or unoptimizeable function is encountered.

While we have described the replacement algorithm here for a 32-bit platform, the compiler implementation is not word-size specific, but queries known operations for suitable specialized replacements of a given size; indeed, SBCL on 64-bit platforms has analogous `-mod64` specialized operators. Standardized Common Lisp functions which can be optimized in this way are shown³ in table 1. Also, by providing simple compiler transformations for `ldb` and `mod` into `logand` where appropriate (always in the case of `ldb`; when the modular argument is a power of two for `mod`), we make the this optimization available irrespective of the idiom that a program author chooses. Similarly, simple transformations for the `boole` operator with constant operation argument enable those operations to be optimized automatically; it is also possible to provide a specialized `boole` for general, non-constant operation argument.

In addition, with a little care we can optimize the arithmetic shift operator, `ash`. Here we have to cater for an asymmetry in its arguments, as the second (shift count) argument certainly must not be masked. We need only cater for leftwards shifts; a right shift is not commutative with a mask in the manner described above, while if the integer to shift rightwards is known to be of the right type, then hardware arithmetic will be used automatically.

The core of the MD5 algorithm consists of four operations executed on $\mathbb{Z}_{2^{32}}$

$$\begin{aligned}
 F(x, y, z) &= (x \& y) \mid (\sim x \& z) \\
 G(x, y, z) &= (x \& z) \mid (y \& \sim z) \\
 H(x, y, z) &= x \wedge y \wedge z \\
 I(x, y, z) &= y \wedge (x \mid \sim z)
 \end{aligned}$$

along with additions and a bitfield rotation operation. These can be written as

```

(declaim (inline f g h i))
(defun f (x y z)
  (logior (logand x y) (logandc1 x z)))
(defun g (x y z)
  (logior (logand x z) (logandc2 y z)))
(defun h (x y z)
  (logxor x y z))

```

³Note that `logior` and `logand` have the property that if their arguments are of the required type, the result is of necessity of the right type; consequently no specialized function for them is necessary.

| Routine | Mean time (seconds) |
|-----------------------------------|---------------------|
| Lisp (without modular arithmetic) | 330.2 ± 0.3 |
| Lisp (with modular arithmetic) | 1.58 ± 0.01 |
| C (-00) | 2.53 ± 0.02 |
| C (-02) | 1.16 ± 0.01 |

Table 3: Time taken to compute the MD5 hash of a 25.6Mb file on a PowerPC 740 (532 bogomips). Lisp timings were taken with SBCL version 0.8.8; C with gcc 2.95. The computation time on a file of this size is dominated by the inner loop of the MD5 algorithm.

```
(defun i (x y z)
  (ldb (byte 32 0) (logxor y (logorc2 x z))))
```

and, when used in a suitable context⁴, will be compiled to machine-level instructions as in table 2. Similar definitions, naturally expressing kernel operations, can be made for the SHA1 cryptographic hash, and for similar algorithms.

3 Extensions and further work

Comparing the efficiency of our MD5 routine with a implementation (as provided by the Debian project [8] in the `textutils` package) in C reveals a significant timing difference (see table 3). While not so pronounced as that caused by generic integer arithmetic, which typically manifests in a slowdown by many orders of magnitude, the Common Lisp routine is about 35% slower to compute the MD5 hash of a 20Mb file than the C program. This can largely be explained by the current lack of an instruction scheduler in SBCL’s compiler for the PowerPC, leading to pipeline stalls when loading the MD5 constants into general-purpose registers. It is expected that implementing even a rudimentary instruction scheduler would improve performance to be competitive with C, but this is outside the scope of this paper.

There are obvious extensions to the techniques discussed here. One such is to provide an analogous environment for signed arithmetic: however, unlike arithmetic on $\mathbb{Z}_{2^{32}}$, there is no simple idiom⁵ for coercion to twos-complement signed integers. The major use for such an optimization would probably be for simulations (or emulations) of hardware (as in [9], for example); it may be of more use to provide a `sldb` operator, analogous to `ldb` but with twos-complement semantics of the high bit, and use this operator to detect and optimize for signed machine word arithmetic.

As mentioned above, bitwise rotation often plays a part in hashing algorithms. There is no analogous operator in Common Lisp; at present, SBCL provides a `rotate-byte` operator with functionality on general byte size, but with specialized implementations for word size natively supported by hardware. A simple extension to our code transformation, however, could detect uses of `(logior (ash x y) (ash x z))`, with $(y - z) \bmod 32 = 0$, in an appropriate context, and convert that to a hardware rotation operation if such is supported.

Our scheme does not currently allow for multiple natural hardware widths on a platform such as the x86 family, where the hardware supports 8- and 16-bit arithmetic as well as 32-bit. This lack has little impact in practice, because of the lack of critical sections in algorithms which could exploit this facility, but it would be good to allow it should the situation arise.

⁴such as in the `sb-md5` module distributed with SBCL.

⁵`(let ((x <expr>)) (dpb x (byte 32 0) (- (ldb (byte 1 31) x))))` is perhaps the clearest.

It is also limited by the scope of flow analysis in the compiler. While the compiler underlying SBCL is sophisticated in some ways, despite its age, it is not currently capable of extension to allow the optimization using this scheme of a form such as `(ldb (byte 32 0) (loop for i of-type (unsigned-byte 32) from x to y summing i))`. Improvements to the other parts of the SBCL compiler will naturally increase the ability of the modular arithmetic optimization to perform.

4 Conclusions

We have described an optimization permitting systems with generic arithmetic nevertheless to achieve close to optimal performance in circumstances where this is possible without sacrificing the mathematical correctness of the operations. The scheme has been tested in several implementations of real-world algorithms, such as MD-5, SHA1 and Blowfish, in each case yielding performance in the same order of magnitude as optimized C implementations without sacrificing the portability of the Lisp source code. We have detailed the current limitations of our implementation of this optimization, and have suggested avenues for further improvement.

Acknowledgements

We acknowledge the work of Rob MacLachlan and of all those involved in the CMUCL project; thanks also to William Harold Newman and Daniel Barlow for their work on SBCL. Pierre Mai, Nathan Froyd and Alain Picard provided implementations of algorithms that spurred the development of the optimization technique described herein.

References

- [1] Kent Pitman and Kathy Chapman, editors. *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI, 1994.
- [2] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., 1992.
- [3] D. Eastlake, III and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, Motorola and Cisco Systems, 2001.
- [4] B. Schneier. Fast software encryption. In *Cambridge Security Workshop Proceedings (December 1993)*. Springer-Verlag, 1994.
- [5] Kevin Redwine and Norman Ramsey. Widening Integer Arithmetic. In *Proceedings of the 13th International Conference on Compiler Construction*, 2004.
- [6] Robert A. MacLachlan. CMU Common Lisp User’s Manual. Technical report, School of Computer Science, Carnegie-Mellon University, 1992.
- [7] Christophe S. Rhodes et al. Steel Bank Common Lisp User Manual. in preparation.
- [8] The Debian Project. <http://www.debian.org/>.
- [9] Bishop Brock, Matt Kaufmann, and J. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design*. Springer-Verlag, 1996.