HZ BOOKS

PEARSON
Addison Wesley

经典计算机科学著作最新版

计算机程序设计艺术

第1卷 第1册 （双语版）

MMIX：新千年的RISC计算机

The Art of Computer Programming, Volume 1
MMIX: A RISC Computer for the New Millennium

Fascicle 1

苏运霖 译

（美）Donald E. Knuth 著

机械工业出版社
China Machine Press

计算机程序设计艺术

第1卷 第1册

MMIX：新千年的RISC计算机

The Art of Computer Programming
Volume 1, Fascicle 1
MMIX: A RISC Computer for the New Millennium

（双语版）

（美） Donald E. Knuth 著
斯坦福大学
苏运霖 译

机械工业出版社
China Machine Press

关于算法分析的这多卷论著已经长期被公认为经典计算机科学的定义性描述。本册更新了《计算机程序设计艺术，第1卷，第3版：基本算法》的部分内容，并且最终将成为该书第4版的一部分。具体地说，它向程序员提供了盼望已久的MMIX——代替原来的MIX的一个以RISC为基础的计算机，并且描述了MMIX汇编语言。另外，本册也介绍了有关子程序、共行程序以及解释性程序的新内容。

本书版权登记号：图字：01-2005-3615

图书在版编目（CIP）数据

计算机程序设计艺术 第1卷 第1册，MMIX：新千年的RISC计算机（双语版）/（美）克努特（Knuth, D. E.）著. 苏运霖译. －北京：机械工业出版社，2006.3

书名原文：The Art of Computer Programming, Volume 1, Fascicle 1: MMIX—A RISC Computer for the New Millennium

ISBN 7-111-18031-3

Ⅰ.计… Ⅱ.①克… ②苏… Ⅲ.程序设计 Ⅳ.TP311.4

中国版本图书馆CIP数据核字（2006）第010963号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：朱起飞
北京瑞德印刷有限公司印刷·新华书店北京发行所发行
2006年4月第1版第1次印刷
787mm × 1020mm 1/16 · 17.25印张
定价：45.00元

PREFACE

fas·ci·cle \\'fasəkəl\\ n ... 1: a small bundle ... an inflorescence consisting of a compacted cyme less capitate than a glomerule ... 2: one of the divisions of a book published in parts

— P. B. GOVE, *Webster's Third New International Dictionary* (1961)

THIS IS THE FIRST of a series of updates that I plan to make available at regular intervals as I continue working toward the ultimate editions of *The Art of Computer Programming*.

I was inspired to prepare fascicles like this by the example of Charles Dickens, who issued his novels in serial form; he published a dozen installments of *Oliver Twist* before having any idea what would become of Bill Sikes! I was thinking also of James Murray, who began to publish 350-page portions of the Oxford English Dictionary in 1884, finishing the letter B in 1888 and the letter C in 1895. (Murray died in 1915 while working on the letter T; my task is, fortunately, much simpler than his.)

Unlike Dickens and Murray, I have computers to help me edit the material, so that I can easily make changes before putting everything together in its final form. Although I'm trying my best to write comprehensive accounts that need no further revision, I know that every page brings me hundreds of opportunities to make mistakes and to miss important ideas. My files are bursting with notes about beautiful algorithms that have been discovered, but computer science has grown to the point where I cannot hope to be an authority on all the material I wish to cover. Therefore I need extensive feedback from readers before I can finalize the official volumes.

In other words, I think these fascicles will contain a lot of Good Stuff, and I'm excited about the opportunity to present everything I write to whoever wants to read it, but I also expect that beta-testers like you can help me make it Way Better. As usual, I will gratefully pay a reward of \$2.56 to the first person who reports anything that is technically, historically, typographically, or politically incorrect.

Charles Dickens usually published his work once a month, sometimes once a week; James Murray tended to finish a 350-page installment about once every 18 months. My goal, God willing, is to produce two 128-page fascicles per year. Most of the fascicles will represent new material destined for Volumes 4 and higher; but sometimes I will be presenting amendments to one or more of the earlier volumes. For example, Volume 4 will need to refer to topics that belong in Volume 3, but weren't invented when Volume 3 first came out. With luck, the entire work will make sense eventually.

Fascicle Number One is about MMIX, the long-promised replacement for MIX. Thirty-seven years have passed since the MIX computer was designed, and computer architecture has been converging during those years towards a rather different style of machine. Therefore I decided in 1990 to replace MIX with a new computer that would contain even less saturated fat than its predecessor.

Exercise 1.3.1–25 in the first three editions of Volume 1 spoke of an extended MIX called MixMaster, which was upward compatible with the old version. But MixMaster itself has long been hopelessly obsolete. It allowed for several gigabytes of memory, but one couldn't even use it with ASCII code to print lowercase letters. And ouch, its standard conventions for calling subroutines were irrevocably based on self-modifying instructions! Decimal arithmetic and self-modifying code were popular in 1962, but they sure have disappeared quickly as machines have gotten bigger and faster. Fortunately the modern RISC architecture has a very appealing structure, so I've had a chance to design a new computer that is not only up to date but also fun.

Many readers are no doubt thinking, "Why does Knuth replace MIX by another machine instead of just sticking to a high-level programming language? Hardly anybody uses assemblers these days." Such people are entitled to their opinions, and they need not bother reading the machine-language parts of my books. But the reasons for machine language that I gave in the preface to Volume 1, written in the early 1960s, remain valid today:

- One of the principal goals of my books is to show how high-level constructions are actually implemented in machines, not simply to show how they are applied. I explain coroutine linkage, tree structures, random number generation, high-precision arithmetic, radix conversion, packing of data, combinatorial searching, recursion, etc., from the ground up.
- The programs needed in my books are generally so short that their main points can be grasped easily.
- People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird.
- Machine language is necessary in any case, as output of some of the software that I describe.
- Expressing basic methods like algorithms for sorting and searching in machine language makes it possible to carry out meaningful studies of the effects of cache and RAM size and other hardware characteristics (memory speed, pipelining, multiple issue, lookaside buffers, the size of cache blocks, etc.) when comparing different schemes.

Moreover, if I did use a high-level language, what language should it be? In the 1960s I would probably have chosen Algol W; in the 1970s, I would then have had to rewrite my books using Pascal; in the 1980s, I would surely have changed everything to C; in the 1990s, I would have had to switch to C++ and then probably to Java. In the 2000s, yet another language will no doubt be *de* *rigueur.* I cannot afford the time to rewrite my books as languages go in and out of fashion; languages aren't the point of my books, the point is rather what you can do in your favorite language. My books focus on timeless truths.

Therefore I will continue to use English as the high-level language in *The Art of Computer Programming*, and I shall continue to use a low-level language to indicate how machines actually compute. Readers who only want to see algorithms that are already packaged in a plug-in way, using a trendy language, should buy other people's books.

The good news is that programming for MMIX is pleasant and simple. This fascicle presents

1) a programmer's introduction to the machine (replacing Section 1.3.1 of the third edition of Volume 1);
2) the MMIX assembly language (replacing Section 1.3.2);
3) new material on subroutines, coroutines, and interpretive routines (replacing Sections 1.4.1, 1.4.2, and 1.4.3).

Of course, MIX appears in many places throughout the existing editions of Volumes 1–3, and dozens of programs need to be rewritten for MMIX before the next editions of those volumes are ready. Readers who would like to help with this conversion process are encouraged to join the MMIXmasters, a happy group of volunteers based at `mmixmasters.sourceforge.net`.

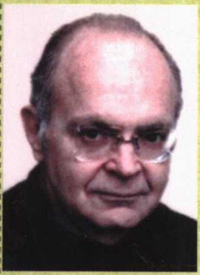
The fourth edition of Volume 1 will not be ready until after Volumes 4 and 5 have been completed; therefore two quite different versions of Sections 1.3.1, 1.3.2, 1.4.1, 1.4.2, and 1.4.3 will coexist for several years. In order to avoid potential confusion, I've temporarily assigned "prime numbers" 1.3.1′, 1.3.2′, 1.4.1′, 1.4.2′, and 1.4.3′ to the new material.

I am extremely grateful to all the people who helped me with the design of MMIX. In particular, John Hennessy and Richard L. Sites deserve special thanks for their active participation and substantial contributions. Thanks also to Vladimir Ivanović for volunteering to be the MMIX grandmaster/webmaster.

Stanford, California
May 1999

D. E. K.

You can, if you want, rewrite forever.
— NEIL SIMON, *Rewrites: A Memoir* (1996)



Donald E. Knuth

（唐纳德 · E. 克努特，中文名高德纳）

由于在算法和程序设计技术方面的先驱性工作，由于发明了计算机排版系统TEX和METAFONT，以及由于他的富于创造力的，影响深远的论著，Knuth名扬全球。作为斯坦福大学计算机程序设计艺术的荣誉退休教授，Knuth现在正投入全部的精力来完成这些分册以及包含这些分册的七卷著作。

CONTENTS
目　录

Chapter 1

Basic Concepts

1.3´. MMIX

In many places throughout this book we will have occasion to refer to a computer's internal machine language. The machine we use is a mythical computer called "MMIX." MMIX — pronounced *EM-micks* — is very much like nearly every general-purpose computer designed since 1985, except that it is, perhaps, nicer. The language of MMIX is powerful enough to allow brief programs to be written for most algorithms, yet simple enough so that its operations are easily learned.

The reader is urged to study this section carefully, since MMIX language appears in so many parts of this book. There should be no hesitation about learning a machine language; indeed, the author once found it not uncommon to be writing programs in a half dozen different machine languages during the same week! Everyone with more than a casual interest in computers will probably get to know at least one machine language sooner or later. Machine language helps programmers understand what really goes on inside their computers. And once one machine language has been learned, the characteristics of another are easy to assimilate. Computer science is largely concerned with an understanding of how low-level details make it possible to achieve high-level goals.

Software for running MMIX programs on almost any real computer can be downloaded from the website for this book (see page ii). The complete source code for the author's MMIX routines appears in the book *MMIXware* [*Lecture Notes in Computer Science* **1750** (1999)]; that book will be called "the *MMIXware* document" in the following pages.

1.3.1´. Description of MMIX

MMIX is a polyunsaturated, 100% natural computer. Like most machines, it has an identifying number — the 2009. This number was found by taking 14 actual computers very similar to MMIX and on which MMIX could easily be simulated, then averaging their numbers with equal weight:

$$\begin{aligned}&(\mathrm{Cray\,I} + \mathrm{IBM\,801} + \mathrm{RISC\,II} + \mathrm{Clipper\,C300} + \mathrm{AMD\,29K} + \mathrm{Motorola\,88K}\\ &\qquad + \mathrm{IBM\,601} + \mathrm{Intel\,i960} + \mathrm{Alpha\,21164} + \mathrm{POWER\,2} + \mathrm{MIPS\,R4000}\\ &\qquad + \mathrm{Hitachi\,SuperH4} + \mathrm{StrongARM\,110} + \mathrm{Sparc\,64})/14\\ &\quad \doteq 28126/14 = 2009. \end{aligned} \tag{1}$$

The same number may also be obtained in a simpler way by taking Roman numerals.

Bits and bytes. MMIX works with patterns of 0s and 1s, commonly called binary digits or *bits*, and it usually deals with 64 bits at a time. For example, the 64-bit quantity

$$1001111000110111011110011011100101111111010010100111110000010110 \tag{2}$$

is a typical pattern that the machine might encounter. Long patterns like this can be expressed more conveniently if we group the bits four at a time and use *hexadecimal digits* to represent each group. The sixteen hexadecimal digits are

$$\begin{array}{llll} \texttt{0} = 0000, & \texttt{4} = 0100, & \texttt{8} = 1000, & \texttt{c} = 1100, \\ \texttt{1} = 0001, & \texttt{5} = 0101, & \texttt{9} = 1001, & \texttt{d} = 1101, \\ \texttt{2} = 0010, & \texttt{6} = 0110, & \texttt{a} = 1010, & \texttt{e} = 1110, \\ \texttt{3} = 0011, & \texttt{7} = 0111, & \texttt{b} = 1011, & \texttt{f} = 1111. \end{array} \tag{3}$$

We shall always use a distinctive typeface for hexadecimal digits, as shown here, so that they won't be confused with the decimal digits 0–9; and we will usually also put the symbol # just before a hexadecimal number, to make the distinction even clearer. For example, (2) becomes

$$\texttt{\#9e3779b97f4a7c16} \tag{4}$$

in hexadecimalese. Uppercase digits ABCDEF are often used instead of abcdef, because #9E3779B97F4A7C16 looks better than #9e3779b97f4a7c16 in some contexts; there is no difference in meaning.

A sequence of eight bits, or two hexadecimal digits, is commonly called a *byte*. Most computers now consider bytes to be their basic, individually addressable units of data; we will see that an MMIX program can refer to as many as 2^{64} bytes, each of which has its own address from #0000000000000000 to #ffffffffffffffff. Letters, digits, and punctuation marks of languages like English are often represented with one byte per character, using the American Standard Code for Information Interchange (ASCII). For example, the ASCII equivalent of MMIX is #4d4d4958. ASCII is actually a 7-bit code with control characters #00–#1f, printing characters #20–#7e, and a "delete" character #7f [see *CACM* **8** (1965), 207–214; **11** (1968), 849–852; **12** (1969), 166–178]. It was extended during the 1980s to an international standard 8-bit code known as Latin-1 or ISO 8859-1, thereby encoding accented letters: *pâté* is #70e274e9.

"Of the 256th squadron?"
"Of the fighting 256th Squadron," Yossarian replied.
. . . "That's two to the fighting eighth power."
— JOSEPH HELLER, *Catch-22* (1961)

A 16-bit code that supports nearly *every* modern language became an international standard during the 1990s. This code, known as Unicode UTF-16 or ISO/IEC 10646 UCS-2, includes not only Greek letters like Σ and σ (#03a3 and #03c3), Cyrillic letters like Щ and щ (#0429 and #0449), Armenian letters like Շ and շ (#0547 and #0577), Hebrew letters like ש (#05e9), Arabic letters like ش (#0634), and Indian letters like श (#0936) or শ (#09b6) or ଶ (#0b36) or ஷ (#0bb7), etc., but also tens of thousands of East Asian ideographs such as the Chinese character for mathematics and computing, 算 (#7b97). It even has special codes for Roman numerals: MMIX = #216f 216f 2160 2169. Ordinary ASCII or Latin-1 characters are represented by simply giving them a leading byte of zero: *pâté* is #0070 00e2 0074 00e9, *à l'Unicode*.

We will use the convenient term *wyde* to describe a 16-bit quantity like the wide characters of Unicode, because two-byte quantities are quite important in practice. We also need convenient names for four-byte and eight-byte quantities, which we shall call *tetrabytes* (or "tetras") and *octabytes* (or "octas"). Thus

$$\begin{aligned}2\text{ bytes} &= 1\text{ wyde};\\ 2\text{ wydes} &= 1\text{ tetra};\\ 2\text{ tetras} &= 1\text{ octa}.\end{aligned}$$

One octabyte equals four wydes equals eight bytes equals sixty-four bits.

Bytes and multibyte quantities can, of course, represent numbers as well as alphabetic characters. Using the binary number system,

an unsigned byte can express the numbers 0 .. 255;
an unsigned wyde can express the numbers 0 .. 65,535;
an unsigned tetra can express the numbers 0 .. 4,294,967,295;
an unsigned octa can express the numbers 0 .. 18,446,744,073,709,551,615.

Integers are also commonly represented by using *two's complement notation*, in which the leftmost bit indicates the sign: If the leading bit is 1, we subtract 2^n to get the integer corresponding to an n-bit number in this notation. For example, -1 is the signed byte `#ff`; it is also the signed wyde `#ffff`, the signed tetrabyte `#ffffffff`, and the signed octabyte `#ffffffffffffffff`. In this way

a signed byte can express the numbers -128 .. 127;
a signed wyde can express the numbers $-32{,}768$ .. 32,767;
a signed tetra can express the numbers $-2{,}147{,}483{,}648$ .. 2,147,483,647;
a signed octa can express the numbers $-9{,}223{,}372{,}036{,}854{,}775{,}808$ ..
    9,223,372,036,854,775,807.

Memory and registers. From a programmer's standpoint, an MMIX computer has 2^{64} cells of *memory* and 2^8 general-purpose *registers*, together with 2^5 special registers (see Fig. 13). Data is transferred from the memory to the registers, transformed in the registers, and transferred from the registers to the memory. The cells of memory are called M[0], M[1], ..., M$[2^{64}-1]$; thus if x is any octabyte, M$[x]$ is a byte of memory. The general-purpose registers are called \$0, \$1, ..., \$255; thus if x is any byte, $\$x$ is an octabyte.

The 2^{64} bytes of memory are grouped into 2^{63} wydes, $M_2[0] = M_2[1] = M[0]M[1]$, $M_2[2] = M_2[3] = M[2]M[3]$, ...; each wyde consists of two consecutive bytes $M[2k]M[2k+1] = M[2k]\times 2^8 + M[2k+1]$, and is denoted either by $M_2[2k]$ or by $M_2[2k+1]$. Similarly there are 2^{62} tetrabytes

$$M_4[4k] = M_4[4k+1] = \cdots = M_4[4k+3] = M[4k]M[4k+1]\ldots M[4k+3],$$

and 2^{61} octabytes

$$M_8[8k] = M_8[8k+1] = \cdots = M_8[8k+7] = M[8k]M[8k+1]\ldots M[8k+7].$$

In general if x is any octabyte, the notations $M_2[x]$, $M_4[x]$, and $M_8[x]$ denote the wyde, the tetra, and the octa that contain byte $M[x]$; we ignore the least

MMIX

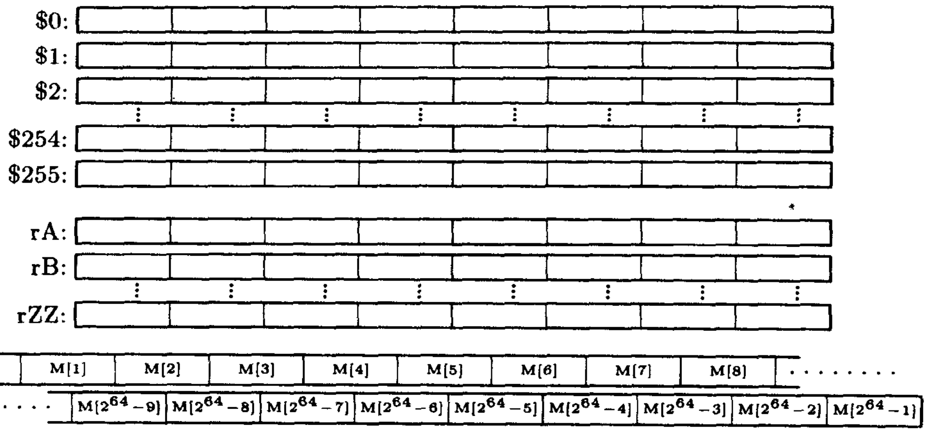


Fig. 13. The MMIX computer, as seen by a programmer, has 256 general-purpose registers and 32 special-purpose registers, together with 2^{64} bytes of virtual memory. Each register holds 64 bits of data.

significant $\lg t$ bits of x when referring to $M_t[x]$. For completeness, we also write $M_1[x] = M[x]$, and we define $M[x] = M[x \bmod 2^{64}]$ when $x < 0$ or $x \geq 2^{64}$.

The 32 special registers of MMIX are called rA, rB, ..., rZ, rBB, rTT, rWW, rXX, rYY, and rZZ. Like their general-purpose cousins, they each hold an octabyte. Their uses will be explained later; for example, we will see that rA controls arithmetic interrupts while rR holds the remainder after division.

Instructions. MMIX's memory contains instructions as well as data. An *instruction* or "command" is a tetrabyte whose four bytes are conventionally called OP, X, Y, and Z. OP is the *operation code* (or "opcode," for short); X, Y, and Z specify the *operands*. For example, #20010203 is an instruction with OP = #20, X = #01, Y = #02, and Z = #03, and it means "Set \$1 to the sum of \$2 and \$3." The operand bytes are always regarded as unsigned integers.

Each of the 256 possible opcodes has a symbolic form that is easy to remember. For example, opcode #20 is `ADD`. We will deal almost exclusively with symbolic opcodes; the numeric equivalents can be found, if needed, in Table 1 below, and also in the endpapers of this book.

The X, Y, and Z bytes also have symbolic representations, consistent with the assembly language that we will discuss in Section 1.3.2′. For example, the instruction #20010203 is conventionally written '`ADD $1,$2,$3`', and the addition instruction in general is written '`ADD $X,$Y,$Z`'. Most instructions have three operands, but some of them have only two, and a few have only one. When there are two operands, the first is X and the second is the two-byte quantity YZ; the symbolic notation then has only one comma. For example, the instruction

'`INCL $X,YZ`' increases register \$X by the amount YZ. When there is only one operand, it is the unsigned three-byte number XYZ, and the symbolic notation has no comma at all. For example, we will see that '`JMP @+4*XYZ`' tells `MMIX` to find its next instruction by skipping ahead XYZ tetrabytes; the instruction '`JMP @+1000000`' has the hexadecimal form `#f003d090`, because `JMP` = `#f0` and 250000 = `#03d090`.

We will describe each `MMIX` instruction both informally and formally. For example, the informal meaning of '`ADD $X,$Y,$Z`' is "Set \$X to the sum of \$Y and \$Z"; the formal definition is '$\mathrm{s}(\$X) \leftarrow \mathrm{s}(\$Y) + \mathrm{s}(\$Z)$'. Here $\mathrm{s}(x)$ denotes the *signed integer* corresponding to the bit pattern x, according to the conventions of two's complement notation. An assignment like $\mathrm{s}(x) \leftarrow N$ means that x is to be set to the bit pattern for which $\mathrm{s}(x) = N$. (Such an assignment causes *integer overflow* if N is too large or too small to fit in x. For example, an `ADD` will overflow if $\mathrm{s}(\$Y) + \mathrm{s}(\$Z)$ is less than -2^{63} or greater than $2^{63} - 1$. When we're discussing an instruction informally, we will often gloss over the possibility of overflow; the formal definition, however, will make everything precise. In general the assignment $\mathrm{s}(x) \leftarrow N$ sets x to the binary representation of $N \bmod 2^n$, where n is the number of bits in x, and it signals overflow if $N < -2^{n-1}$ or $N \geq 2^{n-1}$; see exercise 5.)

Loading and storing. Although `MMIX` has 256 different opcodes, we will see that they fall into a few easily learned categories. Let's start with the instructions that transfer information between the registers and the memory.

Each of the following instructions has a *memory address* A obtained by adding \$Y to \$Z. Formally,

$$\mathrm{A} = \bigl(\mathrm{u}(\$Y) + \mathrm{u}(\$Z)\bigr) \bmod 2^{64} \qquad (5)$$

is the sum of the *unsigned integers* represented by \$Y and \$Z, reduced to a 64-bit number by ignoring any carry that occurs at the left when those two integers are added. In this formula the notation $\mathrm{u}(x)$ is analogous to $\mathrm{s}(x)$, but it considers x to be an unsigned binary number.

- `LDB $X,$Y,$Z` (load byte): $\mathrm{s}(\$X) \leftarrow \mathrm{s}\bigl(\mathrm{M}_1[\mathrm{A}]\bigr)$.
- `LDW $X,$Y,$Z` (load wyde): $\mathrm{s}(\$X) \leftarrow \mathrm{s}\bigl(\mathrm{M}_2[\mathrm{A}]\bigr)$.
- `LDT $X,$Y,$Z` (load tetra): $\mathrm{s}(\$X) \leftarrow \mathrm{s}\bigl(\mathrm{M}_4[\mathrm{A}]\bigr)$.
- `LDO $X,$Y,$Z` (load octa): $\mathrm{s}(\$X) \leftarrow \mathrm{s}\bigl(\mathrm{M}_8[\mathrm{A}]\bigr)$.

These instructions bring data from memory into register \$X, changing the data if necessary from a signed byte, wyde, or tetrabyte to a signed octabyte of the same value. For example, suppose the octabyte $\mathrm{M}_8[1002] = \mathrm{M}_8[1000]$ is

$$\mathrm{M}[1000]\mathrm{M}[1001]\ldots\mathrm{M}[1007] = \texttt{\#0123456789abcdef}. \qquad (6)$$

Then if \$2 = 1000 and \$3 = 2, we have A = 1002, and

```
LDB $1,$2,$3 sets $1 ← #0000000000000045;
LDW $1,$2,$3 sets $1 ← #0000000000004567;
LDT $1,$2,$3 sets $1 ← #0000000001234567;
LDO $1,$2,$3 sets $1 ← #0123456789abcdef.
```

But if $3 = 5, so that A = 1005,

```
LDB $1,$2,$3 sets $1 ← #ffffffffffffffab;
LDW $1,$2,$3 sets $1 ← #ffffffffffff89ab;
LDT $1,$2,$3 sets $1 ← #ffffffff89abcdef;
LDO $1,$2,$3 sets $1 ← #0123456789abcdef.
```

When a signed byte or wyde or tetra is converted to a signed octa, its sign bit is "extended" into all positions to the left.

- `LDBU $X,$Y,$Z` (load byte unsigned): u($X) ← u($M_1$[A]).
- `LDWU $X,$Y,$Z` (load wyde unsigned): u($X) ← u($M_2$[A]).
- `LDTU $X,$Y,$Z` (load tetra unsigned): u($X) ← u($M_4$[A]).
- `LDOU $X,$Y,$Z` (load octa unsigned): u($X) ← u($M_8$[A]).

These instructions are analogous to `LDB`, `LDW`, `LDT`, and `LDO`, but they treat the memory data as *unsigned*; bit positions at the left of the register are set to zero when a short quantity is being lengthened. Thus, in the example above, `LDBU $1,$2,$3` with $2 + $3 = 1005 would set $1 ← `#00000000000000ab`.

The instructions `LDO` and `LDOU` actually have exactly the same behavior, because no sign extension or padding with zeros is necessary when an octabyte is loaded into a register. But a good programmer will use `LDO` when the sign is relevant and `LDOU` when it is not; then readers of the program can better understand the significance of what is being loaded.

- `LDHT $X,$Y,$Z` (load high tetra): u($X) ← u($M_4$[A]) × 2^{32}.

Here the tetrabyte M_4[A] is loaded into the *left* half of $X, and the right half is set to zero. For example, `LDHT $1,$2,$3` sets $1 ← `#89abcdef00000000`, assuming (6) with $2 + $3 = 1005.

- `LDA $X,$Y,$Z` (load address): u($X) ← A.

This instruction, which puts a memory address into a register, is essentially the same as the `ADDU` instruction described below. Sometimes the words "load address" describe its purpose better than the words "add unsigned."

- `STB $X,$Y,$Z` (store byte): s(M_1[A]) ← s($X).
- `STW $X,$Y,$Z` (store wyde): s(M_2[A]) ← s($X).
- `STT $X,$Y,$Z` (store tetra): s(M_4[A]) ← s($X).
- `STO $X,$Y,$Z` (store octa): s(M_8[A]) ← s($X).

These instructions go the other way, placing register data into the memory. Overflow is possible if the (signed) number in the register lies outside the range of the memory field. For example, suppose register $1 contains the number −65536 = `#ffffffffffff0000`. Then if $2 = 1000, $3 = 2, and (6) holds,

```
STB $1,$2,$3 sets M8[1000] ← #0123006789abcdef (with overflow);
STW $1,$2,$3 sets M8[1000] ← #0123000089abcdef (with overflow);
STT $1,$2,$3 sets M8[1000] ← #ffff000089abcdef;
STO $1,$2,$3 sets M8[1000] ← #ffffffffffff0000.
```

- **STBU $X,$Y,$Z** (store byte unsigned):
  $u(M_1[A]) \leftarrow u(\$X) \bmod 2^8$.
- **STWU $X,$Y,$Z** (store wyde unsigned):
  $u(M_2[A]) \leftarrow u(\$X) \bmod 2^{16}$.
- **STTU $X,$Y,$Z** (store tetra unsigned):
  $u(M_4[A]) \leftarrow u(\$X) \bmod 2^{32}$.
- **STOU $X,$Y,$Z** (store octa unsigned): $u(M_8[A]) \leftarrow u(\$X)$.

These instructions have exactly the same effect on memory as their signed counterparts STB, STW, STT, and STO, but overflow never occurs.

- **STHT $X,$Y,$Z** (store high tetra): $u(M_4[A]) \leftarrow \lfloor u(\$X)/2^{32} \rfloor$.

The left half of register $X is stored in memory tetrabyte $M_4[A]$.

- **STCO X,$Y,$Z** (store constant octabyte): $u(M_8[A]) \leftarrow X$.

A constant between 0 and 255 is stored in memory octabyte $M_8[A]$.

Arithmetic operators. Most of MMIX's operations take place strictly between registers. We might as well begin our study of the register-to-register operations by considering addition, subtraction, multiplication, and division, because computers are supposed to be able to compute.

- **ADD $X,$Y,$Z** (add): $s(\$X) \leftarrow s(\$Y) + s(\$Z)$.
- **SUB $X,$Y,$Z** (subtract): $s(\$X) \leftarrow s(\$Y) - s(\$Z)$.
- **MUL $X,$Y,$Z** (multiply): $s(\$X) \leftarrow s(\$Y) \times s(\$Z)$.
- **DIV $X,$Y,$Z** (divide): $s(\$X) \leftarrow \lfloor s(\$Y)/s(\$Z) \rfloor$ $[\$Z \neq 0]$, and
  $s(rR) \leftarrow s(\$Y) \bmod s(\$Z)$.

Sums, differences, and products need no further discussion. The DIV command forms the quotient and remainder as defined in Section 1.2.4; the remainder goes into the special *remainder register* rR, where it can be examined by using the instruction GET $X,rR described below. If the divisor $Z is zero, DIV sets $\$X \leftarrow 0$ and $rR \leftarrow \$Y$ (see Eq. 1.2.4–(1)); an "integer divide check" also occurs.

- **ADDU $X,$Y,$Z** (add unsigned): $u(\$X) \leftarrow \bigl(u(\$Y) + u(\$Z)\bigr) \bmod 2^{64}$.
- **SUBU $X,$Y,$Z** (subtract unsigned): $u(\$X) \leftarrow \bigl(u(\$Y) - u(\$Z)\bigr) \bmod 2^{64}$.
- **MULU $X,$Y,$Z** (multiply unsigned): $u(rH\,\$X) \leftarrow u(\$Y) \times u(\$Z)$.
- **DIVU $X,$Y,$Z** (divide unsigned): $u(\$X) \leftarrow \lfloor u(rD\,\$Y)/u(\$Z) \rfloor$, $u(rR) \leftarrow$ $u(rD\,\$Y) \bmod u(\$Z)$, if $u(\$Z) > u(rD)$; otherwise $\$X \leftarrow rD$, $rR \leftarrow \$Y$.

Arithmetic on unsigned numbers never causes overflow. A full 16-byte product is formed by the MULU command, and the upper half goes into the special *himult register* rH. For example, when the unsigned number #9e3779b97f4a7c16 in (2) and (4) above is multiplied by itself we get

$$rH \leftarrow \texttt{\#61c8864680b583ea}, \quad \$X \leftarrow \texttt{\#1bb32095ccdd51e4}. \tag{7}$$

In this case the value of rH has turned out to be exactly 2^{64} minus the original number #9e3779b97f4a7c16; this is not a coincidence! The reason is that (2) actually gives the first 64 bits of the binary representation of the golden ratio $\phi^{-1} = \phi - 1$, if we place a binary radix point at the *left*. (See Table 2 in Appendix A.) Squaring gives us an approximation to the binary representation of $\phi^{-2} = 1 - \phi^{-1}$, with the radix point now at the left of rH.