MMIX Quick Reference Card

(version 2.0)

Instruction Format

Each MMIX instruction is exactly four bytes long. The first byte contains the opcode; it selects one out of 256 instructions. The next three bytes contain either 24 bit immediate data (e.g. `JMP` *Label*); or an 8 bit register number in the range 0 to 255 followed by 16 bit immediate data (e.g. `SET` *register*`,`*value*); or—the most common form—three 8 bit operands. We will use `$X`, `$Y`, and `$Z` to indicate register operands; `X`, `Y`, and `Z` to indicate immediate 8 bit data; `YZ` for immediate 16 bit data; and `XYZ` for a 24 bit data value.

Since many instructions come in two forms, one with a register `$Z` and one with an immediate unsigned 8 bit number `Z`, we use for brevity `$Z` to stand for either one of them.

Instructions that operate on unsigned integer quantities are generally named like the signed counterpart with an `U` appended to the instruction name, e.g. `ADD` and `ADDU`.

Many instructions specify an absolute memory address using `$Y` and `$Z`. The register `$Y` serves as 64 bit base address and `$Z` as offset. Both values are added to form the absolute address `$Y + $Z`. For convenience, a *Label* can be used in place of `$Y,Z` leaving the computation of `Z` and the selection of `$Y` to the assembler. It is neccessary, however, to have a global register `$Y` close enough to the target (unless the `-x` option of the assembler is used).

Assembler Directives

Is *Label* `IS` *Expression*

Declare *Label* to be a shorthand for *Expression*, e.g. one can write `count IS $3` and use `count` as a synonym for `$3`.

Location `LOC` *Expression*

Continue to assemble instructions or data at the position in memory given by *Expression*. Often used Expressions are: `#100` (where programs start) or `Data_Segment`.

Global Register *Label* `GREG` *Expression*

Set aside a new global register containing the value given by *Expression*. A commonly used expression is `@` (the current location). *Label* can be used as a name for this register.

If the initial value is zero, the value is considered dynamic and can be changed by the program. If the value is not zero, it is considered a constant that will not change during the program.

Allocating Data

Byte Data *Label* `BYTE` *Expressions*
Wyde Data *Label* `WYDE` *Expressions*
Tetra Data *Label* `TETRA` *Expressions*
Octa Data *Label* `OCTA` *Expressions*

Bytes, Wydes, Tetras, or Octas are allocated and initialized by the *Expressions* at the current location (see `LOC`); *Label* becomes a name for the allocated address.

Loading Data

Load a Byte `LDB $X,$Y,$Z`
Load a Wyde (2Byte) `LDW $X,$Y,$Z`
Load a Tetra (4Byte) `LDT $X,$Y,$Z`
Load an Octa (8Byte) `LDO $X,$Y,$Z`

The data at `$Y + $Z` is loaded into register `$X`. The address is rounded off to respect alignment restrictions. The value loaded is considered a signed integer and its sign is extended as needed.

The load instructions for unsigned quantities (`LDBU`, `LDWU`, `LDTU`, `LDOU`) do not perform sign extension.

Storing Data

Store a Byte `STB $X,$Y,$Z`
Store a Wyde (2Byte) `STW $X,$Y,$Z`
Store a Tetra (4Byte) `STT $X,$Y,$Z`
Store a Octa (8Byte) `STO $X,$Y,$Z`

The least significant 1, 2, 4, or 8 byte from register `$X` are stored to Address `$Y + $Z`. If the target address is not properly aligned, it is rounded off to the next valid address.

These operations can cause an overflow, while unsigned store operations never cause overflow.

Setting Registers

Get `GET $X,Z`
Put `PUT X,$Z`
Set `SET $X,$Z`

`GET` moves the value of special register number `Z` to register `$X` and `PUT` moves the value of register `$Z` to special register `X`. Using predefined constants `rA`, `rB`, ... for the different register numbers, you can write e.g. `GET $1,rR`. `SET` will transfer the value of register `$Z`, or of the immediate 16 bit constant `YZ`, to register `$X`.

Load Address `LDA $X,`*Label*

`LDA` computes an absolute address like a load or store instruction and puts it in register `$X`. It is assembled as `ADDU`.

Get Address `GETA $X,`*Label*

`GETA` computes a relative address (similar to a Branch instruction) and moves the result into register `$X`.

Integer Arithmetic

Add `ADD $X,$Y,$Z`
Subtract `SUB $X,$Y,$Z`
Multiply `MUL $X,$Y,$Z`
Divide `DIV $X,$Y,$Z`

`$Y` *operation* `$Z` is computed on integers, where the *operation* is either addition, subtraction, multiplication, or division. The result is stored in register `$X`. An immediate value `Z` is always taken as an unsigned value.

Arithmetic exceptions are reflected in register `rA`. Arithmetic on unsigned integers never causes exceptions.

The remainder of the `DIV` operation is stored in register `rR`. `rH`. `DIVU` performs a 128 bit division prepending register `rD` to `$X`.

The high 64 bit of the `MULU` operation is stored in register `rH`.

Negate `NEG $X,$Z`

`NEG` computes the 2-complement of `$Z`.

Compare `CMP $X,$Y,$Z`

`CMP` will set register `$X` to -1, 0, or 1 depending on whether `$Y` is less than, equal to, or greater than `$Z`.

Bitwise Operations

And `AND $X,$Y,$Z`

`AND` computes the bitwise And of `$Y` and `$Z`; the result is stored into register `$X`. Similarly, the operations `OR`, `XOR`, `NOR`, `NAND`, `ANDN` (and not), `ORN` (or not) and `NXOR` (not xor) are provided. The bitwise complement can be computed using e.g. the `NOR` instruction with immediate operand 0.

Shift Left `SL $X,$Y,$Z`
Shift Right `SR $X,$Y,$Z`

All bits of register `$Y` are shifted left or right by `$Z` bits; the result is stored into register `$X`. If bits (other than the sign bits) are lost in a `SL` operation an arithmetic exception is raised. The signed shift right operation will extend the sign of the operand. The unsigned instructions raise no exceptions and perform no sign extension.

Floating Point Arithmetic

Add `FADD $X,$Y,$Z`
Subtract `FSUB $X,$Y,$Z`
Multiply `FMUL $X,$Y,$Z`
Divide `FDIV $X,$Y,$Z`
Remainder `FREM $X,$Y,$Z`
Square Root `FSQRT $X,$Z`
Round to integer `FINT $X,$Z`

These operations perform IEEE/ANSI Standard 754 floating point operations on `$Y` and `$Z` and store the result in `$X`.

Float from integer `FLOT $X,$Z`
Fixed integer from float `FIX $X,$Z`

`FLOT` converts the signed integer `$Z` to a float in `$X` and `FIX` converts a float in `$Z` to a signed integer in `$X`. Both operations are available as unsigned variants.

The instructions above that have only a single operand `$Z` optionally can use the value of `Y` to specify a rounding mode.

Compare `FCMP $X,$Y,$Z`
Compare using `rE` `FCMPE $X,$Y,$Z`
Equal `FEQL $X,$Y,$Z`
Equal using `rE` `FEQLE $X,$Y,$Z`

`FCMP` is similar to `CMP` setting `$X` to -1, 0, or +1 depending on whether `$Y` is less than, equal to, or greater than `$Z`.

`FEQL` checks if `$Y` and `$Z` are equal and sets `$X` to 1 in this case, otherwise `$X` becomes zero.

The same instructions with the suffix `E` use the epsilon register `rE` to specify the size of an Interval around `$Y` and `$Z`. Then the two intervalls are compared.

Store Short Float	`STSF $X,$Y,$Z`
Load Short Float	`LDSF $X,$Y,$Z`
Short Float from integer	`SFLOT $X,$Z`

For improved memory efficiency, loading and storing of 32 bit floating point numbers, called Short Floats, is supported. `STSF` rounds the 64 bit float in `$X` to a 32 bit float and stores this number at `$Y + $Z`; `LDSF` reverses this operation. `SFLOT` is similar to FLOT, but includes rounding to a 32 bit float before placing the result in `$X` as a 64 bit float.

Jumps and Branches

Jump relative `JMP` *Label*

`JMP` jumps unconditionally to the instruction at position *Label*. The *Label* is a three byte relative address, usually computed by the assembler. Different Opcodes exist for jumping forward and backward, so the full 24 Bits are available to specify the number of instructions to jump in either direction. For even bigger jumps, the `GO` instruction can be used (see "Linking Subroutines").

Branch if Zero `BZ $X,`*Label*

`BZ` tests the value of register `$X` and, if zero, causes the program to branch to the location given by *Label*, a 16 bit relative address, otherwise the program will continue with the next instruction.

Further branch instructions are `BP` (positive), `BN` (negative), `BEV` (even), `BOD` (odd), `BNN` (non negative), `BNP` (non positive), and `BNZ` (non zero). The "branch if negative/positive" test the leftmost bit of `$X` which indicates the sign of `$X`. The "branch if odd/even" test the rightmost bit of `$X`.

As an optimization, a `P` can be prefixed to the instruction, as a hint that the branch will probably be taken; otherwise branch prediction will assume that the branch is not taken.

Subroutines

Push and Jump	`PUSHJ $X,`*Label*
Pop and return	`POP X,YZ`

`PUSHJ` transfers control to the adress specified by *Label*, a 16 bit relative address, similar to a `JMP` instruction.

The address of the instruction following the `PUSHJ` is placed in register `rJ` which is used by the `POP` instruction to return to address `rJ + 4*YZ` (`YZ` is typically 0).

For managing local variables, passing parameters, and returning values, `PUSHJ` and `POP` implement a register stack.

The `X` value of the `PUSHJ` instruction specifies the size of the callers stack frame. Registers `$0` to `$(X-1)` are considered the callers local variables, which become inaccessible during subroutine execution and are fully restored after return. The registers `$(X+1)`, `$(X+2)` ... are meant to contain the subroutine parameters. `PUSHJ` will renumber these registers so that the subroutine will find them allways in registers `$0`, `$1`, ...

The `X` value of the `POP X,YZ` instruction specifies the number of return values stored in `$0` to `$(X-1)`. To restore the callers stackframe, `POP` will undo the renumbering of registers. Hence, registers `$0`, `$1`, ... become `$(X+1)`, `$(X+2)`, ... again. To improve register usage for the common case of a single return value, `POP` moves the "main result" in register `$(X-1)` down the register stack so that the caller finds it in register `$X` (as specified by `PUSHJ $X,YZ`).

`POP` will trim the register stack: it makes all local registers marginal except those in the callers stack frame and those that contain return values.

Global registers are not affected by the renumbering.

If the `PUSHJ` instruction specifies a global register `$X`, only local registers will be renumbered, not the marginal registers. In this case, passing of parameters and return values should be done in global registers.

Push and Go	`PUSHGO $X,$Y,$Z`
Go	`GO $X,$Y,$Z`

`PUSHGO` is similar to `PUSHJ` but computes `$Y + $Z` as an absolute target address. It can be used to accomplish subroutine calls where a 16 bit relative address is not sufficient. `GO` jumps to the absolute address `$Y + $Z` as well, but otherwise is more like a `JMP` (no register renumbering), except that the address of the instruction immediately following the `GO` is stored in `$X`. The value in `$X` can then be used as a return address. The return to `$X` can be accomplished with the `GO` instruction as well. The symmetry between call and return lends itself to the implementation of coroutines.

Calling the Operating System

Halt the Program `TRAP 0,Halt,0`

Calls to the operating system are accomplished with the `TRAP` instruction. Register `$255` is used to pass parameters and return values. If more parameters are needed, the arguments must be placed in consecutive `OCTA`'s in memory, the parameter block, and `$255` must contain the address of the first `OCTA`. The `Y` value is commonly used to specify the operation and `Z` is used as an auxiliary parameter.

If `XYZ` is zero (`Halt`, used above, is defined to be zero) the program terminates with exit code `$255`.

If `XYZ` is one, a default handler for `TRIP` interrupts is invoked. The effect for other values of `XYZ` depend on the operating system. MMIXWare specifies routines for a few such values.

Here are some examples:

Output a string `TRAP 0,Fputs,StdOut`
Before using this instruction, the temporary global register `$255` must be set to the address of a null terminated string. For example:

```
Greeting   BYTE  "Hello␣World!",0
           LDA   $255,Greeting
           TRAP  0,Fputs,StdOut
```

After the `TRAP` instruction, register `$255` contains the number of characters send to the output; it is negative if and only if an error occurred.

Input a string `TRAP 0,Fgets,StdIn`
Before using this instruction, the temporary global register `$255` must be set to the address where the buffer address (not the buffer itself) and the size of the buffer is stored. The instruction will read one line of input, but not more characters than the given size of the buffer. For example:

```
InSize     IS    100
InBuffer   BYTE  0
           LOC   InBuffer+InSize
InArgs     OCTA  InBuffer,InSize
           LDA   $255,InArgs
           TRAP  0,Fgets,StdIn
```

Further system calls defined by MMIXware are:

File open	`Fopen(`*handle*`,`*name*`,`*mode*`)`
File close	`Fclose(`*handle*`)`
File read	`Fread(`*handle*`,`*buffer*`,`*size*`)`
File write	`Fwrite(`*handle*`,`*buffer*`,`*size*`)`

Where *mode* is a predefined constant (`TextRead`, `TextWrite`, `BinaryRead`, `BinaryWrite`, and `BinaryReadWrite`).

Name Spaces

Set Prefix `PREFIX` *Name*

`mmixal` has two kind of names. Fully qualified names start with a colon, like "`:here`", all other names are automatically extended by adding the current prefix to it. When the assembler starts, the current prefix is ":". It can be set to a new value by the `PREFIX` *Name* directive. Note that the old current prefix is used to extend *Name*, if it is not fully qualified, before it replaces the old current prefix.

The following example defines the fully qualified names:

`:count`, `:sub:return`, `:sub:routine:count`, and `:Main`.

Note that e.g. the predefined name `rJ` (Jump register) needs to be written as `:rJ` unless the current prefix is ":".

```
count      IS     $1
           PREFIX sub:
return     IS     :rJ
           PREFIX routine:
count      IS     $3
           PREFIX :
Main       PUT    rJ,0
```