NEWLY AVAILABLE SECTIONS OF
THE CLASSIC WORK

The Art of Computer Programming

VOLUME 1
MMIX
A RISC Computer for the
New Millennium

FASCICLE 1

DONALD E. KNUTH

This page intentionally left blank

an instruction a few lines away. There often is no appropriate name for nearby locations, so programmers have tended to introduce meaningless symbols like `X1`, `X2`, `X3`, etc., with the potential danger of duplication.

5. The reference to `Data_Segment` on line 11 introduces another new idea. In most embodiments of `MMIX`, the 2^{64}-byte virtual address space is broken into two parts, called *user space* (addresses #0000000000000000 .. #7fffffffffffffff) and *kernel space* (addresses #8000000000000000 .. #ffffffffffffffff). The "negative" addresses of kernel space are reserved for the operating system.

User space is further subdivided into four segments of 2^{61} bytes each. First comes the *text segment*; the user's program generally resides here. Then comes the *data segment*, beginning at virtual address #2000000000000000; this is for variables whose memory locations are allocated once and for all by the assembler, and for other variables allocated by the user without the help of the system library. Next is the *pool segment*, beginning at #4000000000000000; command line arguments and other dynamically allocated data go here. Finally the *stack segment*, which starts at #6000000000000000, is used by the `MMIX` hardware to maintain the register stack governed by `PUSH`, `POP`, `SAVE`, and `UNSAVE`. Three symbols,

```
Data_Segment = #2000000000000000,
Pool_Segment = #4000000000000000,
Stack_Segment = #6000000000000000,
```

are predefined for convenience in `MMIXAL`. Nothing should be assembled into the pool segment or the stack segment, although a program may refer to data found there. References to addresses near the beginning of a segment might be more efficient than references to addresses that come near the end; for example, `MMIX` might not be able to access the last byte of the text segment, M[#1fffffffffffffff], as fast as it can read the first byte of the data segment.

Our programs for `MMIX` will always consider the text segment to be *read-only*: Everything in memory locations less than #2000000000000000 will remain constant once a program has been assembled and loaded. Therefore Program P puts the prime table and the output buffer into the data segment.

6. The text and data segments are entirely zero at the beginning of a program, except for instructions and data that have been loaded in accordance with the `MMIXAL` specification of the program. If two or more bytes of data are destined for the same cell of memory, the loader will fill that cell with their bitwise exclusive-or.

7. The symbolic expression '`PRIME1+2*L`' on line 13 indicates that `MMIXAL` has the ability to do arithmetic on octabytes. See also the more elaborate example '`2*(L/10-1)`' on line 60.

8. As a final note about Program P, we can observe that its instructions have been organized so that registers are counted towards zero, and tested against zero, whenever possible. For example, register `jj` holds a quantity that is related to the positive variable j of Algorithm P, but `jj` is normally negative; this change

makes it easy for the machine to decide when j has reached 500 (line 23). Lines 40–61 are particularly noteworthy in this regard, although perhaps a bit tricky. The binary-to-decimal conversion routine in lines 45–55, based on division by 10, is simple but not the fastest possible. More efficient methods are discussed in Section 4.4.

It may be of interest to note a few of the statistics observed when Program P was actually run. The division instruction in line 27 was executed 9538 times. The total time to perform steps P1–P8 (lines 19–33) was $10036\mu+641543v$; steps P9–P11 cost an additional $2804\mu+124559v$, not counting the time taken by the operating system to handle `TRAP` requests.

Language summary. Now that we have seen three examples of what can be done in `MMIXAL`, it is time to discuss the rules more carefully, observing in particular the things that *cannot* be done. The following comparatively few rules define the language.

1. A *symbol* is a string of letters and/or digits, beginning with a letter. The underscore character '`_`' is regarded as a letter, for purposes of this definition, and so are all Unicode characters whose code value exceeds 126. *Examples:* `PRIME1`, `Data_Segment`, `Main`, `__`, `pâté`.

The special constructions d`H`, d`F`, and d`B`, where d is a single digit, are effectively replaced by unique symbols according to the "local symbol" convention explained above.

2. A *constant* is either

a) a *decimal constant*, consisting of one or more decimal digits {`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`}, representing an unsigned octabyte in radix 10 notation; or

b) a *hexadecimal constant*, consisting of a number sign or hash mark `#` followed by one or more hexadecimal digits {`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, `F`}, representing an unsigned octabyte in radix 16 notation; or

c) a *character constant*, consisting of a quote character `'` followed by any character other than newline, followed by another quote `'`; this represents the ASCII or Unicode value of the quoted character.

Examples: `65`, `#41`, `'A'`, `39`, `#27`, `''''`, `31639`, `#7B97`, `'算'`.

A *string constant* is a double-quote character `"` followed by one or more characters other than newline or double-quote, followed by another double-quote `"`. This construction is equivalent to a sequence of character constants for the individual characters, separated by commas.

3. Each appearance of a symbol in an `MMIXAL` program is said to be either a "defined symbol" or a "future reference." A *defined symbol* is a symbol that has appeared in the `LABEL` field of a preceding line of this `MMIXAL` program. A *future reference* is a symbol that has not yet been defined in this way.

A few symbols, like `rR` and `ROUND_NEAR` and `V_BIT` and `W_Handler` and `Fputs`, are predefined because they refer to constants associated with the `MMIX`

hardware or with its rudimentary operating system. Such symbols can be redefined, because MMIXAL does not assume that every programmer knows all their names. But no symbol should appear as a label more than once.

Every defined symbol has an equivalent value, which is either *pure* (an unsigned octabyte) or a *register number* ($0 or $1 or ... or $255).

4. A *primary* is either

a) a symbol; or

b) a constant; or

c) the character @, denoting the current location; or

d) an expression enclosed in parentheses; or

e) a unary operator followed by a primary.

The unary operators are + (affirmation, which does nothing), - (negation, which subtracts from zero), ~ (complementation, which changes all 64 bits), and $ (registerization, which converts a pure value to a register number).

5. A *term* is a sequence of one or more primaries separated by strong binary operators; an *expression* is a sequence of one or more terms separated by weak binary operators. The *strong binary operators* are * (multiplication), / (division), // (fractional division), % (remainder), << (left shift), >> (right shift), and & (bitwise and). The *weak binary operators* are + (addition), - (subtraction), | (bitwise or), and ^ (bitwise exclusive-or). These operations act on unsigned octabytes; $x//y$ denotes $\lfloor 2^{64}x/y \rfloor$ if $x < y$, and it is undefined if $x \geq y$. Binary operators of the same strength are performed from left to right; thus a/b/c is (a/b)/c and a-b+c is (a-b)+c.

Example: #ab<<32+k&~(k-1) is an expression, the sum of terms #ab<<32 and k&~(k-1). The latter term is the bitwise and of primaries k and ~(k-1). The latter primary is the complement of (k-1), a parenthesized expression that is the difference of two terms k and 1. The term 1 is also a primary, and also a constant, in fact it is a decimal constant. If symbol k is equivalent to #cdef00, say, the entire expression #ab<<32+k&~(k-1) is equivalent to #ab00000100.

Binary operations are allowed only on pure numbers, except in cases like $1+2 = $3 and $3−$1 = 2. Future references cannot be combined with anything else; an expression like 2F+1 is always illegal, because 2F never corresponds to a defined symbol.

6. An *instruction* consists of three fields:

a) the LABEL field, which is either blank or a symbol;

b) the OP field, which is either an MMIX opcode or an MMIXAL pseudo-op;

c) the EXPR field, which is a list of one or more expressions separated by commas. The EXPR field can also be blank, in which case it is equivalent to the single expression 0.

7. Assembly of an instruction takes place in three steps:

a) The current location @ is aligned, if necessary, by increasing it to the next multiple of
   8, if OP is OCTA;
   4, if OP is TETRA or an MMIX opcode;
   2, if OP is WYDE.

b) The symbol in LABEL, if present, is defined to be @, unless OP is IS or GREG.

c) If OP is a pseudo-operation, see rule 8. Otherwise OP is an MMIX instruction; the OP and EXPR fields define a tetrabyte as explained in Section 1.3.1′, and @ advances by 4. Some MMIX opcodes have three operands in the EXPR field, others have two, and others have only one.

If OP is ADD, say, MMIXAL will expect three operands, and will check that the first and second operands are register numbers. If the third operand is pure, MMIXAL will change the opcode from #20 ("add") to #21 ("add immediate"), and will check that the immediate value is less than 256.

If OP is SETH, say, MMIXAL will expect two operands. The first operand should be a register number; the second should be a pure value less than 65536.

An OP like BNZ takes two operands: a register and a pure number. The pure number should be expressible as a relative address; in other words, its value should be expressible as $@ + 4k$ where $-65536 \le k < 65536$.

Any OP that refers to memory, like LDB or GO, has a two-operand form $X,A as well as the three-operand forms $X,$Y,$Z or $X,$Y,Z. The two-operand option can be used when the memory address A is expressible as the sum $Y + Z of a base address and a one-byte value; see rule 8(b).

8. MMIXAL includes the following pseudo-operations.

a) OP = IS: The EXPR should be a single expression; the symbol in LABEL, if present, is made equivalent to the value of this expression.

b) OP = GREG: The EXPR should be a single expression with a pure equivalent, x. The symbol in LABEL, if present, is made equivalent to the number of a global register that will contain x when the program begins. If $x \neq 0$, the value of x is considered to be a *base address*, and the program should not change that global register. If $x = 0$, or if x is a base address that hasn't already occurred, a new global register is allocated (as large as possible).

c) OP = LOC: The EXPR should be a single expression with a pure equivalent, x. The value of @ is set to x. For example, the instruction 'T LOC @+1000' defines symbol T to be the address of the first of a sequence of 1000 bytes, and advances @ to the byte following that sequence.

d) OP = BYTE, WYDE, TETRA, or OCTA: The EXPR field should be a list of pure expressions that each fit in 1, 2, 4, or 8 bytes, respectively.

9. MMIXAL restricts future references so that the assembly process can work quickly in one pass over the program. A future reference is permitted only

a) in a relative address: as the operand of JMP, or as the second operand of a branch, probable branch, PUSHJ, or GETA; or

b) in an expression assembled by OCTA.

```
 % Example program ... Table of primes
L IS 500          The number of primes to find
t  IS  $255     Temporary storage
n GREG      ;; Prime candidate
q GREG  /* Quotient */
r GREG  // Remainder
jj GREG 0      Index for PRIME[j]
    ⋮
  PBN  mm,2B
 LDA t,NewLn; TRAP 0,Fputs,StdOut
 CMP t,mm,2*(L/10-1) ; PBNZ t,3B;     TRAP 0,Halt,0;
```

Fig. 15. Program P as a computer file: The assembler tolerates many formats.

MMIXAL also has a few additional features relevant to system programming that do not concern us here. Complete details of the full language appear in the *MMIXware* document, together with the complete logic of a working assembler.

A free format can be used when presenting an MMIXAL program to the assembler (see Fig. 15). The LABEL field starts at the beginning of a line and continues up to the first blank space. The next nonblank character begins the OP field, which continues to the next blank, etc. The whole line is a comment if the first nonblank character is not a letter or digit; otherwise comments start after the EXPR field. Notice that the GREG definitions for n, q, and r in Fig. 15 have a blank EXPR field (which is equivalent to the single expression '0'); therefore the comments on those lines need to be introduced by some sort of special delimiter. But no such delimiter is necessary on the GREG line for jj, because an explicit EXPR of 0 appears there.

The final lines of Fig. 15 illustrate the fact that two or more instructions can be placed on a single line of input to the assembler, if they are separated by semicolons. If an instruction following a semicolon has a nonblank label, the label must immediately follow the ';'.

A consistent format would obviously be better than the hodgepodge of different styles shown in Fig. 15, because computer files are easier to read when they aren't so chaotic. But the assembler itself is very forgiving; it doesn't mind occasional sloppiness.

Primitive input and output. Let us conclude this section by discussing the special TRAP operations supported by the MMIX simulator. These operations provide basic input and output functions on which facilities at a much higher level could be built. A two-instruction sequence of the form

$$\texttt{SET \$255,}\langle\text{arg}\rangle\texttt{; TRAP 0,}\langle\text{function}\rangle\texttt{,}\langle\text{handle}\rangle \qquad (2)$$

is usually used to invoke such a function, where ⟨arg⟩ points to a parameter and ⟨handle⟩ identifies the relevant file. For example, Program H uses

```
GETA $255,String;  TRAP 0,Fputs,StdOut
```

to put a string into the standard output file, and Program P is similar.

After the TRAP has been serviced by the operating system, register \$255 will contain a return value. In each case *this value will be negative if and only if an error occurred.* Programs H and P do not check for file errors, because they assume that the correctness or incorrectness of the standard output will speak for itself; but error detection and error recovery are usually important in well-written programs.

• Fopen(*handle*, *name*, *mode*). Each of the ten primitive input/output traps applies to a *handle*, which is a one-byte integer. Fopen associates *handle* with an external file whose name is the string *name*, and prepares to do input and/or output on that file. The third parameter, *mode*, must be one of the values TextRead, TextWrite, BinaryRead, BinaryWrite, or BinaryReadWrite, all of which are predefined in MMIXAL. In the three ...Write modes, any previous file contents are discarded. The value returned is 0 if the handle was successfully opened, otherwise −1.

The calling sequence for Fopen is

$$\texttt{LDA \$255,Arg; TRAP 0,Fopen,}\langle\text{handle}\rangle \tag{3}$$

where Arg is a two-octabyte sequence

$$\texttt{Arg OCTA }\langle\text{name}\rangle\texttt{,}\langle\text{mode}\rangle \tag{4}$$

that has been placed elsewhere in memory. For example, to call the function Fopen(5,"foo",BinaryWrite) in an MMIXAL program, we could put

```
Arg  OCTA  1F,BinaryWrite
1H   BYTE  "foo",0
```

into, say, the data segment, and then give the instructions

```
LDA $255,Arg; TRAP 0,Fopen,5 .
```

This would open handle 5 for writing a new file of binary output,* to be named "foo".

Three handles are already open at the beginning of each program: The standard input file StdIn (handle 0) has mode TextRead; the standard output file StdOut (handle 1) has mode TextWrite; the standard error file StdErr (handle 2) also has mode TextWrite.

• Fclose(*handle*). If *handle* has been opened, Fclose causes it to be closed, hence no longer associated with any file. Again the result is 0 if successful, or −1 if the file was already closed or unclosable. The calling sequence is simply

$$\texttt{TRAP 0,Fclose,}\langle\text{handle}\rangle \tag{5}$$

because there is no need to put anything in \$255.

* Different computer systems have different notions of what constitutes a text file and what constitutes a binary file. Each MMIX simulator adopts the conventions of the operating system on which it resides.