VMAX: A VIRTUAL MACHINE FOR THE PCMAX2

Version 2.00

1990 July 16

Roger House

Everex

Sebastopol R&D

707-823-0733

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

VMAX: A Virtual Machine for the PCMAX2

Introduction

VMAX is one component of a system which will allow a programmer to compile and execute C programs on a PCMAX2 board on a PC running DOS. This system is based on a C compiler, GCC, which can be tailored to generate code for most any 32-bit machine that addresses 8-bit bytes and has several general registers.

Since the PCMAX2 has a rather small code space (the Writable Control Store or WCS) and a rather small data space (the DataRam), it is not feasible to tailor GCC to generate PCMAX2 microcode. Instead, GCC is tailored to generate code for a virtual computer, the VMAX, which is then executed on the PCMAX2 by an interpreter which simulates the VMAX.

This document describes the architecture of VMAX: The address space, register structure, instruction formats, and detailed actions of all instructions. The intention is that this document should provide a complete and detailed description of VMAX which contains all the information needed for a programmer to write a VMAX interpreter.

Although VMAX is a general-purpose computer which could be described independently of both the PCMAX2 and GCC, it does not seem to be a good idea to treat VMAX as if if exists in a vacuum. The only reason the VMAX computer has been designed is so that GCC can be ported to the PCMAX2. Thus, both GCC and the PCMAX2 are mentioned frequently in this document.

Design Criteria

The design of the VMAX architecture was driven by two major requirements:

1. The VMAX interpreter on the PCMAX2 must be efficient in terms of speed.

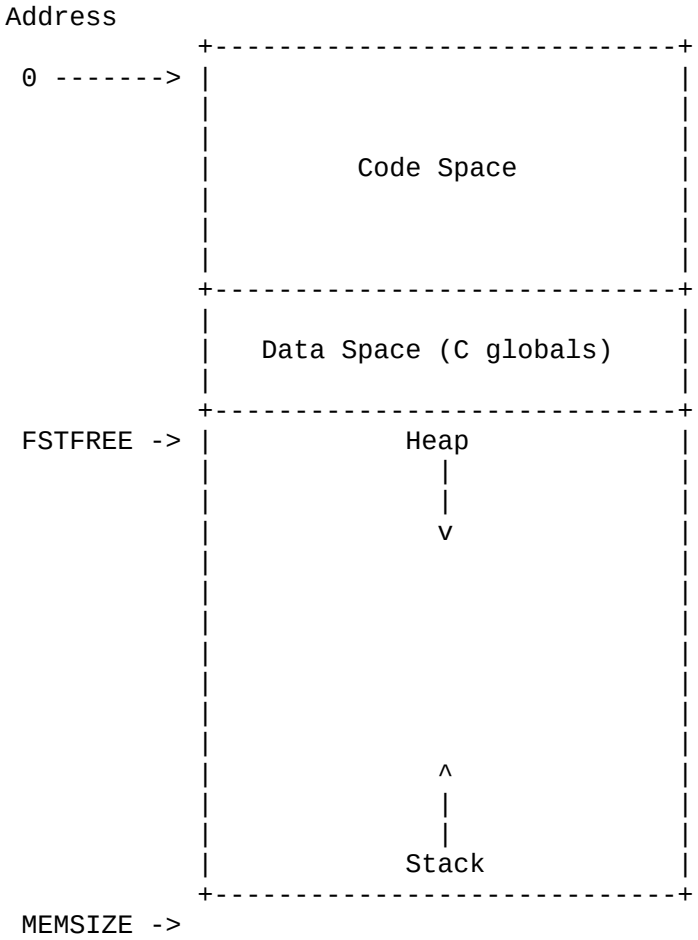
2. It must be possible to describe VMAX to GCC so that GCC can generate reasonably good code for the machine.

Although considerable care was taken to satisfy these requirements, it is anticipated that the definition of VMAX will be an iterative process. The machine definition will undoubtedly change as experience is gained with GCC and as the interpreter is developed.

VMAX Memory and Address Space

The VMAX memory is organized as a flat 32-bit address space of up to 2^32 bytes. The first byte has address 0, and the last byte address 2^32-1. Each byte can be addressed, so in the general case an address requires 32 bits (= 4 bytes). Data operands have no alignment requirements, e.g., a 2-byte word operand need not be word-aligned in memory. However, all instructions must be word-aligned (all instructions occupy an even number of bytes).

VMAX programs generated by the C compiler organize the VMAX memory like this:



It should be noted that there is nothing in the VMAX architecture itself which forces the above memory organization (except, perhaps, the fact that the VMAX stack grows from high addresses to low addresses). This organization is very convenient and efficient for programs generated from C

source code, but if one were to write programs directly in VMAX assembly language, it would be possible to intermingle code, data, and the stack in anyway the programmer desired.

The VMAX address space is mapped onto the PCMAX2 Vram: Address zero of VMAX memory is address zero of Vram. Thus the Vram is the VMAX memory.

The VMAX stack begins at the highest available address of Vram and grows downward towards lower addresses.

The VMAX interpreter can use PCMAX2 DataRam in a number of ways: All VMAX registers might be stored in DataRam, or some of them might be stored in PCMAX2 registers. Portions of VMAX memory might be cached in DataRam, e.g., the memory near the current top of the stack, and the memory near the current value of the program counter. However, these are all implementation issues, more or less independent of the definition of VMAX, so no more will be said about them at this time.

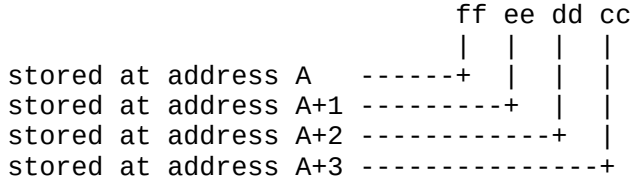
One advantage of the mapping from VMAX memory to Vram is that VMAX executable programs do not need to be relocated. They always load at byte zero of Vram.

Operand Types

VMAX instructions operate on 9 operand types. The following table lists the types and their corresponding C declarations:

VMAX data type	Size (bytes)	C declaration
signed byte	1	signed char
unsigned byte	1	unsigned char
signed word	2	short
unsigned word	2	unsigned short
signed long	4	long
unsigned long	4	unsigned long
float	4	float
double	8	double
address	4	pointer to whatever

The float and double data types use the standard IEEE 754 formats for single-and double-precision floating-point numbers. All other data types are integer types which are stored in 2's complement form. VMAX is a "little endian" machine, which means that the loworder bytes of a data item are stored at lower addresses than the highorder bytes of the item. In other words, the order of the bytes in memory is opposite to the order in which numbers are usually written. As an example, consider these four bytes stored at memory address A:



The following table shows how these bytes are interpreted when address A appears in an instruction requiring an operand of a specific data type:

operand type	order in mem.	usually written as	value
signed byte	ff	ff	-1
unsigned byte	ff	ff	+255
signed word	ffee	eeff	-4,353
unsigned word	ffee	eeff	+61,183
signed long	ffeeddcc	ccddeeff	-857,870,593
unsigned long	ffeeddcc	ccddeeff	+3,437,096,703

[Note: In this document, unless stated otherwise, numbers are written in the usual order expected by humans, not in the little endian order in which they are actually stored in memory.]

A data item representing an address is the same as an unsigned long. GCC treats the C type "int" as a 32-bit integer, i.e., as a long. Thus there is little mention of "int" in this document, because an "int" is always a long.

As already mentioned, there are no alignment requirements for data operands. An operand may begin at any address in VMAX memory.

VMAX Registers

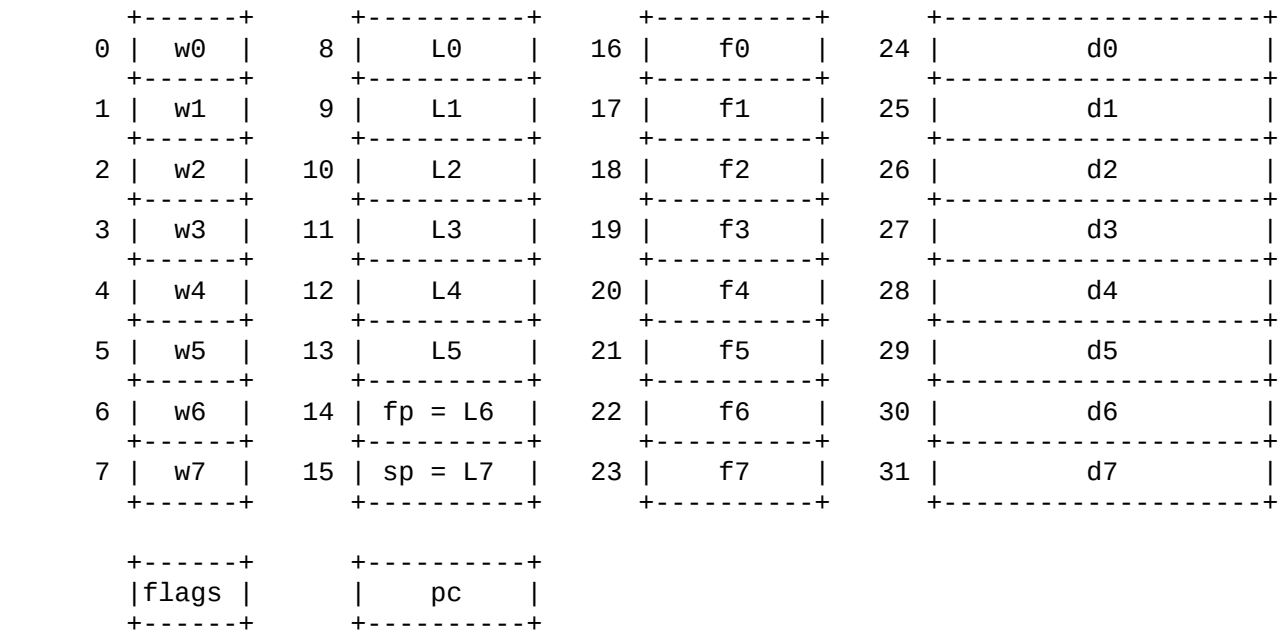
Since GCC is very good at optimizing register usage, VMAX offers a fairly large number of registers:

```
        general registers
            8 16-bit word registers:    w0, w1, w2, w3, w4, w5, w6, w7
            8 32-bit long registers:    L0, L1, L2, L3, L4, L5, L6, L7
            8 32-bit float registers:   f0, f1, f2, f3, f4, f5, f6, f7
            8 64-bit double registers:  d0, d1, d2, d3, d4, d5, d6, d7

        other registers
            1 32-bit program counter:   pc
            1 16-bit flags register:    flags
                                         LUF:  less than unsigned flag
                                         LF:   less than signed flag
                                         EF:   equal flag
                                         GF:   greater than signed flag
                                         GUF:  greater than unsigned flag
```

The two long registers L6 and L7 are special in that some instructions use them as implicit operands. Thus, these registers are usually denoted by fp and sp:

```
               fp (= L6) is the frame pointer
               sp (= L7) is the stack pointer
```

The following diagram shows all the VMAX registers:



Note that all registers other than the flags and pc registers are numbered from 0 through 31. This numbering is used in a few instructions which reference registers of any type (most instructions only reference registers of one type).

General Registers

The decision to provide several types of registers for VMAX was based primarily on the fact that the PCMAX2 is a word machine (word = 2-bytes), which means that operations on operands longer than a word are inherently slower than operations on words. Thus, by providing word registers on the VMAX, operations on C short integers will execute faster than operations on long (4-byte) integers. Also, the new ANSI definition of C allows operands of type float to be operated on with single-precision floating point operations rather than double-precision operations. Thus, when the extra precision of doubles is not needed, speed can be gained by using floats.

VMAX has no byte registers as such. However, individual bytes can be moved to and from the loworder bytes of word and long registers, so all byte operations can be handled. For example, to implement A = B + C, where all three variables are unsigned characters, this VMAX code can be used:

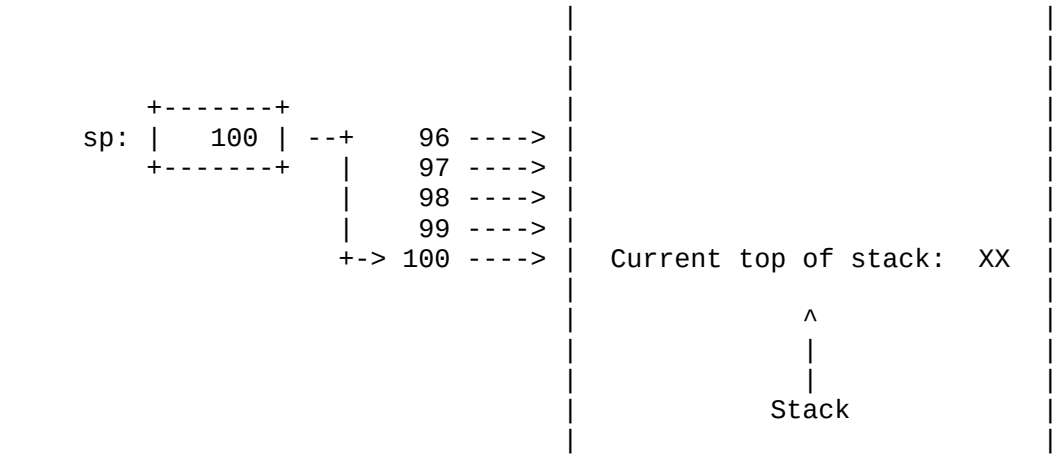
```
        movbzw  B       w0      ; Move byte B zero-extended to word w0
        movbzw  C       w1      ; Move byte C zero-extended to word w1
        addw    w1      w0      ; Add word w1 to w0
        stowb   w0      A       ; Store loworder byte of w0 into byte A
```

Stack Pointer

Since the primary aim of VMAX is to execute C programs, it is almost imperative that VMAX have a hardware stack. The stack pointer register is an unsigned long register that contains the memory address of the top word in the stack. (The stack grows from high addresses to low addresses, so the "top" of the stack is at a lower address than other elements on the stack.)

Push and pop operations handle variable-length data items, e.g., push word, push long, push float, and push double. A word is the shortest item that can be pushed or popped; there are no instructions to push or pop single bytes.

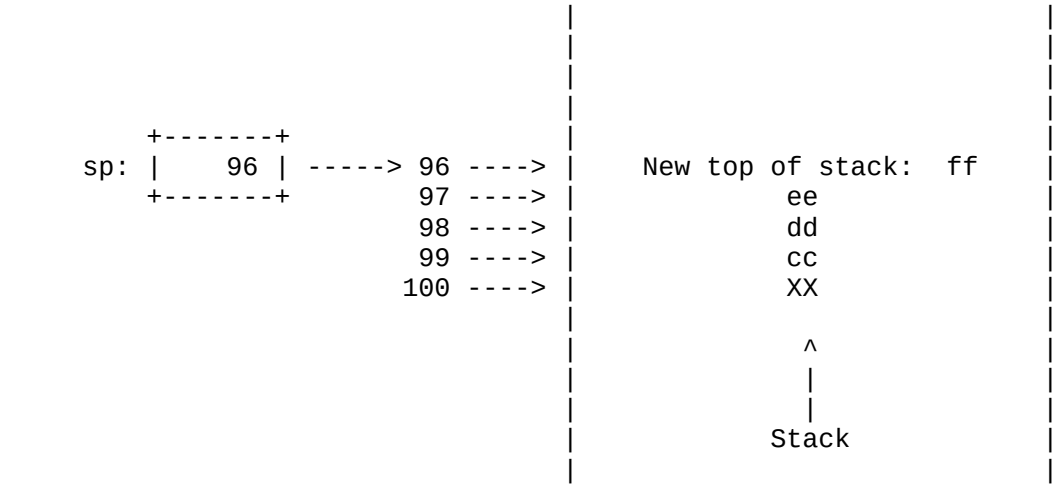
As an example of how the stack pointer changes, consider this situation:



The stack pointer contains the value 100, which means that the current top of the stack is at memory address 100. Now say that the instruction

```
pushl 0xccddeeff
```

is executed to push a long value onto the stack. After this instruction is executed, the stack looks like this:



The pushl operation decrements sp by 4 and moves 4 bytes to the address contained in sp. The inverse operation, popl, fetches the 4 bytes starting with the byte addressed by sp, and then increments sp by 4.

Note that the long value 0xccddeeff is pushed onto the stack with its low-order byte at a lower address than its highorder byte. This is essential to maintain the "little endian" order of operands in memory.

Since the VMAX has a flat 32-bit address space, there are no segment registers. In particular, there is no register containing the base of the stack. The stack pointer is an absolute memory address, not an offset relative to some base register. If it is conceptually helpful, the stack can be considered to have a base address of zero.

Frame Pointer

A language like C can be implemented easily enough without a frame pointer, provided that memory can be accessed relative to sp (which is not the case for 80x86 processors). However, GCC seems to require a frame pointer, so VMAX provides one, namely the fp register. This is a 32-bit unsigned long register which usually contains a memory address (almost always the address of something on the stack). Addressing modes exist for accessing operands relative to fp.

To illustrate the use of fp, consider the following C function prototype, definition, and call:

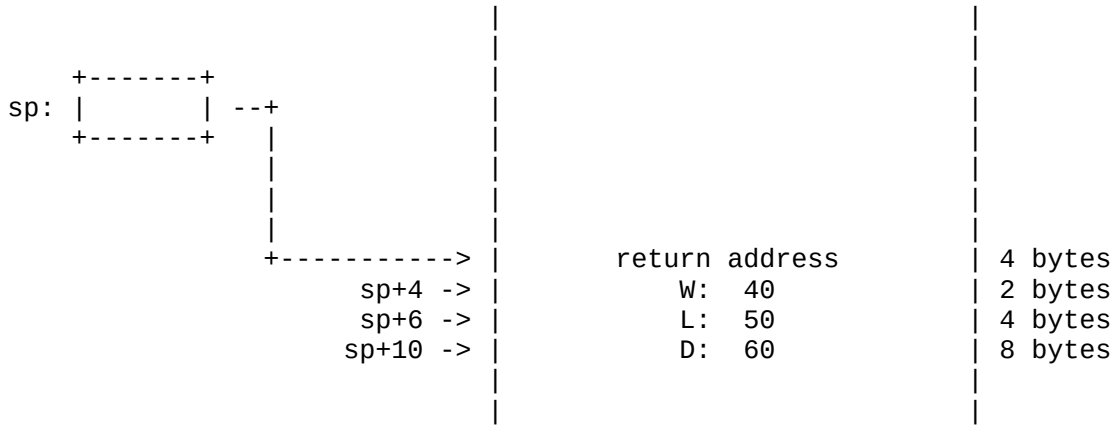
```
void F(short W, long L, double D);


void F(short W, long L, double D)
{
        short   WA;
        long    LA;

        . . .

} /* end F */


F(40, 50L, 60.0);
```

The call of F generates the following VMAX code:

```
pushd   60.0            ; Push third parameter onto stack
pushl   50              ; Push second parameter onto stack
pushw   40              ; Push first parameter onto stack
call    F               ; Call function F
addl    14      sp      ; Clear parameters from the stack
```

Note that the parameters in the call of F are pushed onto the stack in reverse order, so that the first parameter is at the top of the stack when F is entered. The parameters occupy a total of 14 bytes on the stack (2 for W, 4 for L, and 8 for D), so after the call, sp is incremented by 14 to clear the parameters from the stack. These conventions are usual in C implementations because a function may have a variable number of parameters.

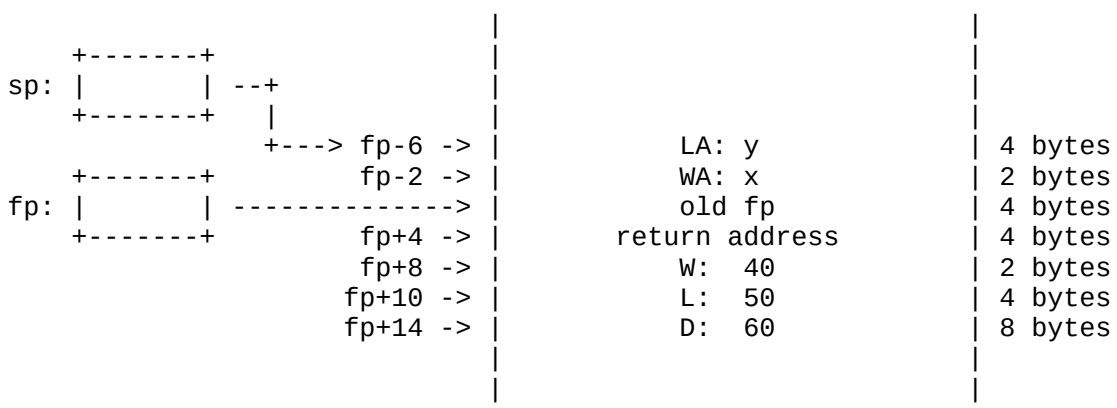
When function F is entered, the stack looks like this:



Here is the code generated for the definition of function F:

```
F:      pushl   fp              ; Save old fp on the stack
        movl    sp      fp      ; Set fp to sp
        subl    6       sp      ; Reserve stack space for locals
        . . .
        movl    fp      sp      ; Clear locals from the stack
        popl    fp              ; Restore old fp
        ret                     ; Return to caller
```

After the subl instruction at the beginning of F is executed, the stack looks like this:



Now, all parameters and local variables of F can be accessed relative to fp. Parameters are accessed with positive offsets, and locals are accessed with negative offsets.

The code at the end of the function clears the local variables from the stack (movl fp sp), restores the old fp (popl fp), and returns to the caller (ret).

A few notes on the preceding:

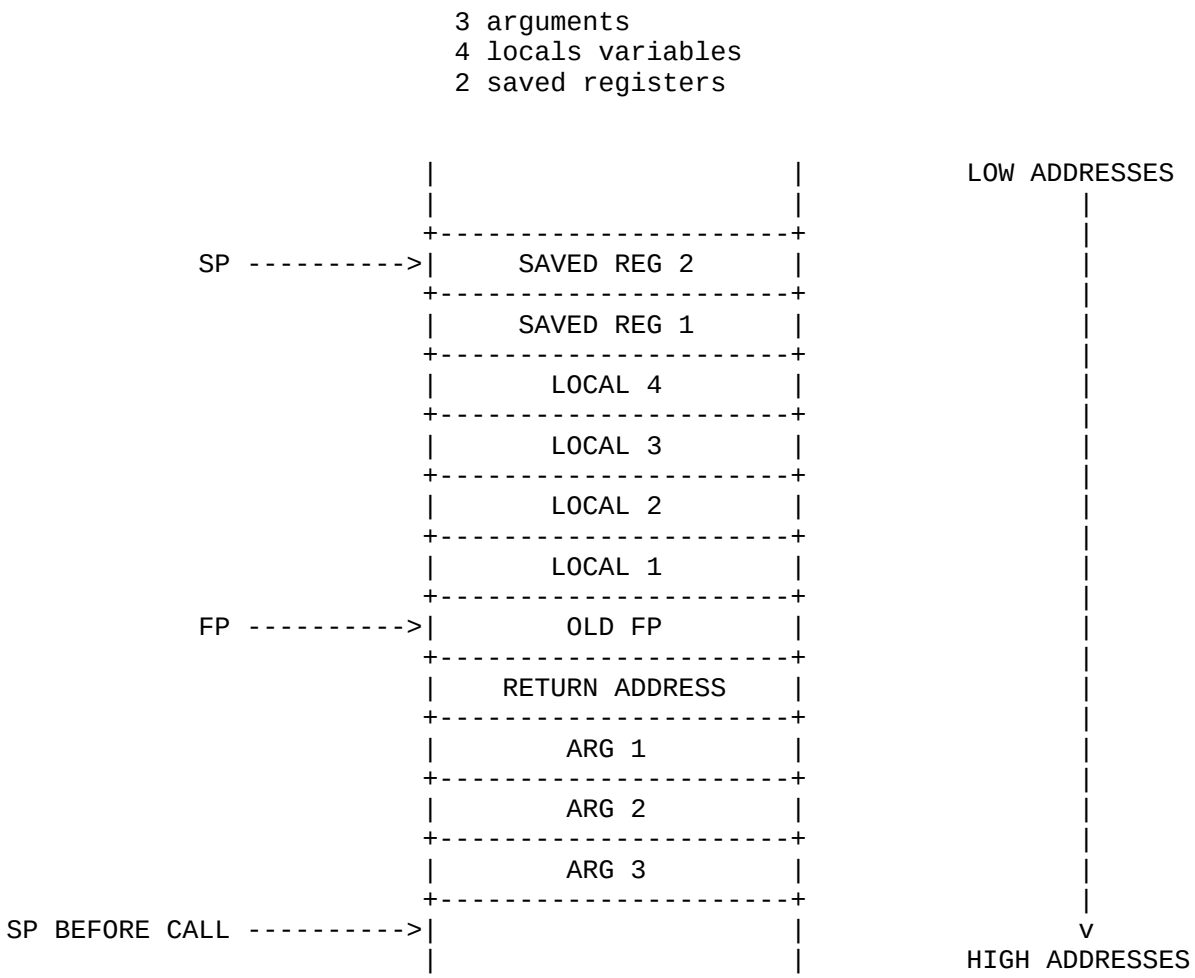
1. The function prologue and epilogue shown in the example above are not what are actually generated for VMAX. Instead the enter, entersav, leave, and leaveres instructions are used. These instructions perform the same actions as shown in the example, but they require quite a bit fewer bytes of code.

2. Traditional C required that all integer parameters shorter than int in a function call be promoted to int. Thus parameter W in the preceding example would be promoted to an int and pushed onto the stack as 4 bytes instead of as 2 bytes. Similarly, floats were promoted to doubles. However, ANSI C now allows shorts and floats to be passed as they are without promotion (if a prototype for the function specifies the parameters as shorts and floats). The example shown assumes this new standard.

3. Traditional C required that the code for every function be prepared to handle a variable number of arguments. This has changed with ANSI C: If a function prototype does not explicitly indicate that the function takes a variable number of parameters, the compiler can assume that the function takes a fixed number of parameters of fixed types, namely those specified by the prototype. This means that it is now possible for a function to clear its own parameters off the stack, instead of requiring the caller to do this. Although the example above shows the traditional C implementation, GCC will generate code for the new standard when appropriate.

[A note on sp and fp: In the preceding, it has been stated that sp and fp are unsigned long registers. This is not quite correct. In the vast majority of cases sp and fp contain memory addresses, which are unsigned longs. However, all instructions which operate on longs may have either sp or fp as an operand, which means that these two registers are treated like any other long registers. Thus, when it is useful, their contents can be thought of as signed longs.]

GCC requires that, when necessary, a function preserve certain registers, so part of the function prologue saves registers on the stack. Here is a diagram showing a typical stack frame just after the function prologue has been executed for a function with:

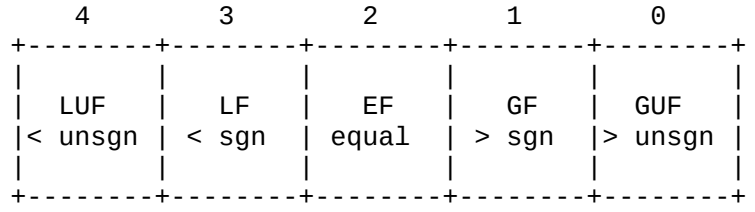


Program Counter

The program counter register, pc, is an unsigned long register which always contains the memory address of the next instruction to execute. The only way this register can be changed under program control is by jump, call, leave, leaveres, and return instructions. Since instructions are always word-aligned and consist of an even number of bytes, the value of pc is always an even number. In addition, when a C program is interpreted, pc must always point to an address in the Code Space (see a figure in an earlier section).

Flags Register

The flags register is an unsigned word register whose highorder 11 bits are always zero. The loworder five bits are used to flag various conditions:



Traditionally, computers tend to have a zero flag, a sign flag, an overflow flag, and a carry flag, all of which are changed by many instructions. Because of the nature of the PCMAX2, it would be rather expensive in terms of time to implement this style of flags. Thus, another approach has been taken:

1. The ONLY instructions which change the flags register are compare instructions.

2. Floating point compares set exactly one of the LF, EF, or GF bits, and clear all the others.

3. Integer compares work like this:

   If the operands are equal, EF is set and all other flags are cleared.

   If the operands are not equal, one of (LF, GF) and one of (LUF, GUF) are set, and all other flags are cleared.

Thus, after an integer compare, it is possible to determine the relationship between the two operands as either signed quantities or unsigned quantities.

This scheme makes it possible to avoid the overhead of setting flag bits after most instructions. Only the compare instructions set the flags. This fits in very well with the way GCC works: It handles conditional branches using the model "compare and branch on the result of the compare".

Of course, the flags scheme described here is not really completely adequate in general. For example, there are no provisions for checking for overflow of arithmetic operations. This is not a problem for GCC because the C language makes no provisions for overflow. However, for general programming it is imperative that some sort of overflow checking be provided on some basis. This will be addressed in a future version of VMAX.

[At present VMAX has no control flags for floating point operations. The IEEE 754 floating point standard requires a rather extensive set of control flags, exception flags, etc. These will be added as the VMAX floating point package evolves. Note that we may end up with a flags register of 32 bits rather than 16 bits, or perhaps we will have two flags registers, one for floating point and one for fixed point.]

Addressing Modes

The typical VMAX instruction has two operands, one of which is almost always a register. The other operand is a general operand which can be any of the following:

- register
- immediate operand
- memory address
  - absolute address of the operand
  - based addressing:
    - base_register
    - base_register + displacement
  - indexed addressing:
    - index_register
    - index_register + displacement
  - based and indexed addressing:
    - base_register + index_register
    - base_register + index_register + displacement

All long registers can be used as both base registers and index registers. When indexing is used, the value of the index register can be scaled by a factor of 1, 2, 4, or 8.

The displacements used in based, indexed, and based-indexed addressing modes are of varying lengths. The following list shows the displacement sizes available, and the notation commonly used for these addressing modes:

Based Addressing

b	base
bd1	base + 1-byte-disp
bd2	base + 2-byte-disp
bd3	base + 3-byte-disp
bd4	base + 4-byte-disp

Indexed Addressing

i	index
id1	index + 1-byte-disp
id3	index + 3-byte-disp
id4	index + 4-byte-disp

Indexed Addressing

bi	base + index
bid2	base + index + 2-byte-disp
bid4	base + index + 4-byte-disp

Instruction Formats

General Considerations

This section discusses the general characteristics of VMAX instructions, as well as the reasons behind the overall design of the instruction formats.

Instruction length:
: All VMAX instructions occupy an even number of bytes; the possible lengths are 2, 4, 6, 8, and 10 bytes (the latter length only occurs when an instruction contains an 8-byte double as an immediate operand). The primary reason that instructions are an even number of bytes long is that the PCMAX2 Vram (where all VMAX instructions are stored) can only be accessed at the word level, not at the byte level. Thus, we increase interpretation speed by using a bit of space (some instructions could be one byte shorter if an odd number of bytes were allowed in an instruction).

Separate opcode byte:
: The first byte of every instruction is an opcode; no other information is encoded in the first byte (although opcodes often imply operand type). Thus, the opcode byte can be used as an index to a jump table in the PCMAX2 DataRam which contains WCS addresses of routines for interpreting opcodes. At present about 145 VMAX opcodes have been defined, and it is anticipated that at most 40 more will be needed. This leaves at least 60 unused opcode values, which allows plenty of leeway for later optimization of the interpreter after experience shows what new instructions would speed up applications.

Data types indicated by opcodes:
: Instead of encoding operand type information in the operand bytes of an instruction, each opcode operates on a specific data type. Thus, there are four add instructions: Add word, add long, add float, and add double. The format of the operand bytes in these four instructions is identical (except for immediate operands) but the operand bytes describe different types of data depending on the opcode. In effect, type information is carried by the opcode, not by the operand bytes. This allows a more efficient encoding of information in an instruction (see the description of the qr instruction format below).

Two-operand instructions:

Most instructions have two operands, one of which is always a register. The other operand is general, i.e., a register, a memory reference, or an immediate operand. Care has been taken to insure that register-to-register instructions are of minimum length, i.e., 2 bytes. For most instructions, both operands have the same type, i.e., both are words, both are longs, etc. However, some instructions exist for converting one type to another, so, of necessity, the two operands are of different types.

Uniformity:

The instruction formats are fairly uniform, adhering to general schemes with few exceptions. This cuts down on the work needed to interpret the instructions.

[A note on notation: Examples of VMAX instructions in this document are shown in a sort of pseudo assembly language, for example:

```
movw    B       w5      ; Move contents of B to w5
stow    w5      A       ; Store w5 into A
```

Mnemonics for opcodes reflect the actual opcode stored in memory, not what might normally be written in assembly language. For example, in a real assembly language, the single opcode sto might be used instead of stow, stol, stof, and stod. The assembler can figure out which opcode to generate based on the types of the operands.]

Overview of Formats

The VMAX instruction formats are summarized in the following table:

format	number operands	first operand	second operand	examples
qr	2	general	register	add, move
qc	2	general	cond. code	store cond.
qo	1	general	-	push, pop
mr	2	general	register	gmov, gsto
ir	2	imm. int	register	shift, rotate
ij	2	cond. code	jump adr	jump, call
a3	1	3-byte adr	-	jump, call
b1	1	1-byte int	-	ret
b14	2	1-byte int	4-byte msk	entersav
n0	0	-	-	halt, nop
n04	1	4-byte msk	-	pushregs

qr-format

Most instructions have the qr-format, in which the q-operand is a general operand (a register, a memory reference, or an immediate value), and the r-operand is a register. The q-operand can involve based addressing, indexed addressing, or both.

qc-format

This format is used by instructions which store a 0 or a 1 depending on the condition codes. One operand is a general q-operand, and the other specifies a condition.

qo-format

This is the "q-only" format, i.e., there is only one operand, which is a general q-operand. Push and pop instructions use this format.

mr-format
: This format is used by only two instructions: gmov and gsto. It includes all the addressing modes of the qr-format except immediate operands, and in addition allows data to be moved regardless of type (for example, 4 word registers can be moved to one double register).

ir-format
: In this format, one operand is an immediate 5-bit integer, and the other is a register. At present this format is used only by some shift and rotate instructions.

ij-format
: This format is used by all conditional jump instructions. One operand is a condition, and the other is an address to jump to. Several addressing modes are available for the address.

a3-format
: This format consists of a single operand, a 3-byte memory address. It is used by jump and call instructions.

b1-format
: There is one operand, a 1-byte integer. The enter, leave, and ret and instructions have this format.

b14-format
: This is the same as the b1-format except that in addition to the 1-byte integer operand there is a 4-byte mask operand. The entersav and leaveres instructions have this format.

n0-format
: This is the simplest format there is: There are no operands, only an opcode. An example is nop.

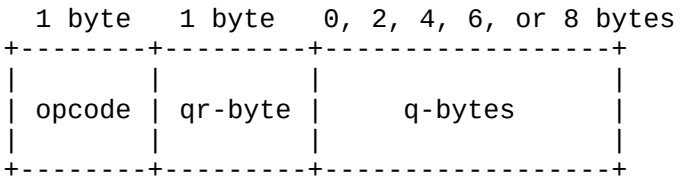
n04-format
: There is one operand, a 4-byte mask. Only the pushregs and popregs instructions have this format.

Each instruction format is described in detail in the following sections.

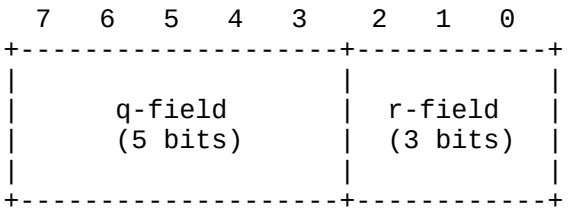
The qr Format

The qr-format is the most widely used format in the VMAX instruction set. It specifies two operands, a source and a destination, one of which is a general operand, and the other of which is a register.

A qr-format instruction consists of an opcode, a qr-byte, and, for some addressing modes, a sequence of q-bytes.



The qr-byte consists of two fields, the q-field and the r-field:



The q-field specifies the q-operand, a general operand, and the r-field specifies the r-operand, which is always a register. With a few exceptions, the q-operand is the source operand, and the r-operand is the destination operand. In some cases, e.g., the set of sto instructions which store registers to memory, the r-operand is the source, and the q-operand is the destination.

The sequence of q-bytes which may follow the qr-byte depends on the addressing mode specified in the q-field. All cases are described in detail in following sections.

Since the r-operand is simpler than the q-operand, it is described first.

The r-operand

The register specified by the r-operand is fully determined by two things: The 3-bit integer in the r-field and the operand type implied by the op-code. The following table shows the possible combinations:

r-field bit pattern	decimal value	operand type specified by opcode: word	long	float	double
000	0	w0	L0	f0	d0
001	1	w1	L1	f1	d1
010	2	w2	L2	f2	d2
011	3	w3	L3	f3	d3
100	4	w4	L4	f4	d4
101	5	w5	L5	f5	d5
110	6	w6	fp	f6	d6
111	7	w7	sp	f7	d7

As an example, consider this movl (move long) instruction:

opcode	q-field	r-field
movl	00:001	110

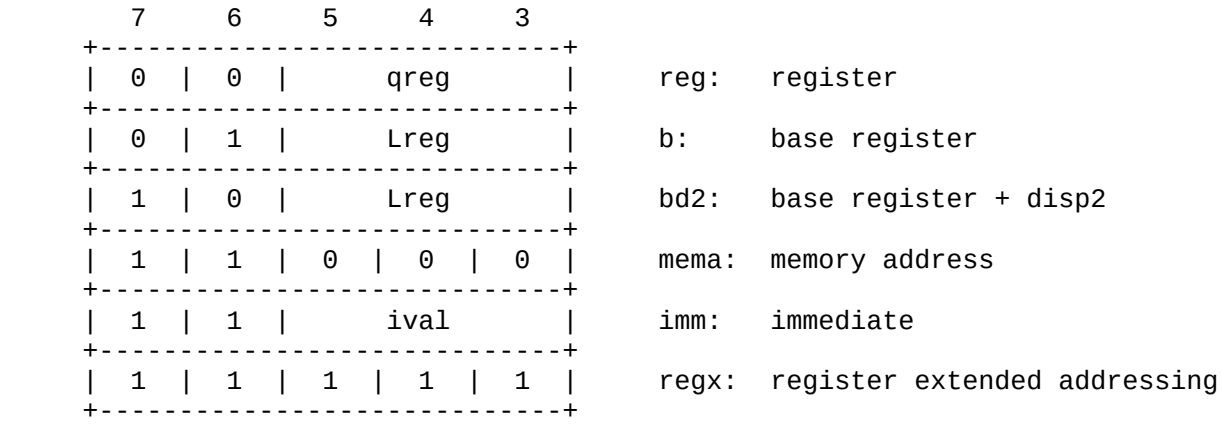
The operand type implied by the opcode is long, so the two registers appearing in the instruction are long registers. Thus the instruction moves the contents of long register 1 to long register 6, i.e., from L1 to fp.

If the opcode is changed to movw (move word), then the registers appearing in the instruction are word registers, so the instruction moves the contents of word register 1 to word register 6, i.e., from w1 to w6.

Note that fp (the frame pointer) and sp (the stack pointer) are long registers. Thus, any instruction that operates on long operands can operate on fp and sp.

The q-operand

The q-operand is more complex than the r-operand since it may be a register, an immediate value, or a value in memory accessed via one of several addressing modes. The following options are encoded in the q-field:



The regx option indicates that there are bytes following the qr-byte which specify the addressing mode. These bytes are described in detail in the following pages.

Before describing each of the q-operand options, some discussion of addresses and values is appropriate.

A q-operand can be either a value or an address. For example, when memory is moved to a register, the q-operand is a VALUE, namely, the contents of a memory location. However, when a register is moved to memory, the q-operand is an ADDRESS, namely, the address of the memory location where the register is to be stored.

For the reg and imm q-operand options, the distinction between values and addresses is not of great importance, because neither registers nor immediate values have addresses (although by stretching things a bit we could probably define some kind of addresses for them).

However, for all other q-operand options, the distinction between values and addresses is important, because these kinds of q-operands are addressing modes. Thus, for each mode the "effective address" is described. For an instruction which requires an address as a q-operand, the effective address is the operand. For an instruction which requires a value as a q-operand, the value stored at the effective address is the operand. The notation EA will be used for "effective address".

Now that the distinction between addresses and values is clear, each of the q-operand options are described in detail:

* reg: register: q-operand is register contents

The 3-bit qreg field contains an integer which, when combined with the operand type, specifies a register. The encoding is exactly the same as for an r-operand register, as shown in the table for r-operands on a previous page.

* b: base register: q-operand EA is long register contents

The 3-bit Lreg field contains an integer which specifies any one of the eight long registers, encoded exactly the same as for an r-operand register, as shown in the table for r-operands on a previous page.

The contents of the specified long register is the effective address. As an example of the use of this addressing mode, consider the C statement *(P+1) = B, where B is a short and P is a pointer to short (both are globals). This can be implemented in VMAX as

```
movl    P       L3
addl    2       L3      ; L3 = P + 1
movw    B       w0
stow    w0      [L3]    ; *L3 = B
```

Note that both sp and fp can be used in the Lreg field of the base register mode.

* bd2: base register + disp2: q-operand EA is long register contents plus a 2-byte displacement

This is the same as the base register mode except that the qr-byte is followed by a signed 2-byte displacement which is added to the contents of the long register specified by Lreg to determine the effective address. Conceptually the displacement is sign-extended to a 4-byte signed value before it is added to the register. (Note: The register contents are NOT changed by this mode.)

As an example, consider the following instruction which stores register w0 to memory. If L3 contains 10, then the instruction stores w0 into memory address 10+2 = 12.

```
                          opcode   qr-byte   2 bytes
                         +-------+----------+------+
                         |       |      |   |      |
    stow  w0  [L3+2]     | stow  | bd2  | 0 | 0002 |
                         |       |  L3  |   |      |
                         +-------+----------+------+
```

*   mema: memory address: q-operand EA is memory address in instruction

    The qr-byte is followed by a 4-byte unsigned long which is the effective address. As an example, the following instruction stores register w0 into memory location 65536:

```
                          opcode   qr-byte    4 bytes
                         +-------+----------+----------+
                         |       |      |   |          |
    stow  w0 m65536      | stow  | mema | 0 | 00010000 |
                         |       |      |   |          |
                         +-------+----------+----------+
```

*   imm: immediate: q-operand is an immediate value

    The 3-bit ival field specifies the immediate value:

```
    001 (imm_1):  -1
    010 (imm0):    0
    011 (imm1):   +1
    100 (immv):   The immediate value follows the qr-byte
    101 (imm2):   2-byte immediate value for long operands
```

    When the immediate value is one of the special values -1, 0, or +1, then there is no need to follow the qr-byte with the value. Thus, the instruction is only 2 bytes long instead of 4 or more bytes. However, the actual operand represented by imm_1, imm0, or imm1 depends on the operand type specified by the opcode:

type	imm_1	imm0	imm1
byte:	ff	00	01
word:	ffff	0000	0001
long:	ffffffff	00000000	00000001
float:	bf800000	00000000	3f800000
double:	bff0000000000000	0000000000000000	3ff0000000000000

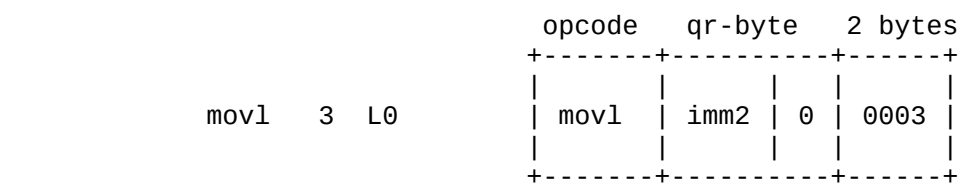
Similarly, in the immv case, the immediate value following the qr-byte depends on the operand type:

```
type    immediate operand format
-------------------------------------------------
byte:    2-byte integer; highorder byte always zero
word:    2-byte integer
long:    4-byte integer
float:   4-byte float
double:  8-byte double
```

Following are examples of immv, in which all types of the immediate value 3 are moved to registers. Note that the qr-bytes of all the instructions are identical, but the byte strings representing the immediate values are different, depending on the operand type indicated by the opcode:

```
                      opcode   qr-byte   2 bytes
                     +-------+----------+------+
                     |       |      |   |      |
movbw  3  w0         | movbw | immv | 0 | 0003 |
                     |       |      |   |      |
                     +-------+----------+------+

                      opcode   qr-byte   2 bytes
                     +-------+----------+------+
                     |       |      |   |      |
movw   3  w0         | movw  | immv | 0 | 0003 |
                     |       |      |   |      |
                     +-------+----------+------+

                      opcode   qr-byte    4 bytes
                     +-------+----------+----------+
                     |       |      |   |          |
movl   3  L0         | movl  | immv | 0 | 00000003 |
                     |       |      |   |          |
                     +-------+----------+----------+

                      opcode   qr-byte    4 bytes
                     +-------+----------+----------+
                     |       |      |   |          |
movf   3  f0         | movf  | immv | 0 | 40400000 |
                     |       |      |   |          |
                     +-------+----------+----------+

                      opcode   qr-byte    8 bytes
                     +-------+----------+------------------+
                     |       |      |   |                  |
movd   3  d0         | movd  | immv | 0 | 4008000000000000 |
                     |       |      |   |                  |
                     +-------+----------+------------------+
```

The imm2 option can only be used with instructions that operate on long data. It indicates that the long immediate value following the qr-byte consists of only 2 bytes. Conceptually these 2 bytes are sign-extended to a 4-byte immediate value. Here is an example of an instruction which moves the value 3 to long register L0:



The imm2 option reduces the instruction length from 6 bytes to 4 bytes for many long instructions which have an immediate value.

* regx: register extended addressing: q-operand specified by x-bytes

The regx option is not really an addressing mode. It is an escape code that indicates that one or two x-bytes follow the qr-byte.

The x-bytes specify one of these addressing modes for the q-operand:

```
i:     index
id1:   index + disp1
id3:   index + disp3
id4:   index + disp4

b:     base
bd1:   base + disp1
bd3:   base + disp3
bd4:   base + disp4

bi:    base + index
bid2:  base + index + disp2
bid4:  base + index + disp4
```

The first byte following the qr-byte is the x1-byte. Bits 4 and 5 of the x1-byte specify the major addressing mode:

Bits	Meaning	Specified by
0x	Index register only	x1-byte
10	Base register only	x1-byte
11	Both base and index registers	x1- and x2-bytes

The exact bit encoding of each addressing mode is shown in the following diagrams, where this notation is used:

breg	A 3-bit base register number (a long register)
ireg	A 3-bit index register number (a long register)
s	An 2-bit index scale factor 00 = 1 (Note: Shifting the index reg. 01 = 2 left by the number of bits in 10 = 4 the s field is the same as mul- 11 = 8 tiplying by the scale factor.)
<0-byte>	A byte consisting of all zero bits
<dispN>	An N-byte displacement

Index Only: x1-byte

	7	6	5	4	3	2	1	0		
i:	0	0	0	s		ireg			<0-byte>	
id1:	0	1	0	s		ireg			<disp1>	
id3:	1	0	0	s		ireg			<disp3>	
id4:	1	1	0	s		ireg			<0-byte>	<disp4>

Base Only: x1-byte

	7	6	5	4	3	2	1	0		
b:	0	0	1	0	0	breg			<0-byte>	
bd1:	0	1	1	0	0	breg			<disp1>	
bd3:	1	0	1	0	0	breg			<disp3>	
bd4:	1	1	1	0	0	breg			<0-byte>	<disp4>

Base and Index: x1-byte x2-byte

	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
bi:	0	0	1	1	0	breg			0	0	0	s		ireg			
bid2:	1	0	1	1	0	breg			0	0	0	s		ireg			<disp2>
bid4:	1	1	1	1	0	breg			0	0	0	s		ireg			<disp4>

Note that bits 6 and 7 of the x1-byte specify the displacement and/or 0-byte filler needed for each addressing mode. In a sense, bits 6 and 7 are a submode of the major mode specified by bits 4 and 5.

The way in which the effective address of the q-operand is determined for the addressing modes above can be represented by this formula, where B is the contents of the base register, I is the contents of the index register, S is the scale factor, and D is the value of the displacement:

```
B + S*I + sign-extend(D)
```

This formula works for all the addressing modes if we define B as zero when there is no base register, I as zero when there is no index register, and D as zero when there is no displacement. (Note: A displacement is sign-extended to a 4-byte value before being used in an address calculation.)

[A note on unused bit patterns in the qr-format: The five bits of the q-field can store 32 different bit patterns. At present all bit patterns are used except for one:

```
11110
```

The q-field never contains this bit pattern. Also, pattern 11101 (imm2) can only appear in instructions which operate on long data. Numerous bit patterns are not used in the two x-bytes. At some point the unused bit patterns should be listed so that the interpreter can test for them during error checking.]

Notation for qr Operands

In the description of a qr-format instruction, the following notation is used to show what types of operands the instruction takes:

q-field

bvw	byte value (word register)
bvl	byte value (long register)
wv	word value
lv	long value
fv	float value
dv	double value
baw	byte address (word register)
bal	byte address (long register)
wa	word address
la	long address
fa	float address
da	double address

r-field

wr	word register
lr	long register
fr	float register
dr	double register

The r-field always specifies a register. The q-field, on the other hand, specifies either a value or an address. When an address is required, it means that the q-operand CANNOT be an immediate value. However, the q-operand can be a register. In effect, when an address is required, it means that the operand must be something in which values can be stored, i.e., a memory location or a register, but not an immediate value.

Here are some examples:

```
movl    lv      lr
```

The "move long" instruction requires a long VALUE as a q-operand (source), and a long register as an r-operand (destination).

```
stol    lr      la
```

The "store long" instruction requires a long ADDRESS as a q-operand (destination), and a long register as an r-operand (source).

Decoding the q-field

Although the above list contains 12 specifiers for q-operands, the VMAX interpreter does not need 12 different functions for decoding a q-operand.

For example, the process of determining an address is exactly the same for all address specifiers, except when the "address" is a register. Also, except for immediate values, the process of determining a value consists of determining the address of the value, and then accessing the value at the address.

Thus, except for the reg and imm options of a q-operand, the type of operand (word, long, etc.) is not a factor in decoding the q-field. However, for reg and imm options the operand type is critical. For example, the bit pattern 000 stands for 4 different registers (w0, L0, f0, and d0) depending on the type of opcode. Also, the format of an immediate operand in an instruction depends on the opcode type.

The byte value and byte address specifiers (bvw, bvl, baw, and bal) require some discussion. The VMAX architecture is somewhat inconsistent when it comes to byte operands. It is possible to address individual bytes of memory and to move single bytes back and forth between memory and registers, but there are no byte registers. Thus, the loworder bytes of word and long registers are used as byte registers. This means that when a byte value or a byte address specifier is used, it is necessary to indicate whether the byte operand is in a word register or in a long register. Thus the suffixes "w" and "l" are appended to bv and ba, resulting in the specifiers bvw, bvl, baw, and bal.

It should be stressed that the "w" and "l" suffixes are only relevant for the reg q-operand option, not for any other options. For example, consider the description of the "move byte to loworder byte of word register" instruction:

```
movbw   bvw     wr
```

The q-operand is a byte value, and the r-operand is a word register. Thus,

```
movbw   [sp+2]  w3
```

moves the single byte stored at memory address sp+2 to the loworder byte of word register w3. The decoding of sp+2 does not depend on the fact that the q-operand is a byte or that the byte will be placed in a word register. However, in this instruction

```
movbw   w2      w3
```

the q-operand is a register, so the "w" suffix of bvw indicates that the q-register is a word register, not a long register.

When the q-operand specifier is bvw or bvl, and the imm option is used, the format of the immediate value is the same: A 2-byte integer with a zero highorder byte. The "w" and "l" suffixes do not effect the format of an immediate value. All byte immediate values have the same format. For example, the following two instructions are bitwise identical except for the opcode bytes:

```
movbw   18      w0
movbl   18      L0
```

[Note: The VMAX inconsistency regarding bytes imposes a bit of a restriction on register-to-register moves of bytes: The source and destination registers of the move must be of the same type. For example, if the destination register is a word register, then the source register must also be a word register, even though from a strictly logical point of view, there is no reason why it shouldn't be possible for the source register to be a long register.]

The qc Format

The qc-format instruction has exactly the same format as a qr-format instruction, except that the r-field is called the c-field, and it contains a condition code instead of a register number. Actually, the c-field only contains part of a condition code, namely, the loworder 3 bits of the code. The highorder bit of the code is determined by the opcode. There are only 4 instructions which use the qc-format, which is a special format for converting conditions into the values 0 and 1. This is best described by a discussion of two instructions:

```
        set0w   Store condition(0) in word
        set1w   Store condition(1) in word
```

The only difference between these two instructions is the way in which the condition code is created: For set0w, the condition code is the c-field with a zero bit appended at the beginning, and for set1w, the condition code is the c-field with a one bit appended at the beginning. In both cases, the result is a 4-bit condition code with a value in the range [0, 15]. See the description of the i-operand of the ij-format in a later section for all condition codes and what they mean.

The effect of set0w and set1w is simple: If the condition is TRUE, set the destination word to the value 1, otherwise set it to the value 0. The q-field describes the destination operand in exactly the same way as the q-field of a qr-format instruction. The q-field must describe an address (memory location or register); an immediate value is not allowed.

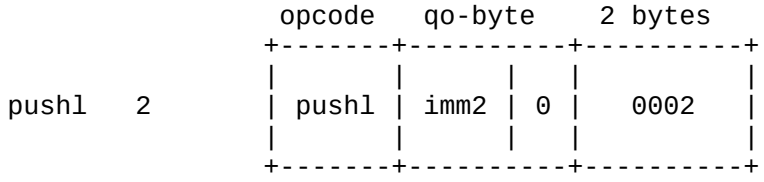
The other two instructions which have the qc-format are

```
        set0l   Store condition(0) in long
        set1l   Store condition(1) in long
```

The action of these instructions is exactly the same as set0w and set1w except that the type of the destination operand is long.

The qo Format

The qo-format instruction has exactly the same format as a qr-format instruction, except that the r-field is not used (it is always 000). The qo instructions are 1-operand instructions, most of which are push and pop instructions. Here is an example of a qo instruction which pushes long integer 2 onto the stack:



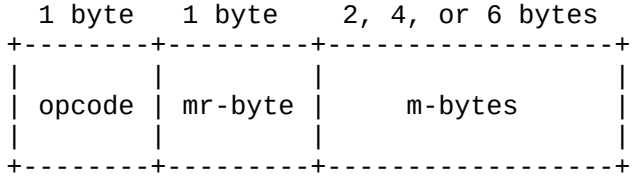
The same operand specifiers used for the q-field of qr instructions are used for the q-field of qo instructions. Some examples:

```
pushl   lv
popd    da
```

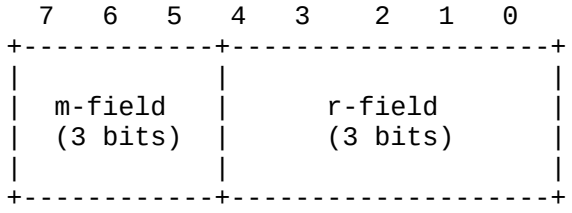
The mr Format

The mr-format is used by only two instructions, gmov (general move) and gsto (general store). These instructions allow 1, 2, 4, or 8 bytes to be moved from memory to any register (and vice versa), and from any register type to any other register type.

An mr-format instruction consists of an opcode, an mr-byte, and a sequence of m-bytes.



The mr-byte consists of two fields, the m-field and the r-field:



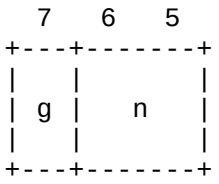
The m-field specifies the m-operand, a general operand (except that the m-operand cannot be an immediate value), and the r-field specifies the r-operand, which is always a register.

The sequence of m-bytes which may follow the mr-byte depends on the addressing mode specified in the m-field. All cases are described in detail in following sections.

The r-field contains a register number without regard to type. The 5-bits of this field contain a value from 0 through 31, which indicates a register as shown in the diagram in the General Registers section near the beginning of this document. Thus, 0 indicates w0, 9 indicates L1, and 31 indicates d7. Another way of looking at the r-field is to consider bits 3 and 4 as type bits, and bits 0, 1, and 2 as a register number within a type. The types are these:

00	word
01	long
10	float
11	double

The m-field consists of two subfields:

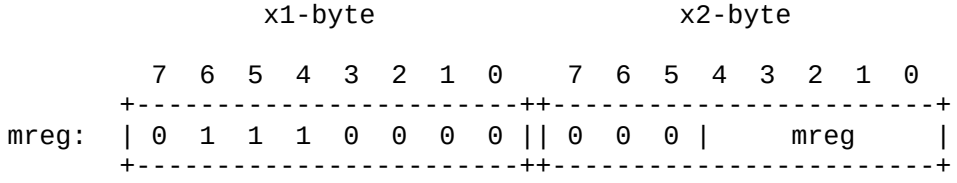


The n-field specifies the number of bytes to move, using these encodings:

00	1
01	2
10	4
11	8

The g-field indicates an addressing mode. If g = 0, then the mr-byte is followed by a 4-byte memory address, which is the address to move to or from (this is equivalent to the mema addressing mode of the qr-format).

If g = 1, then the mr-byte is followed by one or two x-bytes, just as for the qr-format. Thus, all based, indexed, and based + indexed addressing modes are available for mr-format instructions. In addition, for the mr-format, the x-bytes can specify the following addressing mode (which is not available in the x-bytes of the qr-mode):



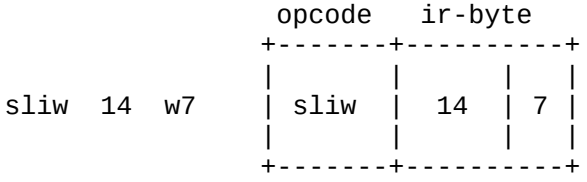
The mreg is a 5-bit field specifying a register of any type, using the register numbers shown in the General Registers section near the beginning of this document. This addressing mode provides for register-to-register operations in the mr-format.

The operand specifiers for mr instructions are these:

gv	general value
ga	general address
gr	general register

The ir Format

The ir-format instruction has exactly the same format as a qr-format instruction, except that the q-field is a 5-bit integer instead of a general q-operand specifier. The r-field specifies a register, as in the qr-format. The integer in the i-field can be used for various purposes, depending on the opcode, but the most common use of this field is for a shift count in shift-immediate instructions. As an example, consider the following instruction which shifts word register w7 left by 14 bits:



All ir instructions are 2-operand instructions in which one operand is a 5-bit integer and the other operand is a register.

The same operand specifiers used for the r-field of qr instructions are used for the r-field of ir instructions. The specifiers used for the i-field depend on the opcode; at present there is only one specifier:

i-field	
sc	shift count

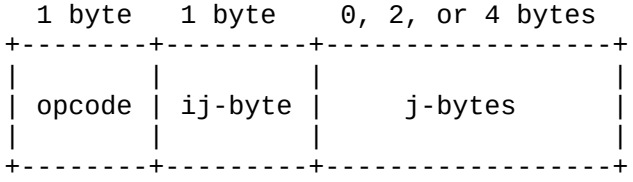
An example of a description of a ir instruction:

```
sliw    sc      wr
```

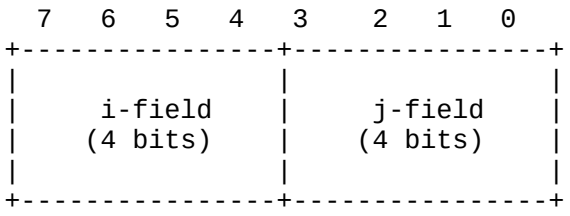
The ij Format

The ij-format is used for conditional and unconditional jumps and calls. It specifies two operands, a condition and a memory address of an instruction. Any address in the VMAX address space can be specified with the ij-format.

An ij-format instruction consists of an opcode, an ij-byte, and, for some addressing modes, a sequence of j-bytes.



The ij-byte consists of two fields, the i-field and the j-field:



The i-field specifies a condition, and the j-field is an addressing mode for determining the target address of the instruction. The sequence of j-bytes which may follow the ij-byte depends on the addressing mode specified in the j-field. The i-field, j-field, and j-bytes are described in detail in the following sections.

The i-operand

The i-operand specifies a condition based on the bits in the flags register:

i-field contents	decimal value	sym-bol	condition	expression
0000	0	UNC	unconditional	TRUE
0001	1	LEU	less than or equal unsigned	LUF=1 || EF=1
0010	2	LU	less than unsigned	LUF=1
0011	3	L	less than signed	LF=1
0100	4	LE	less than or equal signed	LF=1 || EF=1
0101	5	E	equal	EF=1
0110	6	NE	not equal	EF=0
0111	7	GE	greater than or equal signed	GF=1 || EF=1
1000	8	G	greater than signed	GF=1
1001	9	GU	greater than unsigned	GUF=1
1010	10	GEU	greater than or equal unsigned	GUF=1 || EF=1
1011	11		RESERVED	
1100	12		RESERVED	
1101	13		RESERVED	
1110	14		RESERVED	
1111	15		RESERVED	

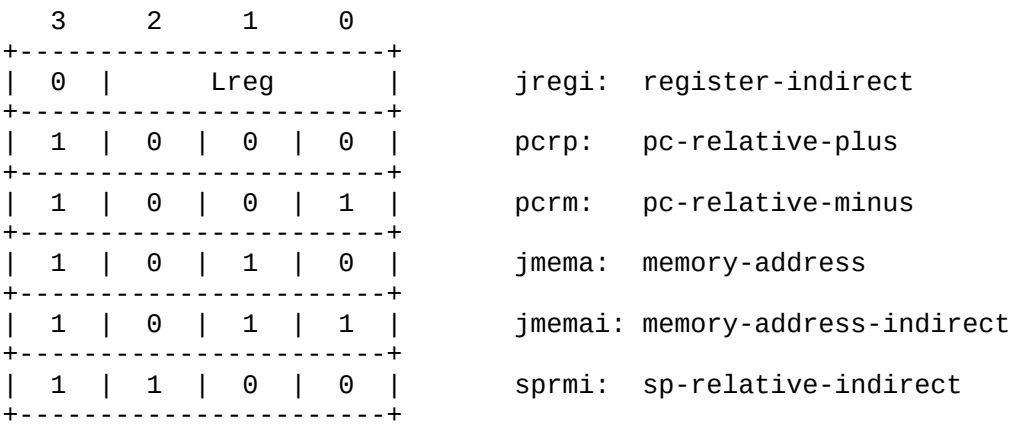
When an ij instruction is executed, the result depends on the expression in the rightmost column above. If the expression is FALSE, then the ij instruction is equivalent to a NOP, i.e., control passes to the following instruction.

If the expression is TRUE, then pc is changed to the memory address specified by the j-operand, i.e., a jump or a call is performed.

Note that at present only 11 bit patterns are defined for the i-field. No bit patterns other than those shown above may appear in the i-field.

The j-operand

The j-operand is specified by one of these encodings in the j-field:



Each of the above kinds of j-operands is described in the following pages.

* jregi: register-indirect: j-operand EA is long register contents

  The 3-bit Lreg field contains an integer which specifies one of the eight long registers. The contents of the specified long register is the effective address to jump to or to call.

* pcrp: pc-relative-plus: j-operand EA is pc contents plus an offset

  The ij-byte is followed by a 2-byte unsigned offset which is scaled by a factor of 2. If the i-field condition is satisfied, then pc is changed like this:

  pc = pc + 2*offset

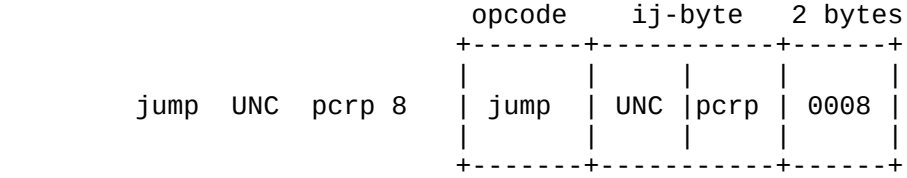
  The offset is scaled so that the range of addresses reachable by this addressing mode is doubled. Since all instructions start at word boundaries, the loworder bit of an instruction address is always 0, so there is no need to carry it. Thus, this addressing mode can be used to reach any instruction located at an address in the range pc + [0, 2^17-2 = 131,070].

  The C compiler generates jump instructions using the pcrp and pcrm modes, so the maximum safe size for a C function is 128KB. This should not be a restriction. Any C function which results in code of such a size is probably a logical mess which should be rewrit-

ten anyway. (Note: One of the advantages of using pc-relative addresses for jump instructions is that such instructions do not need to be relocated.)

At the point when pc is incremented, it addresses the byte following the current instruction. Thus, if the offset is zero, the effect is the same as a NOP: Control passes to the next instruction just as if the jump condition were not satisfied.

Here is an example of an unconditional jump to pc+8:



* pcrm: pc-relative-minus: j-operand EA is pc contents minus an offset

This mode is exactly the same as pcrp except that the 2-byte unsigned offset is subtracted from pc:

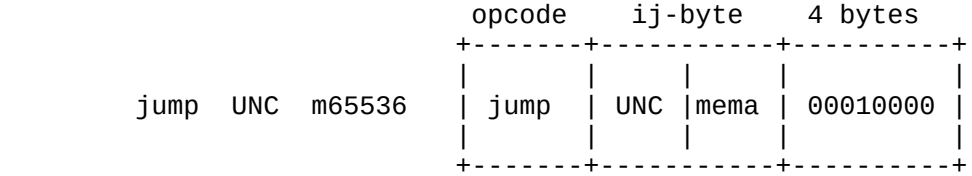
pc = pc - 2*offset

This addressing mode can be used to reach any instruction located at an address in the range pc - [0, 2^17-2 = 131,070].

* jmema: memory-address: j-operand EA is memory address in instruction

The ij-byte is followed by a 4-byte unsigned long which is the effective address. This address is placed in pc (without scaling) to transfer control. Note that any address in the VMAX address space can be reached with the jmema addressing mode.

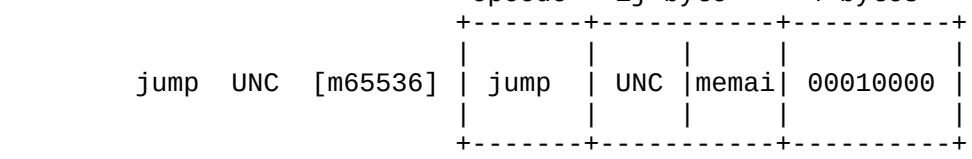
As an example, the following instruction jumps to address 65536:



* jmemai: memory-address-indirect: j-operand EA is memory address in instruction, indirect

The ij-byte is followed by a 4-byte unsigned long which is the address of a memory location where the effective address is located. The effective address is moved to pc to transfer control.

As an example, if 00000004 is stored at memory address 65536, then the following instruction jumps to memory address 4:



* sprmi: sp-relative-indirect: j-operand EA is in the stack

The ij-byte is followed by a 2-byte unsigned offset which is scaled by a factor of 2. If the i-field condition is satisfied, then pc is changed like this:

```
pc = [sp - 2 * offset]
```

In other words, the long value located on the stack at address sp-2*offset is the address to jump to. This mode is similar to pcrm in that the ij-byte is followed by a 2-byte scaled offset. However, this offset is relative to sp rather than to pc, and a level of indirectness is involved.

The sprmi mode is a bit ad hoc, but it turns out to be quite useful in the epilogue of a C function when the function pops its own arguments off the stack, and there are more than 510 bytes of arguments (which means that leave, leaveres, and ret cannot be used to pop the arguments and return).

[Note: The bit patterns 1101, 1110, and 1111 are currently not used in the j-field. Thus, up to 3 more addressing modes for jump and call instructions could be defined. Also, it is not clear if the jmemai mode is really useful for code generated by a compiler. As experience is gained with GCC and how it deals with jumps and calls, it may turn out that the the j-field addressing modes should be reconsidered.]

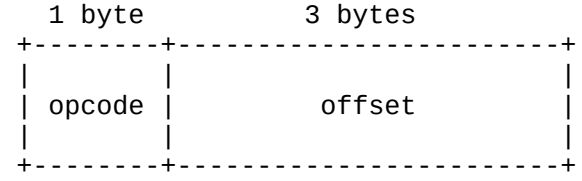
Notation for ij Operands

In the description of an ij-format instruction, the notation used to show what types of operands the instruction takes is very simple:

```
i-field
  cc            condition code


j-field
  ma            memory address
```

The a3 Format

The a3-format is used for unconditional jumps and calls. These instructions consist of an opcode and a 3-byte operand:



The operand is a 3-byte unsigned offset which is scaled by a factor of 2. This offset is used to change pc like this:

```
pc = pc +- 2*offset
```

Whether the offset is added or subtracted depends on the opcode (the following instructions are the only ones which have the a3 format):

opcode	instruction	change to pc
jumpf	jump forward	Add offset to pc
callf	call forward	Add offset to pc
jumpb	jump backward	Subtract offset from pc
callb	call backward	Subtract offset from pc

A jump or call using the a3-format can reach a memory address as far away as 2^25-2 = 33,554,430.

The C compiler generates call instructions using the a3-format, so the maximum safe size for a compiled C program is 32MB. This should be big enough for the near future. (Note: If a program gets too large, the compiler can be told to generate ij-format calls with the mema option. This will increase the size of every call from a 4-byte instruction to a 6-byte instruction, but then there are no limits at all except for the 4GB address space.)

One of the advantages of using the a3 format for calls is that no relocation is needed for these instructions.

At the point when pc is incremented or decremented, it addresses the byte following the current instruction.

The specifier used in descriptions of a3 instructions is

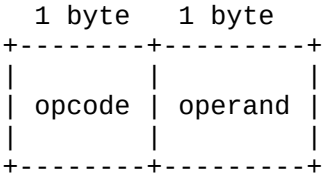
ma3	3-byte memory address

An example of a description of an a3 instruction:

```
jumpf   ma3
```

The b1 Format

The b1-format is used for instructions which require only a single byte as an operand:



The 1-byte operand is used for various purposes, depending on the opcode. See the descriptions of the enter and ret instructions for examples.

The specifiers used in descriptions of b1 instructions depend on the opcode; at present there is only one specifier:

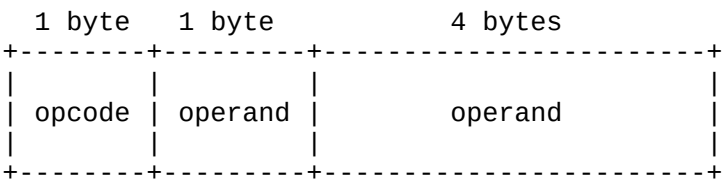
stkc	stack change

An example of a description of a b1 instruction:

```
                ret     stkc
```

The b14 Format

The b14-format is the same as the b1 format with an additional 4-byte operand:



The two operands are used for various purposes, depending on the opcode. See the descriptions of the entersav and leaveres instructions for examples.

The specifiers used in descriptions of b14 instructions depend on the op-code; at present only these specifiers are used:

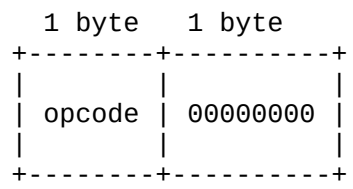
```
            stkc        stack change
            bmsk        bit mask
```

An example of a description of a b14 instruction:

```
            entersav  stkc  bmsk
```

The n0 Format

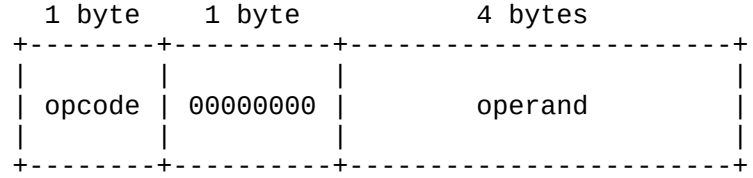
The n0-format is used for instructions which have no operands at all:



Since an instruction must be an even number of bytes long, the opcode of an n0-format instruction is followed by a single byte which is always zero. Examples of n0-format instructions are nop and halt.

The n04 Format

The n04-format is the same as the n0 format with an additional 4-byte operand:



The operand is used for various purposes, depending on the opcode. See the descriptions of the pushregs and popregs instructions for examples.

The specifiers used in descriptions of n04 instructions depend on the op-code; at present only one specifiers is used:

bmsk	bit mask

An example of a description of a n04 instruction:

popregs	bmsk

Data Movement Instructions

Move Instructions

[Terminology: For VMAX instructions, "move" means move TO a register, and "store" means store FROM a register (usually into memory).]

The following instructions move an operand (memory, immediate, or register) to a register:

movw	Move word
movl	Move long
movf	Move float
movd	Move double
movbw	Move byte to loworder byte of word
movbl	Move byte to loworder byte of long
movwl	Move word to loworder word of long
gmov	General move

For movw, movl, movf, and movd, the source and destination operands are of the same type. For movbw and movbl, the source operand is a byte, and the destination operand is a word register (movbw) or a long register (movbl). These move instructions change ONLY the loworder byte of the destination register; the other bytes in the destination register are left unchanged. The movwl instruction moves a word to a long register, changing ONLY the loworder word of the long register.

The general move instruction, gmov, is provided to allow data of any type to be moved to any type of register, using multiple registers if needed. This instruction allows 1, 2, 4, or 8 bytes to be moved from any register or from memory to any register. For example, a double register can be moved to 4 word registers with gmov, or 8 bytes of memory can be moved to 2 long registers. The source operand of gmov cannot be an immediate value.

[Note: GCC uses all the move instructions defined above. However, at the moment GCC does not utilize the fact that movbw, movbl, and movwl leave part of the destination register unchanged. It seems that GCC has the capability to use this fact, but it is not yet clear how to tell GCC about this. The gmov instruction was defined solely because GCC requires it. However, it turns out that it is probably a convenient and useful instruction to have anyway.]

Store Instructions

[Terminology: For VMAX instructions, "move" means move TO a register, and "store" means store FROM a register (usually into memory).]

The following instructions store a register into memory:

stow	Store word
stol	Store long
stof	Store float
stod	Store double
stowb	Store loworder byte of word into byte
stolb	Store loworder byte of long into byte
stolw	Store loworder word of long into word
gsto	General store

For stow, stol, stof, and stod, the source and destination operands are of the same type. For stowb and stolb, the source operand is the loworder byte of a word register (stowb) or a long register (stolb), and the destination is a single byte. These instructions change ONLY one byte. The stolw instruction stores the loworder word of a long register into a word.

The general store instruction, gsto, is provided to allow any type of data to be stored from any type of register into memory. This instruction allows 1, 2, 4, or 8 bytes to be stored from any register (or group of contiguous registers) into memory. For example, 2 word registers can be stored into 4 bytes of memory with gsto, or 2 long registers can be stored into 8 bytes of memory.

[Note: GCC uses all the store instructions defined above. The gsto instruction was defined solely because GCC requires it. However, it turns out that it is probably a convenient and useful instruction to have anyway.]

Note that sto instructions can be used to move from register to register. However, this is usually done with a mov instruction rather than with a sto instruction. [Perhaps we should not allow a sto instruction to be used for register-to-register moves?]

Load Address Instruction

There is one instruction for loading an address into a register:

leal	Load effective address

This instruction computes the effective address of its operand and places the effective address in a long register. It is also possible to use leal to perform calculations on the contents of long registers. For example, if L0 is referenced as a base register and L1 is referenced as an index register, leal can be used to add the contents of L0 and L1 and place the result in a third long register, say L2.

[Note: GCC uses the leal instruction.]

Flags Instruction

The following instructions load and store the flags register:

movflags	Move word to flags register
stoflags	Store flags register into word
set0w	Store condition(0) into word
set1w	Store condition(1) into word
set0l	Store condition(0) into long
set1l	Store condition(1) into long

The movflags instruction moves a 16-bit value into the flags register, changing all the flags. The stoflags instruction stores the flags register into a word. Thus, individual flag bits or groups of bits can be changed by moving the flags register into a word register with stoflags, changing bits of the word register, and then moving the word register to the flags register with movflags.

The set instructions are used to store a 0 or a 1 into a word or a long, depending on the condition codes.

[Note: GCC uses the set instructions, but it does not use movflags and stoflags.]

Arithmetic Instructions

Add Instructions

The following instructions perform addition operations, producing a sum from two addends:

addw	Add word
addl	Add long
addf	Add float
addd	Add double
addcl	Add long with carry
adduwl	Add unsigned word to long
addswl	Add signed word to long

For a given add instruction, the addends and sum are all of the same type except for adduwl and addswl. For these two instructions the addends are words, but the sum is long.

Note that addw can be used for both unsigned addition and signed addition. The same is true of addl.

[Note: GCC uses all the add instructions defined above except for addcl, adduwl, and addswl. These instructions can probably be used in a VMAX C program via the asm feature which allows assembly language to be embedded in C source code. Combining this with the in-line function capability of GCC should make these instructions usable in a fairly reasonably way.]

Subtract Instructions

The following instructions perform subtraction operations, producing a difference from two operands:

subw	Subtract word
subl	Subtract long
subf	Subtract float
subd	Subtract double
subcl	Subtract long with carry
subuwl	Subtract unsigned word from long
subswl	Subtract signed word from long
negw	Negate word
negl	Negate long
negf	Negate float
negd	Negate double

For a given subtraction instruction, the operands and difference are all of the same type except for subuwl and subswl. For these two instructions the operands are words, but the difference is long.

Note that subw can be used for both unsigned subtraction and signed subtraction. The same is true of subl.

The negate instructions perform arithmetic negation operations on their operands. (The negate instructions are unary in one sense, but binary in another: The negation of the source operand is placed in the destination register. Thus, both a negation and a move are performed in the general case.)

[Note: GCC uses all the subtraction instructions defined above except for subcl, subuwl, and subswl. These instructions can probably be used via asm and in-line functions, as discussed in the above section on add instructions.]

Multiply Instructions

The following instructions perform multiplication operations generating a product from a multiplier and a multiplicand:

muluw	Multiply unsigned word
mulsw	Multiply signed word
mulul	Multiply unsigned long
mulsl	Multiply signed long
mulf	Multiply float
muld	Multiply double
muluwl	Multiply unsigned words yielding long
mulswl	Multiply signed words yielding long

For a given multiply instruction, the multiplier, multiplicand, and product are all of the same type except for muluwl and mulswl. For these two instructions the multiplier and multiplicand are words, but the product is long.

[Note: GCC uses all the multiply instructions defined above.]

Divide Instructions

The following instructions perform division operations, producing a quotient, a remainder, or both:

divuw	Divide unsigned word
divsw	Divide signed word
divul	Divide unsigned long
divsl	Divide signed long
divf	Divide float
divd	Divide double
remuw	Remainder unsigned word
remsw	Remainder signed word
remul	Remainder unsigned long
remsl	Remainder signed long
divruw	Divide with remainder unsigned word
divrsw	Divide with remainder signed word
divrul	Divide with remainder unsigned long
divrsl	Divide with remainder signed long
divrulw	Divide with remainder unsigned long by word
divrslw	Divide with remainder signed long by word

The first group of instructions (div) produce only a quotient; no remainder is calculated. The second group of instructions (rem) produce only a remainder; no quotient is calculated. The third group of instructions (divr) produce both a quotient and a remainder. (Actually, divr instructions produce both a quotient and a remainder ONLY when the divisor is in a register. If the divisor is not in a register, then no remainder is produced.)

This table shows divide instructions grouped by operand type to make clear which operations are available for which data types:

operand types	divide	remainder	both
unsigned word	divuw	remuw	divruw
signed word	divsw	remsw	divrsw
unsigned long	divul	remul	divrul
signed long	divsl	remsl	divrsl
unsigned long/word			divrulw
signed long/word			divrslw
float	divf		
double	divd		

For a given divide instruction, the divisor, dividend, quotient, and remainder are all of the same type except for divrulw and divrslw. For

these two instructions the divisor, quotient, and remainder are words, but the dividend is long.

[Note: GCC uses all the divide instructions defined above except for divrulw and divrslw, the instructions which divide a long by a word and produce word results. GCC can use an instruction which divides an 8-byte integer by a long, but it does not seem useful to have such an instruction for VMAX, since VMAX does not support integers longer than 4 bytes. The divrulw and divrslw instructions can probably be used via asm and in-line functions, as discussed in the above section on add instructions.]

Since integer division of signed quantities can be defined in more than one way, it is important to make clear just what VMAX integer division instructions do. We use the following notation:

```
d - divisor (denominator)
n - dividend (numerator)
q - quotient
r - remainder
```

Given inputs d and n, the VMAX divide instructions produce the unique q and r which satisfy these relationships:

```
Unsigned:       n = d * q + r
                0 <= r < q

Signed:         n = d * q + r
                sgn(r) = sgn(n)
                0 <= |r| < |q|
```

Note that in signed division, if n is negative and r != 0, then r is negative. Thus, the remainder produced by a VMAX divide instruction is NOT n mod d in the mathematical sense (because n mod d is usually defined to be a nonnegative value). However, the mathematical mod function is rather easily obtained from the remainder:

```
if (r < 0)  r += abs(d);
```

[Note: If we later want to add some VMAX instructions to compute the mathematical mod, it might be a good idea to look at the Intel 80960MC. This machine has separate instructions for remainder (as it is defined for VMAX) and mod. The mod instructions produce a remainder with the same sign as the divisor (instead of with the same sign as the dividend).]

This section concludes with detailed definitions and examples of unsigned and signed division:

Unsigned integer division:

Input:

There are two unsigned inputs:

n is the dividend (numerator)
d is the divisor (denominator)

Output:

If d = 0 then an exception is generated. Otherwise, there are two outputs:

q is the quotient
r is the remainder

The outputs are the unique unsigned integers which satisfy the following:

n = d * q + r
0 <= r < q

Notes:

If n, d, q, and r are all the same type (i.e., consist of the same number of bits), then unsigned division can NEVER result in overflow.

Examples:

n / d	q	r	d * q + r = n
0 / 4	0	0	4 * 0 + 0 = 0
1 / 4	0	1	4 * 0 + 1 = 1
2 / 4	0	2	4 * 0 + 2 = 2
3 / 4	0	3	4 * 0 + 3 = 3
4 / 4	1	0	4 * 1 + 0 = 4
5 / 4	1	1	4 * 1 + 1 = 5
6 / 4	1	2	4 * 1 + 2 = 6
7 / 4	1	3	4 * 1 + 3 = 7
8 / 4	2	0	4 * 2 + 0 = 8

Signed integer division:

Input:

There are two signed inputs:

n is the dividend (numerator)
d is the divisor (denominator)

Output:

If d = 0 then an exception is generated. Otherwise, there are two outputs:

q is the quotient
r is the remainder

The outputs are the unique signed integers which satisfy the following:

n = d * q + r
sgn(r) = sgn(n)
0 <= |r| < |q|

Notes:

If n and d are both nonnegative, then signed division produces exactly the same outputs as unsigned division.

If n, d, q, and r are all the same type (i.e., consist of the same number of bits), then signed division can occur in ONLY ONE case: n = smallest possible negative number, and q = -1. In this case, the quotient is the largest positive number + 1, which is too big to fit. For example, if the operands are 16 bits long, then -32768/-1 = +32768, and +32767 is the largest positive number that will fit in a 16-bit signed quantity.

Examples:

n / d	q	r	d * q + r = n
0 / -4	0	0	-4 * 0 + 0 = 0
1 / -4	0	1	-4 * 0 + 1 = 1
2 / -4	0	2	-4 * 0 + 2 = 2
3 / -4	0	3	-4 * 0 + 3 = 3
4 / -4	-1	0	-4 * -1 + 0 = 4
5 / -4	-1	1	-4 * -1 + 1 = 5
6 / -4	-1	2	-4 * -1 + 2 = 6
7 / -4	-1	3	-4 * -1 + 3 = 7
8 / -4	-2	0	-4 * -2 + 0 = 8

```
-1 /  4      0  -1       4 *  0 + -1 = -1
-2 /  4      0  -2       4 *  0 + -2 = -2
-3 /  4      0  -3       4 *  0 + -3 = -3
-4 /  4     -1   0       4 * -1 +  0 = -4
-5 /  4     -1  -1       4 * -1 + -1 = -5
-6 /  4     -1  -2       4 * -1 + -2 = -6
-7 /  4     -1  -3       4 * -1 + -3 = -7
-8 /  4     -2   0       4 * -2 +  0 = -8

-1 / -4      0  -1      -4 *  0 + -1 = -1
-2 / -4      0  -2      -4 *  0 + -2 = -2
-3 / -4      0  -3      -4 *  0 + -3 = -3
-4 / -4      1   0      -4 *  1 +  0 = -4
-5 / -4      1  -1      -4 *  1 + -1 = -5
-6 / -4      1  -2      -4 *  1 + -2 = -6
-7 / -4      1  -3      -4 *  1 + -3 = -7
-8 / -4      2   0      -4 *  2 +  0 = -8
```

Other Arithmetic Instructions

The following instructions take absolute values and square roots:

```
        absw            Absolute value of word
        absl            Absolute value of long
        absf            Absolute value of float
        absd            Absolute value of double

        sqrtf           Square root of float
        sqrtd           Square root of double
```

[Note: GCC uses all the instructions defined above.]

Shift Instructions

The following instructions perform shift and rotate operations:

General shift instructions:

slw	Shift left word
sll	Shift left long
srlw	Shift right logical word
srll	Shift right logical long
sraw	Shift right arithmetic word
sral	Shift right arithmetic long

Shift count is an immediate operand:

sliw	Shift left immediate word
slil	Shift left immediate long
srliw	Shift right logical immediate word
srlil	Shift right logical immediate long
sraiw	Shift right arithmetic immediate word
srail	Shift right arithmetic immediate long

General rotate instructions:

rlw	Rotate left word
rll	Rotate left long
rrw	Rotate right word
rrl	Rotate right long

Rotate count is an immediate operand:

rliw	Rotate left immediate word
rlil	Rotate left immediate long
rriw	Rotate right immediate word
rril	Rotate right immediate long

Only word registers and long registers can be shifted. There are three different kinds of shifts:

left	shift to the left, fill with zero bit
right logical	shift to the right, fill with zero bit
right arithmetic	shift to the right, fill with sign bit

Each shift instruction comes in two forms: The shift count is a general

operand, or the shift count is an immediate value. The latter type of shift instructions are shorter than the general shift instructions.

Rotate instructions are very similar to shift instructions, but bits shifted out of one end of a register are shifted into the other end of the register.

Logical Instructions

The following instructions perform logical operations:

andw	And word
andl	And long
orw	Or word
orl	Or long
xorw	Exclusive or word
xorl	Exclusive or long
notw	Not word
notl	Not long

The and, or, and xor instructions perform the usual bitwise logical operations on their two operands. The not instructions perform logical negation operations on their operands. (The not instructions are unary in one sense, but binary in another: The negation of the source operand is placed in the destination register. Thus, both a negation and a move are performed, in the general case.)

[Note: GCC uses all the logical instructions defined above.]

Convert Instructions

The following instructions convert from one type of data to another type of data:

Byte -> Word

cvtbzw	Convert byte zero-extended to word
cvtbsw	Convert byte sign-extended to word

Byte -> Long

cvtbzl	Convert byte zero-extended to long
cvtbsl	Convert byte sign-extended to long

Word -> Long

cvtwzl	Convert word zero-extended to long
cvtwsl	Convert word sign-extended to long

Long -> Float

cvtulf	Convert unsigned long to float
cvtslf	Convert signed long to float

Long -> Double

cvtuld	Convert unsigned long to double
cvtsld	Convert signed long to double

Float -> Long

cvttful	Convert truncated float to unsigned long
cvttfsl	Convert truncated float to signed long

Double -> Long

cvttdul	Convert truncated double to unsigned long
cvttdsl	Convert truncated double to signed long

Float <-> Double

cvtdf	Convert double to float
cvtfd	Convert float to double

It is important to note that cvttful, cvttfsl, cvttdul, and cvttdsl convert from floating to integer by TRUNCATING TOWARD ZERO. This is exactly what is wanted for C, but it does not conform to the IEEE 754 standard for floating point operations, which defines a variety of rounding methods, usually controlled by rounding control bits of a floating point control word. Later we may want to define other floating -> integer conversion instructions which take rounding control bits into account. This is done on the Intel 80960MC: There are floating -> integer conversion instructions which ignore the rounding control bits, and there are floating -> integer conversion instructions which use the rounding control bits.

Later it may also prove useful to provide an instruction which truncates a floating value to a FLOATING integer. However, such an instruction does not seem to be required by GCC, so for now we forget about it.

Although VMAX provides numerous conversion instructions, there is not one instruction for every type of conversion required by C. For example, there is no instruction to convert a word to a floating value. The table on the following page shows how all possible C conversions can be implemented with VMAX instructions. A single instruction suffices except when integers other than longs are converted to floating, and vice versa. In these cases, two VMAX instructions are needed. (Of course, if it proves useful, we can later define other conversion instructions, e.g., word to double.)

Some of the conversions shown in the following table may seem somewhat peculiar. For example, if the value 0xff stored in a signed char is converted to to an unsigned short, the result is 0xffff, i.e., the source value is sign-extended to the size of the destination value. Since the destination is an unsigned entity, it might seem that the converted value should be 0x00ff. However, according to ANSI C, the correct thing to do is to sign-extend. As best as could be determined, all conversions shown in the table conform to ANSI C (and Microsoft C 5.1 and 6.0).

[Note: GCC uses all the conversion instructions defined above.]

VMAX Instructions for C Type Conversions

from \ to	UCHAR	SCHAR	UWORD	SWORD	ULONG	SLONG	FLOAT	DOUBLE
UCHAR	X	SBP	cvtbzw	cvtbzw	cvtbzl	cvtbzl	cvtbzl cvtulf	cvtbzl cvtuld
SCHAR	SBP	X	cvtbsw	cvtbsw	cvtbsl	cvtbsl	cvtbsl cvtslf	cvtbsl cvtsld
UWORD	stowb	stowb	X	SBP	cvtwzl	cvtwzl	cvtwzl cvtulf	cvtwzl cvtuld
SWORD	stowb	stowb	SBP	X	cvtwsl	cvtwsl	cvtwsl cvtslf	cvtwsl cvtsld
ULONG	stolb	stolb	stolw	stolw	X	SBP	cvtulf	cvtuld
SLONG	stolb	stolb	stolw	stolw	SBP	X	cvtslf	cvtsld
FLOAT	cvttfsl stolb	cvttfsl stolb	cvttfsl stolw	cvttfsl stolw	cvttful	cvttfsl	X	cvtfd
DOUBLE	cvttdsl stolb	cvttdsl stolb	cvttdsl stolw	cvttdsl stolw	cvttdul	cvttdsl	cvtdf	X

SBP: Same Bit Pattern X: No conversion needed

	VMAX type	C type
UCHAR	signed byte	signed char
SCHAR	unsigned byte	unsigned char
UWORD	unsigned word	unsigned short
SWORD	signed word	signed short
ULONG	unsigned long	unsigned long
SLONG	signed long	signed long
FLOAT	float	float
DOUBLE	double	float

Compare Instructions

The following instructions compare two operands:

cmpw	Compare word
cmpl	Compare long
cmpf	Compare float
cmpd	Compare double
cmpwb	Compare loworder byte of word to byte
cmplb	Compare loworder byte of long to byte

For all the compare instructions, the two operands to be compared are of the same type. However, since VMAX has no byte registers, at least one operand of a byte compare must be in either a word register or a long register. The cmpwb and cmplb instructions handle these two cases.

[Note: GCC uses all the compare instructions defined above.]

The floating point compare instructions cmpf and cmpd set a single flag bit and clear the others. Only these three patterns are possible:

LU	L	EQ	G	GU	Examples
0	0	0	1	0	compare 3.0 and -1.0
0	0	1	0	0	compare 3.0 and 3.0
0	1	0	0	0	compare 1.0 and 3.0

The fixed point compare instructions cmplb, cmpwb, cmpw, and cmpl set one or two flag bits and clear the others. In concept, fixed point compare instructions involve three steps:

1. Clear all the flag bits to zero.

2. Compare the operands as unsigned values and set a single bit.

3. Compare the operands as signed values and set a single bit.

If the operands are equal (bit-for-bit the same), then steps 2 and 3 above both set the EQ bit. Thus, only one bit of the five is set. However, if the operands are not equal, then two of the five bits are set. Step 2 sets one of LTU or GTU, and step 3 sets one of LT or GT. Thus a total of five patterns are possible:

LU	L	EQ	G	GU	Examples (bytes)	Unsigned	Signed
0	0	0	1	1	compare 0f and 01	15 and 1	15 and 1
0	1	0	0	1	compare ff and 0f	255 and 15	-1 and 15
1	0	0	1	0	compare 0f and ff	15 and 255	15 and -1
0	0	1	0	0	compare ff and ff	255 and 255	-1 and -1
1	1	0	0	0	compare 01 and 0f	1 and 15	1 and 15

When the patterns resulting from floating point compares are combined with those resulting from fixed point compares, a total of seven distinct patterns are possible:

LU	L	EQ	G	GU	Decimal value
0	0	0	1	0	2
0	0	0	1	1	3
0	0	1	0	0	4
0	1	0	0	0	8
0	1	0	0	1	9
1	0	0	1	0	18
1	1	0	0	0	24

Jump Instructions

The following instructions jump and call functions, both conditionally and unconditionally:

call	Call
callb	Call backward
callf	Call forward
jump	Jump
jumpb	Jump backward
jumpf	Jump forward
ret	Return from call
leave	Leave function
leaveres	Leave function and restore registers

Except for the fact that call instructions push the program counter onto the stack, call and jump instructions behave in the same way, so in the following only jump instructions are discussed.

The jump instruction either jumps or doesn't based on the condition bits in the instruction and the current settings of the condition codes. The following conditions can be tested for:

UNC	Unconditional
LU	Less than (unsigned)
LEU	Less than or equal (unsigned)
L	Less than
LE	Less than or equal
EQ	Equal
NE	Not equal
GE	Greater than or equal
G	Greater than
GEU	Greater than or equal (unsigned)
GU	Greater than (unsigned)

A jump instruction also specifies one of several jump adressing modes, as described in an earlier section. One of the modes specifies a 4-byte absolute address, so any instruction in a 4-gigabyte address space can be reached with a jump instruction.

The jump forward and jump backward instructions are unconditional. These instructions jump relative to the program counter with a range of 32MB.

The return instruction is used to exit a function and return to the caller, possible clearing arguments from the stack. The leave instruction is similar, but it does even more stack cleanup. The leaveres instruction is the same as the leave instruction, but it restores registers saved in the stack.

[Note: GCC uses all the instructions defined above.]

Stack Instructions

The following instructions operate on the stack:

pushw	Push word
pushl	Push long
pushf	Push float
pushd	Push double
pushregs	Push multiple registers
popw	Pop word
popl	Pop long
popf	Pop float
popd	Pop double
popregs	Pop multiple registers
enter	Enter function
entersav	Enter function and save registers

The push and pop instructions allow data of all types to be pushed onto the stack and popped off the stack. The pushregs instruction allows any subset of the 32 major VMAX registers to be pushed with one instruction; popregs is the inverse function.

The enter and entersav instructions are used to handle common operations required upon function entry. The entersav instruction performs the same tasks as enter, and in addition it saves registers in the stack.

[Note: GCC uses all the instructions defined above.]

Miscellaneous Instructions

The following instructions do not fit in any other categories:

halt	Halt the VMAX machine
nop	No operation

[Note: Surprisingly enough, GCC uses nop: In certain instances when GCC is not optimizing, a nop is generated as a convenient place to attach a label that might be helpful when running a debugger on the compiled program. GCC does not use the halt instruction.]

Notation Sumary

Most of the notation used in the following instruction descriptions has already been defined earlier in this document, but for ease of reference, all the notation is summarized here in one place.

qr instructions: q-operand r-operand

q-operand is either a value or an address

bvw	byte value (word register)
bvl	byte value (long register)
wv	word value
lv	long value
fv	float value
dv	double value
baw	byte address (word register)
bal	byte address (long register)
wa	word address
la	long address
fa	float address
da	double address

r-operand is always a register

wr	word register:	w0, w1, w2, w3, w4, w5, w6, w7
lr	long register:	L0, L1, L2, L3, L4, L5, L6, L7
fr	float register:	f0, f1, f2, f3, f4, f5, f6, f7
dr	double register:	d0, d1, d2, d3, d4, d5, d6, d7

qc instructions: q-operand c-operand

q-operand is same as for a qr instruction (a value or an address)

c-operand is a condition code

c0:	cUNC, cLEU, cLU, cL, cLE, cE, cNE, cGE
c1:	cG, cGU, cGEU

qo instructions: q-operand

q-operand is same as for a qr instruction (a value or an address)

```
mr instructions:    m-operand       r-operand

    m-operand is either a value or an address
            gv      general value (immediate operand not allowed)
            ga      general address

    r-operand is always a register
            gr      general register (i.e., any register)



ir instructions:    i-operand       r-operand

    i-operand is a 5-bit integer
            sc      shift count

    r-operand is same as for a qr instruction (a register)



ij instructions:    i-operand       j-operand

    i-operand is a condition code
            cc:     cUNC, cLEU, cLU, cL, cLE, cE, cNE, cGE
                    cG,   cGU,  cGEU

    j-operand is a memory address
            ma      memory address



a3 instructions:    a3-operand

    a3-operand
            ma3     3-byte memory address




b1 instructions:    b1-operand

    b1-operand is a 1-byte integer
        stkc        stack change



b14 instructions:   b1-operand      b4-operand

    b1-operand is a 1-byte integer
        stkc        stack change

    b4-operand is a 4-byte bit mask
        bmsk        bit mask
```

n0 instructions:

no operands

n04 instructions: n4-operand

n4-operand is a 4-byte bit mask
bmsk bit mask

Some miscellaneous notation:

Synonyms for registers:
fp: frame pointer (= L6)
sp: stack pointer (= L7

Registers other than general registers:
pc: program counter
flags: flags register
LUF: less than unsigned flag
LF: less than signed flag
EF: equal flag
GF: greater than signed flag
GUF: greater than unsigned flag

Bytes and words:
lob: loworder byte of a word or a long
hob: highorder byte of a word or a long
low: loworder word of a long
how: highorder word of a long

In all cases in which an opcode takes typed operands, the last letter of the opcode indicates the type of the operands:

b: byte
w: word
l: long
f: float
d: double

Instruction Descriptions

The following pages contained detailed descriptions of all VMAX instructions, in alphabetical order by opcode name.

[Note: Each instruction description contains a "Flags" section, but at present there is no discussion of how flags are set by the instructions.]

absd - Absolute value of double

Format: qr Flags:

Syntax: absd dv dr

Semantics: dr := (dv < 0 ? -dv : dv)

Description: The absolute value of double dv is stored in double register dr.

Flags:

Examples:
```
absd    d1      d1      ; d1 := abs(d1)
absd    d0      d1      ; d1 := abs(d0)
absd    [L1]    d0      ; d0 := abs(double value
                                addressed by L1)
```

absf - Absolute value of float

Format: qr Flags:

Syntax: absf fv fr

Semantics: fr := (fv < 0 ? -fv : fv)

Description: The absolute value of float fv is stored in float register fr.

Flags:

Examples:
```
absf    f1      f1      ; f1 := abs(f1)
absf    f0      f1      ; f1 := abs(f0)
absf    [L1]    f0      ; f0 := abs(float value
                                addressed by L1)
```

absl - Absolute value of long

Format: qr Flags:

Syntax: absl lv lr

Semantics: lr := (lv < 0 ? -lv : lv)

Description: The absolute value of long lv is stored in long register lr.

Flags:

Examples:
```
absl    L1      L1      ; L1 := abs(L1)
absl    L0      L1      ; L1 := abs(L0)
absl    [L1]    L0      ; L0 := abs(long value
                                addressed by L1)
```

absw - Absolute value of word

Format: qr Flags:

Syntax: absw wv wr

Semantics: wr := (wv < 0 ? -wv : wv)

Description: The absolute value of word wv is stored in word register wr.

Flags:

Examples:
```
absw    w1      w1      ; w1 := abs(w1)
absw    w0      w1      ; w1 := abs(w0)
absw    [L1]    w0      ; w0 := abs(word value
                                addressed by L1)
```

addcl - Add long with carry

Format: qr Flags:

Syntax: addcl lv lr

Semantics: lr := lr + lv + carry bit

Description: Long value lv and the carry bit are added to long register lr. This instruction makes it possible to write multiple-precision arithmetic.

Flags: NOTE: The carry bit is not yet defined, so this instruction is not yet available.

```
Examples:       addcl   L1      L1      ; L1 := L1 + L1 + carry bit
                addcl   L0      L1      ; L1 := L1 + L0 + carry bit
                addcl   [L1]    L0      ; L0 := L0 + long value
                                                addressed by L1 + carry bit
```

addd - Add double

Format: qr Flags:

Syntax: addd dv dr

Semantics: dr := dr + dv

Description: Double value dv is added to double register dr.

Flags:

```
Examples:       addd    d1      d1      ; d1 := d1 + d1
                addd    d0      d1      ; d1 := d1 + d0
                addd    [L1]    d0      ; d0 := d0 + double value
                                                addressed by L1
```

addf - Add float

Format: qr Flags:

```
Syntax:         addf    fv      fr

Semantics:      fr := fr + fv

Description:    Float value fv is added to float register fr.

Flags:

Examples:       addf    f1      f1      ; f1 := f1 + f1
                addf    f0      f1      ; f1 := f1 + f0
                addf    [L1]    f0      ; f0 := f0 + float value
                                                addressed by L1
```

addl - Add long

Format: qr Flags:

```
Syntax:         addl    lv      lr

Semantics:      lr := lr + lv

Description:    Long value lv is added to long register lr.

Flags:

Examples:       addl    L1      L1      ; L1 := L1 + L1
                addl    L0      L1      ; L1 := L1 + L0
                addl    [L1]    L0      ; L0 := L0 + long value
                                                addressed by L1
```

addswl - Add signed word to long

Format: qr Flags:

Syntax: addswl wv lr

Semantics: lr := lr + sign-extend(wv)

Description: Word value wv is sign-extended to 32 bits, and then added to long register lr.

Flags:

```
Examples:       addswl  w1      L1      ; L1 := L1 + sign-extend(w1)
                addswl  w0      L1      ; L1 := L1 + sign-extend(w0)
                addswl  [L1]    L0      ; L0 := L0 + sign-extend(word value
                                                      addressed by L1)
```

adduwl - Add unsigned word to long

Format: qr Flags:

Syntax: adduwl wv lr

Semantics: lr := lr + zero-extend(wv)

Description: Word value wv is zero-extended to 32 bits, and then added to long register lr.

Flags:

```
Examples:       adduwl  w1      L1      ; L1 := L1 + zero-extend(w1)
                adduwl  w0      L1      ; L1 := L1 + zero-extend(w0)
                adduwl  [L1]    L0      ; L0 := L0 + zero-extend(word value
                                                      addressed by L1)
```

addw - Add word

Format: qr Flags:

```
Syntax:         addw    wv      wr

Semantics:      wr := wr + wv

Description:    Word value wv is added to word register wv.

Flags:

Examples:       addw    w1      w1      ; w1 := w1 + w1
                addw    w0      w1      ; w1 := w1 + w0
                addw    [L1]    w0      ; w0 := w0 + word value
                                                addressed by L1
```

andl - And long

Format: qr Flags:

```
Syntax:         andl    lv      lr

Semantics:      lr := lr & lv

Description:    Long value lv is bitwise and-ed to long register lr.

Flags:

Examples:       andl    L1      L1      ; L1 := L1 & L1
                andl    L0      L1      ; L1 := L1 & L0
                andl    [L1]    L0      ; L0 := L0 & long value
                                                addressed by L1
```

andw - And word

Format: qr Flags:

Syntax: andw wv wr

Semantics: wr := wr & wv

Description: Word value wv is bitwise and-ed to word register wr.

Flags:

```
Examples:       andw    w1      w1      ; w1 := w1 & w1
                andw    w0      w1      ; w1 := w1 & w0
                andw    [L1]    w0      ; w0 := w0 & word value
                                                addressed by L1
```

call - Call

Format: ij Flags:

Syntax: call cc ma

Semantics: if (cc) call EA(ma)

Description: If the condition indicated by condition code cc is TRUE, then call the routine at the effective address specified by ma. The return address is pushed onto the stack:

```
        sp := sp - 4
        Store pc at address contained in sp
        pc := EA(ma)
```

Flags:

```
Examples:       call    cUNC    ReadOne ; Unconditionally call ReadOne
                call    cG      Sort    ; if (GF == 1) call Sort
                call    cNE     [L1]    ; if (EF == 0) call routine whose
                                                address is in L1
```

callb - Call backward

Format: a3 Flags:

Syntax: callb ma3

Semantics: call EA(ma3)

Description: Call the routine at the effective address specified by ma3. The return address is pushed onto the stack:

```
sp := sp - 4
Store pc at address contained in sp
pc := pc - 2*ma3
```

Flags:

Examples:
```
callb   ReadOne         ; Call ReadOne
callb   Sort            ; Call Sort
```

callf - Call forward

Format: a3 Flags:

Syntax: callf ma3

Semantics: call EA(ma3)

Description: Call the routine at the effective address specified by ma3. The return address is pushed onto the stack:

```
sp := sp - 4
Store pc at address contained in sp
pc := pc + 2*ma3
```

Flags:

Examples:
```
callf   ReadOne         ; Call ReadOne
callf   Sort            ; Call Sort
```

cmpd - Compare double

Format: qr Flags:

Syntax: cmpd dv dr

Semantics: Set flags as if dr - dv were executed

Description: Double value dv is subtracted from double register dr, and the flags register is set accordingly. The dr register is NOT changed.

Flags:

```
Examples:       cmpd    d1      d0      ; Compare d1 with d0
                cmpd    [sp+2]  d1      ; Compare double at sp+2 with d1
                cmpd    [L1]    d0      ; Compare double value addressed
                                                by L1 with d0
```

cmpf - Compare float

Format: qr Flags:

Syntax: cmpf fv fr

Semantics: Set flags as if fr - fv were executed

Description: Float value fv is subtracted from float register fr, and the flags register is set accordingly. The fr register is NOT changed.

Flags:

```
Examples:       cmpf    f1      f0      ; Compare f1 with f0
                cmpf    [sp+2]  f1      ; Compare float at sp+2 with f1
                cmpf    [L1]    f0      ; Compare float value addressed
                                                by L1 with f0
```

cmpl - Compare long

Format: qr Flags:

Syntax: cmpl lv lr

Semantics: Set flags as if lr - lv were executed

Description: Long value lv is subtracted from long register lr, and the flags register is set accordingly. The lr register is NOT changed.

Flags:

```
Examples:       cmpl    L1      L0      ; Compare L1 with L0
                cmpl    [sp+2]  L1      ; Compare long at sp+2 with L1
                cmpl    [L1]    L0      ; Compare long value addressed
                                                by L1 with L0
```

cmplb - Compare loworder byte of long to byte

Format: qr Flags:

Syntax: cmplb bvl lr

Semantics: Set flags as if lob(lr) - bvl were executed

Description: Byte value bvl is subtracted from the loworder byte of long register lr, and the flags register is set accordingly. The lr register is NOT changed.

Flags:

```
Examples:       cmplb   L1      L2      ; Compare lob(L1) with lob(L2)
                cmplb   [sp+2]  L1      ; Compare byte at sp+2 with lob(L1)
                cmplb   [L1]    L0      ; Compare byte value addressed by
                                                L1 with lob(L0)
```

cmpw - Compare word

Format: qr Flags:

Syntax: `cmpw wv wr`

Semantics: Set flags as if wr - wv were executed

Description: Word value wv is subtracted from word register wr, and the flags register is set accordingly. The wr register is NOT changed.

Flags:

Examples:
```
cmpw    w1      w0      ; Compare w1 with w0
cmpw    [sp+2]  w1      ; Compare word at sp+2 with w1
cmpw    [L1]    w0      ; Compare word value addressed
                                by L1 with w0
```

cmpwb - Compare loworder byte of word to byte

Format: qr Flags:

Syntax: `cmpwb bvw wr`

Semantics: Set flags as if lob(wr) - bvw were executed

Description: Byte value bvw is subtracted from the loworder byte of word register wr, and the flags register is set accordingly. The wr register is NOT changed.

Flags:

Examples:
```
cmpwb   w1      w2      ; Compare lob(w1) with lob(w2)
cmpwb   [sp+2]  w1      ; Compare byte at sp+2 with lob(w1)
cmpwb   [L1]    w0      ; Compare byte value addressed by
                                L1 with lob(w0)
```

cvtbsl - Convert byte sign-extended to long
Format: qr Flags:

Syntax: cvtbsl bvl lr

Semantics:
```
lob(lr) := bvl
the three highorder bytes of lr := sign bit of bvl
```

Description: Byte value bvl is moved to the loworder byte of long register lr, and the sign of bvl is extended to fill the three highorder bytes of lr. Thus, the three highorder bytes of lr are either all zero bits or else all one bits.

Flags:

Examples:
```
cvtbsl  L1      L0      ; Move lob(L1) sign-extended to L0
cvtbsl  [sp+2]  L1      ; Move byte at sp+2 sign-ext to L1
cvtbsl  [L1]    L0      ; Move byte addressed by L1 sign-
                            extended to L0
```

cvtbsw - Convert byte sign-extended to word
Format: qr Flags:

Syntax: cvtbsw bvw wr

Semantics:
```
lob(wr) := bvw
hob(wr) := sign bit of bvw
```

Description: Byte value bvw is moved to the loworder byte of word register wr, and the sign of bvw is extended to fill the highorder byte of wr. Thus, the highorder byte of wr is either all zero bits or else all one bits.

Flags:

Examples:
```
cvtbsw  w1      w0      ; Move lob(w1) sign-extended to w0
cvtbsw  [sp+2]  w1      ; Move byte at sp+2 sign-ext to w1
cvtbsw  [L1]    w0      ; Move byte addressed by L1 sign-
                            extended to w0
```

cvtbzl - Convert byte zero-extended to long
Format: qr Flags:

Syntax: cvtbzl bvl lr

Semantics: lob(lr) := bvl
the three highorder bytes of lr := 0

Description: Byte value bvl is moved to the loworder byte of long register lr, and the three highorder bytes of lr are set to zero.

Flags:

Examples:
```
cvtbzl  L1      L0      ; Move lob(L1) zero-extended to L0
cvtbzl  [sp+2]  L1      ; Move byte at sp+2 zero-ext to L1
cvtbzl  [L1]    L0      ; Move byte addressed by L1 zero-
                                extended to L0
```

cvtbzw - Convert byte zero-extended to word
Format: qr Flags:

Syntax: cvtbzw bvw wr

Semantics: lob(wr) := bvw
hob(wr) := 0

Description: Byte value bvw is moved to the loworder byte of word register wr, and the highorder byte of wr is set to zero.

Flags:

Examples:
```
cvtbzw  w1      w0      ; Move lob(w1) zero-extended to w0
cvtbzw  [sp+2]  w1      ; Move byte at sp+2 zero-ext to w1
cvtbzw  [L1]    w0      ; Move byte addressed by L1 zero-
                                extended to w0
```

cvtdf - Convert double to float

Format: qr Flags:

Syntax: cvtdf dv fr

Semantics: fr := (float) dv

Description: Double value dv is converted to float and stored in float register fr.

QUESTION: What loss of precision, etc., can happen??

Flags:

Examples:
```
cvtdf   d1      f0      ; f0 := (float) d1
cvtdf   [sp+2]  f1      ; f1 := (float) (double at sp+2)
cvtdf   [L1]    f0      ; f0 := (float) (double addressed
                                          by L1)
```

cvtfd - Convert float to double

Format: qr Flags:

Syntax: cvtfd fv dr

Semantics: dr := (double) fv

Description: Float value fv is converted to double and stored in double register dr.

Flags:

Examples:
```
cvtfd   f1      d0      ; d0 := (double) f1
cvtfd   [sp+2]  d1      ; d1 := (double) (float at sp+2)
cvtfd   [L1]    d0      ; d0 := (double) (float addressed
                                          by L1)
```

cvtsld - Convert signed long to double
Format: qr Flags:

Syntax: cvtsld lv dr

Semantics: dr := (double) lv

Description: Signed long value lv is converted to double and stored in double register dr.

QUESTION: What about loss of precision?

Flags:

Examples:
```
cvtsld  L1      d0      ; d0 := (double) L1
cvtsld  [sp+2]  d1      ; d1 := (double) (long at sp+2)
cvtsld  [L1]    d0      ; d0 := (double) (long addressed
                                          by L1)
```

cvtslf - Convert signed long to float
Format: qr Flags:

Syntax: cvtslf lv fr

Semantics: fr := (float) lv

Description: Signed long value lv is converted to float and stored in float register fr.

QUESTION: What about loss of precision?

Flags:

Examples:
```
cvtslf  L1      f0      ; f0 := (float) L1
cvtslf  [sp+2]  f1      ; f1 := (float) (long at sp+2)
cvtslf  [L1]    f0      ; f0 := (float) (long addressed
                                          by L1)
```

cvttdsl - Convert truncated double to signed long
Format: qr Flags:

Syntax: cvttdsl dv lr

Semantics: lr := (long) dv

Description: Double value dv is converted to a signed long and stored in long register lr.

QUESTION: What loss of precision, etc., can happen??

Flags:

```
Examples:       cvttdsl d1      L0      ; L0 := (long) d1
                cvttdsl [sp+2]  L1      ; L1 := (long) (double at sp+2)
                cvttdsl [L1]    L0      ; L0 := (long) (double addressed
                                                  by L1)
```

cvttdul - Convert truncated double to unsigned long
Format: qr Flags:

Syntax: cvttdul dv lr

Semantics: lr := (unsigned long) dv

Description: Double value dv is converted to an unsigned long and stored in long register lr.

QUESTION: What loss of precision, etc., can happen??

Flags:

```
Examples:       cvttdul d1      L0      ; L0 := (unsigned long) d1
                cvttdul [sp+2]  L1      ; L1 := (unsigned long) (double
                                                  at sp+2)
                cvttdul [L1]    L0      ; L0 := (unsigned long) (double
                                                  addressed by L1)
```

cvttfsl - Convert truncated float to signed long
Format: qr Flags:

Syntax: cvttfsl fv lr

Semantics: lr := (long) fv

Description: Float value fv is converted to a signed long and stored in long register lr.

QUESTION: What loss of precision, etc., can happen??

Flags:

Examples:
```
cvttfsl f1      L0      ; L0 := (long) f1
cvttfsl [sp+2]  L1      ; L1 := (long) (float at sp+2)
cvttfsl [L1]    L0      ; L0 := (long) (float addressed
                                  by L1)
```

cvttful - Convert truncated float to unsigned long
Format: qr Flags:

Syntax: cvttful fv lr

Semantics: lr := (unsigned long) fv

Description: Float value fv is converted to an unsigned long and stored in long register lr.

QUESTION: What loss of precision, etc., can happen??

Flags:

Examples:
```
cvttful f1      L0      ; L0 := (unsigned long) f1
cvttful [sp+2]  L1      ; L1 := (unsigned long) (float
                                  at sp+2)
cvttful [L1]    L0      ; L0 := (unsigned long) (float
                                  addressed by L1)
```

cvtuld - Convert unsigned long to double
Format: qr Flags:

Syntax: cvtuld lv dr

Semantics: dr := (double) lv

Description: Unsigned long value lv is converted to double and stored in double register dr.

QUESTION: What about loss of precision?

Flags:

Examples:
```
cvtuld  L1      d0      ; d0 := (double) L1
cvtuld  [sp+2]  d1      ; d1 := (double) (long at sp+2)
cvtuld  [L1]    d0      ; d0 := (double) (long addressed
                                          by L1)
```

cvtulf - Convert unsigned long to float
Format: qr Flags:

Syntax: cvtulf lv fr

Semantics: fr := (float) lv

Description: Unsigned long value lv is converted to float and stored in float register fr.

QUESTION: What about loss of precision?

Flags:

Examples:
```
cvtulf  L1      f0      ; f0 := (float) L1
cvtulf  [sp+2]  f1      ; f1 := (float) (long at sp+2)
cvtulf  [L1]    f0      ; f0 := (float) (long addressed
                                          by L1)
```

cvtwsl - Convert word sign-extended to long
Format: qr Flags:

Syntax: `cvtwsl wv lr`

Semantics:
```
low(lr) := wv
how(lr) := sign bit of wv
```

Description: Word value wv is moved to the loworder word of long register lr, and the sign of wv is extended to fill the high-order word of lr. Thus, the highorder word of lr is either all zero bits or else all one bits.

Flags:

Examples:
```
cvtwsl  w1      L0      ; Move w1 sign-extended to L0
cvtwsl  [sp+2]  L1      ; Move word at sp+2 sign-ext to L1
cvtwsl  [L1]    L0      ; Move word addressed by L1 sign-
                                extended to L0
```

cvtwzl - Convert word zero-extended to long
Format: qr Flags:

Syntax: `cvtwzl wv lr`

Semantics:
```
low(lr) := wv
how(lr) := 0
```

Description: Word value wv is moved to the loworder word of long register lr, and the highorder word of lr is set to zero.

Flags:

Examples:
```
cvtwzl  w1      L0      ; Move w1 zero-extended to L0
cvtwzl  [sp+2]  L1      ; Move word at sp+2 zero-ext to L1
cvtwzl  [L1]    L0      ; Move word addressed by L1 zero-
                                extended to L0
```

divd - Divide double

Format: qr Flags:

Syntax: divd dv dr

Semantics: dr := dr / dv

Description: Double register dr is divided by double value dv, and the quotient is placed in dr.

Flags:

Examples:
```
divd    d1      d1      ; d1 := d1 / d1
divd    d0      d1      ; d1 := d1 / d0
divd    [L1]    d0      ; d0 := d0 / double value
                                addressed by L1
```

divf - Divide float

Format: qr Flags:

Syntax: divf fv fr

Semantics: fr := fr / fv

Description: Float register fr is divided by float value fv, and the quotient is placed in fr..

Flags:

Examples:
```
divf    f1      f1      ; f1 := f1 / f1
divf    f0      f1      ; f1 := f1 / f0
divf    [L1]    f0      ; f0 := f0 / float value
                                addressed by L1
```

divrsl - Divide with remainder signed long
Format: qr Flags:

Syntax: divrsl lv lr

Semantics: lr := lr / lv
lv := remainder (if lv is a register)

Description: Long register lr is divided by long value lv, and the quotient is placed in lr.. If lv is a register, then lv is set to the remainder of the division (so the original divisor is overwritten). If lv is not a register then no remainder is produced by the operation. All quantities are treated as SIGNED integers.

Flags:

Examples:
```
divrsl  L2      L1      ; L1 := L1 / L2;  L2 := remainder
divrsl  [sp+2]  L1      ; L1 := L1 / long at sp+2; no rmdr
divrsl  [L1]    L0      ; L0 := L0 / long value addressed
                                by L1 (no remainder)
```

divrslw - Divide with remainder signed long by word
Format: qr Flags:

Syntax: divrslw wv lr

Semantics: lr := lr / wv
wv := remainder (if wv is a register)

Description: Long register lr is divided by word value wv, and the quotient is placed in lr.. If wv is a register, then wv is set to the remainder of the division (so the original divisor is overwritten). If wv is not a register then no remainder is produced by the operation. All quantities are treated as SIGNED integers.

Flags:

Examples:
```
divrslw w2      L1      ; L1 := L1 / w2;  w2 := remainder
divrslw [sp+2]  L1      ; L1 := L1 / word at sp+2; no rmdr
divrslw [L1]    L0      ; L0 := L0 / word value addressed
                                by L1 (no remainder)
```

divrsw - Divide with remainder signed word
Format: qr Flags:

Syntax: divrsw wv wr

Semantics: wr := wr / wv
wv := remainder (if wv is a register)

Description: Word register wr is divided by word value wv, and the quotient is placed in wr.. If wv is a register, then wv is set to the remainder of the division (so the original divisor is overwritten). If wv is not a register then no remainder is produced by the operation. All quantities are treated as SIGNED integers.

Flags:

Examples:
```
divrsw  w2      w1      ; w1 := w1 / w2;  w2 := remainder
divrsw  [sp+2]  w1      ; w1 := w1 / word at sp+2; no rmdr
divrsw  [L1]    w0      ; w0 := w0 / word value addressed
                                by L1 (no remainder)
```

divrul - Divide with remainder unsigned long
Format: qr Flags:

Syntax: divrul lv lr

Semantics: lr := lr / lv
lv := remainder (if lv is a register)

Description: Long register lr is divided by long value lv, and the quotient is placed in lr.. If lv is a register, then lv is set to the remainder of the division (so the original divisor is overwritten). If lv is not a register then no remainder is produced by the operation. All quantities are treated as UNSIGNED integers.

Flags:

Examples:
```
divrul  L2      L1      ; L1 := L1 / L2;  L2 := remainder
divrul  [sp+2]  L1      ; L1 := L1 / long at sp+2; no rmdr
divrul  [L1]    L0      ; L0 := L0 / long value addressed
                                by L1 (no remainder)
```

divrulw - Divide with remainder unsigned long by word
Format: qr Flags:

Syntax: divrulw wv lr

Semantics: lr := lr / wv
wv := remainder (if wv is a register)

Description: Long register lr is divided by word value wv, and the quotient is placed in lr.. If wv is a register, then wv is set to the remainder of the division (so the original divisor is overwritten). If wv is not a register then no remainder is produced by the operation. All quantities are treated as UNSIGNED integers.

Flags:

Examples:
```
divrulw w2      L1      ; L1 := L1 / w2;  w2 := remainder
divrulw [sp+2]  L1      ; L1 := L1 / word at sp+2; no rmdr
divrulw [L1]    L0      ; L0 := L0 / word value addressed
                                by L1 (no remainder)
```

divruw - Divide with remainder unsigned word
Format: qr Flags:

Syntax: divruw wv wr

Semantics: wr := wr / wv
wv := remainder (if wv is a register)

Description: Word register wr is divided by word value wv, and the quotient is placed in wr.. If wv is a register, then wv is set to the remainder of the division (so the original divisor is overwritten). If wv is not a register then no remainder is produced by the operation. All quantities are treated as UNSIGNED integers.

Flags:

Examples:
```
divruw  w2      w1      ; w1 := w1 / w2;  w2 := remainder
divruw  [sp+2]  w1      ; w1 := w1 / word at sp+2; no rmdr
divruw  [L1]    w0      ; w0 := w0 / word value addressed
                                by L1 (no remainder)
```

divsl - Divide signed long

Format: qr Flags:

Syntax: `divsl lv lr`

Semantics: `lr := lr / lv`

Description: Long register lr is divided by long value lv, and the quotient is placed in lr.. No remainder is calculated. All quantities are treated as SIGNED integers.

Flags:

Examples:

```
divsl   L2      L1      ; L1 := L1 / L2
divsl   [sp+2]  L1      ; L1 := L1 / long at sp+2
divsl   [L1]    L0      ; L0 := L0 / long value addressed
                            by L1
```

divsw - Divide signed word

Format: qr Flags:

Syntax: `divsw wv wr`

Semantics: `wr := wr / wv`

Description: Word register wr is divided by word value wv, and the quotient is placed in wr.. No remainder is calculated. All quantities are treated as SIGNED integers.

Flags:

Examples:

```
divsw   w2      w1      ; w1 := w1 / w2
divsw   [sp+2]  w1      ; w1 := w1 / word at sp+2
divsw   [L1]    w0      ; w0 := w0 / word value addressed
                            by L1
```

divul - Divide unsigned long

Format: qr Flags:

Syntax: `divul lv lr`

Semantics: `lr := lr / lv`

Description: Long register lr is divided by long value lv, and the quotient is placed in lr.. No remainder is calculated. All quantities are treated as UNSIGNED integers.

Flags:

Examples:

```
divul   L2      L1      ; L1 := L1 / L2
divul   [sp+2]  L1      ; L1 := L1 / long at sp+2
divul   [L1]    L0      ; L0 := L0 / long value addressed
                            by L1
```

divuw - Divide unsigned word

Format: qr Flags:

Syntax: `divuw wv wr`

Semantics: `wr := wr / wv`

Description: Word register wr is divided by word value wv, and the quotient is placed in wr.. No remainder is calculated. All quantities are treated as UNSIGNED integers.

Flags:

Examples:

```
divuw   w2      w1      ; w1 := w1 / w2
divuw   [sp+2]  w1      ; w1 := w1 / word at sp+2
divuw   [L1]    w0      ; w0 := w0 / word value addressed
                            by L1
```

enter - Enter function

Format: b1 Flags:

Syntax: enter stkc

Semantics:
```
sp := sp - 4
Move contents of fp to address contained in sp
fp := sp
sp := sp - 2*stkc
```

Description: This instruction is used as the first instruction in a routine generated from a C function definition. Its effect is exactly equivalent to the action of these three instructions:

```
        pushl   fp
        movl    sp      fp
        subl    2*stkc  sp
```

Typically, 2*stkc is the number of bytes needed by local automatic variables of the function. Note that stkc is an unsigned quantity, scaled by a factor of 2 (because the smallest entity that can be pushed or popped is a word). Thus, at most 510 bytes can be reserved on the stack for local variables by the enter instruction. If more are needed, this is easily handled by generating an addl instruction. For example, say that a function requires 528 bytes for locals. Then the first two instructions in the function are

```
        enter   510
        subl     18     sp
```

Flags:

Examples:
```
enter   22              ; Reserve 22 bytes for locals
enter    0              ; Reserve no bytes for locals
```

entersav - Enter function and save registers
Format: b14 Flags:

Syntax: entersav stkc bmsk

Semantics:
```
sp := sp - 4
Move contents of fp to address contained in sp
fp := sp
sp := sp - 2*stkc
pushregs  bmsk
```

Description: This instruction is used as the first instruction in a routine generated from a C function definition. Its effect is exactly equivalent to the action of these four instructions:

```
pushl    fp
movl     sp     fp
subl     2*stkc sp
pushregs bmsk
```

Or, more succinctly, the effect of entersav is exactly equivalent to these two instructions:

```
enter    stkc
pushregs bmsk
```

Thus, entersav performs the function entry housekeeping of enter, followed by saving of up to 32 registers on the stack.

Flags:

Examples:
```
entersav 22 0x80000003  ; Reserve 22 bytes for locals and
                                          push w0, w1, and d7
entersav  0 0x30000008  ; Reserve no bytes for locals and
                                          push w3, d4, and d5
```

gmov - General move

Format: mr Flags:

Syntax: gmov n ga gr

Semantics: gr := n bytes from ga

Description: Move n bytes (n = 1, 2, 4, or 8) from general address ga to general register gr. The general address can be any memory address or any register; it cannot be an immediate value.

NOTE: How bytes are aligned in multiple registers and partial registers will be determined later.

Flags:

```
Examples:       gmov    8   d1     w0   ; w0 thru w3 := 8 bytes from d1
                gmov    1   [sp+2] f2   ; part of f2 := byte at sp+2
                gmov    4   [L1]   d0   ; part of d0 := 4 bytes addressed
                                                      by L1
```

gsto - General store

Format: mr Flags:

Syntax: gsto n gr ga

Semantics: ga := n bytes from gr

Description: Move n bytes (n = 1, 2, 4, or 8) from general register gr to general address ga. The general address can be any memory address or any register; it cannot be an immediate value.

NOTE: How bytes are aligned in multiple registers and partial registers will be determined later.

Flags:

```
Examples:       gsto    8   w0     d1   ; d1 := w0 thru w3
                gsto    1   f2   [sp+2] ; byte at sp+2 : = byte from f2
                gsto    4   d0   [L1]   ; 4 bytes addressed by L1 := bytes
                                                      from d0
```

halt - Halt the VMAX machine

Format: n0 Flags:

Syntax: halt

Semantics: Halt processing

Description: The VMAX stops processing instrutions.

QUESTION: We need to define exactly what happens: A message is sent from PCMAX2 to PC? Maybe some sort of return code should be transmitted?

Flags:

Example: halt ; Halt processing

jump - Jump

Format: ij Flags:

Syntax: jump cc ma

Semantics: if (cc) goto EA(ma)

Description: If the condition indicated by condition code cc is TRUE, then jump to the effective address specified by ma.

Flags:

Examples:
```
jump    cUNC    ReadOne ; Unconditionally goto ReadOne
jump    cG      Sort    ; if (GF == 1) goto Sort
jump    cNE     [L1]    ; if (EF == 0) goto routine whose
                                          address is in L1
```

jumpb - Jump backward

Format: a3 Flags:

Syntax: jumpb ma3

Semantics: goto EA(ma3)

Description: Jump to the effective address specified by ma3:

pc := pc - 2*ma3

Flags:

Examples:
```
jumpb   ReadOne         ; goto ReadOne
jumpb   Sort            ; goto Sort
```

jumpf - Jump forward

Format: a3 Flags:

Syntax: jumpf ma3

Semantics: goto EA(ma3)

Description: Jump to the effective address specified by ma3:

pc := pc + 2*ma3

Flags:

Examples:
```
jumpf   ReadOne         ; goto ReadOne
jumpf   Sort            ; goto Sort
```

leal - Load effective address

Format: qr Flags:

Syntax: leal lv lr

Semantics: lr := EA(lv)

Description: Move the effective address of long value lv into long register lr. If lv is an immediate value or the contents of a register, the effect is the same as movl lv lr.

Flags:

Examples:
```
leal    [L1+L2]  L0     ; L0 := L1 + L2
leal    14       L1     ; L1 := 14
leal    [L1+14]  L0     ; L0 := L1 + 14
```

leave - Leave function

Format: b1 Flags:

Syntax: leave stkc

Semantics:
```
sp := fp
Move long addressed by sp to fp
sp := sp + 4
Move long addressed by sp to pc
sp := sp + 4
sp := sp + 2*stkc
```

Description: This instruction is used to return from a routine generated from a C function definition. Its effect is exactly equivalent to the action of these three instructions:

```
movl    fp      sp
popl    fp
ret     stkc
```

Typically, 2*stkc is the number of bytes needed for parameters to the function. Part of the task of leave is to clear these parameters out of the stack. Note that stkc is an unsigned quantity, scaled by a factor of 2 (because the smallest entity that can be pushed or popped is a word). Thus, at most 510 bytes can be cleared from the stack by the leave instruction. If more must be cleared, then this must be done by a addl to sp following the call of the function.

Many implementations of C assume that every function can take a variable number of parameters. Thus "leave 0" is always generated, and the stack is adjusted by a addl instruction following the call.

Note that the stkc operand for a leave instruction is the number of bytes used by PARAMETERS, while the stkc operand for an enter instruction is the number of bytes used by LOCALS. Thus, the two instructions are not exactly symmetrical.

Flags:

Examples:
```
leave   22              ; Clear 22 bytes from stack
leave    0              ; Clear no bytes from stack
```

leave - Leave function and restore registers
Format: b14 Flags:

Syntax: leaveres stkc bmsk

Semantics:
```
popregs  bmsk
sp := fp
Move long addressed by sp to fp
sp := sp + 4
Move long addressed by sp to pc
sp := sp + 4
sp := sp + 2*stkc
```

Description: This instruction is used as the last instruction in a routine generated from a C function definition. Its effect is exactly equivalent to the action of these four instructions:

```
        popregs bmsk
        movl    fp      sp
        popl    fp
        ret     stkc
```

Or, more succinctly, the effect of leaveres is exactly equivalent to these two instructions:

```
        popregs bmsk
        leave   stkc
```

Thus, leaveres restores of up to 32 registers from the stack, and then performs the function exit housekeeping of leave.

Flags:

Examples:
```
leaveres 22 0x80000003  ; Pop d7, w1, w0, and clear 22
                                            bytes from stack
leaveres  0 0x30000008  ; Pop d5, d4, w3, and clear no
                                            bytes from stack
```

movbl - Move byte to loworder byte of long
Format: qr Flags:

Syntax: movbl bvl lr

Semantics: lob(lr) := bvl

Description: Byte value bvl is moved to the loworder byte of long register lr. The three highorder bytes of lr are NOT changed.

Flags:

```
Examples:       movbl   L1      L0      ; Move lob(L1) to lob(L0)
                movbl   [sp+2]  L1      ; Move byte at sp+2 to lob(L1)
                movbl   [L1]    L0      ; Move byte addressed by L1 to
                                                lob(L0)
```

movbw - Move byte to loworder byte of word
Format: qr Flags:

Syntax: movbw bvw wr

Semantics: lob(wr) := bvw

Description: Byte value bvw is moved to the loworder byte of word register wr. The highorder byte of wr is NOT changed.

Flags:

```
Examples:       movbw   w1      w0      ; Move lob(w1) to lob(w0)
                movbw   [sp+2]  w1      ; Move byte at sp+2 to lob(w1)
                movbw   [L1]    w0      ; Move byte addressed by L1 to
                                                lob(w0)
```

movd - Move double

Format: qr Flags:

Syntax: movd dv dr

Semantics: dr := dv

Description: Move double value dv to double register dr.

Flags:

Examples:
```
movd    d1      d0      ; d0 := d1
movd    [sp+2]  d1      ; d1 := double at sp+2
movd    [L1]    d0      ; d0 := double addressed by L1
```

movf - Move float

Format: qr Flags:

Syntax: movf fv fr

Semantics: fr := fv

Description: Move float value fv to float register fr.

Flags:

Examples:
```
movf    f1      f0      ; f0 := f1
movf    [sp+2]  f1      ; f1 := float at sp+2
movf    [L1]    f0      ; f0 := float addressed by L1
```

movflags - Move word to flags register

Format: qo Flags:

Syntax: movflags wv

Semantics: flags := wv

Description: Move word value wv to the flags register.

Flags:

```
Examples:       movflags  w1            ; flags := w1
                movflags  [sp+2]        ; flags := word at sp+2
                movflags  [L1]          ; flags := word addressed by L1
```

movl - Move long

Format: qr Flags:

Syntax: movl lv lr

Semantics: lr := lv

Description: Move long value lv to long register lr.

Flags:

```
Examples:       movl    L1      L0      ; L0 := L1
                movl    [sp+2]  L1      ; L1 := long at sp+2
                movl    [L1]    L0      ; L0 := long addressed by L1
```

movw - Move word

Format: qr Flags:

```
Syntax:         movw    wv      wr

Semantics:      wr := wv
```

Description: Move word value wv to word register wr.

Flags:

```
Examples:       movw    w1      w0      ; w0 := w1
                movw    [sp+2]  w1      ; w1 := word at sp+2
                movw    [L1]    w0      ; w0 := word addressed by L1
```

movwl - Move word to loworder word of long

Format: qr Flags:

```
Syntax:         movwl   wv      lr

Semantics:      low(lr) := wv
```

Description: Word value wv is moved to the loworder word of long register lr. The highorder word of lr is NOT changed.

Flags:

```
Examples:       movwl   w1      L0      ; Move w1 to low(L0)
                movwl   [sp+2]  L1      ; Move word at sp+2 to low(L1)
                movwl   [L1]    L0      ; Move word addressed by L1 to
                                                low(L0)
```

muld - Multipy double

Format: qr Flags:

Syntax: muld dv dr

Semantics: dr := dr * dv

Description: Double register dr is multiplied by double value dv.

Flags:

```
Examples:       muld    d1      d1      ; d1 := d1 * d1
                muld    d0      d1      ; d1 := d1 * d0
                muld    [L1]    d0      ; d0 := d0 * double value
                                                addressed by L1
```

mulf - Multipy float

Format: qr Flags:

Syntax: mulf fv fr

Semantics: fr := fr * fv

Description: Float register fr is multiplied by float value fv.

Flags:

```
Examples:       mulf    f1      f1      ; f1 := f1 * f1
                mulf    f0      f1      ; f1 := f1 * f0
                mulf    [L1]    f0      ; f0 := f0 * float value
                                                addressed by L1
```

mulsl - Multipy signed long

Format: qr Flags:

Syntax: mulsl lv lr

Semantics: lr := lr * lv

Description: Long register lr is multiplied by long value lv. Both quantities are treated as SIGNED integers.

Flags:

Examples:
```
mulsl   L2      L1      ; L1 := L1 * L2;
mulsl   [sp+2]  L1      ; L1 := L1 * long at sp+2;
mulsl   [L1]    L0      ; L0 := L0 * long value addressed
                            by L1
```

mulsw - Multipy signed word

Format: qr Flags:

Syntax: mulsw wv wr

Semantics: wr := wr * wv

Description: Word register wr is multiplied by word value wv. Both quantities are treated as SIGNED integers.

Flags:

Examples:
```
mulsw   w2      w1      ; w1 := w1 * w2;
mulsw   [sp+2]  w1      ; w1 := w1 * word at sp+2;
mulsw   [L1]    w0      ; w0 := w0 * word value addressed
                            by L1
```

mulswl - Multipy signed words yielding long
Format: qr Flags:

Syntax: mulswl wv lr

Semantics: lr := lr * wv

Description: The loworder word of long register lr is multiplied by word value wv, and the result is placed in long register lr. Note that the highorder word of lr can have any value before this instruction is executed; it is ignored. After the instruction is executed, the highorder word of lr is part of the product computed by this instruction. All quantities are treated as SIGNED integers.

Flags:

Examples:
```
mulswl  w2      L1      ; L1 := low(L1) * w2;
mulswl  [sp+2]  L1      ; L1 := low(L1) * word at sp+2;
mulswl  [L1]    L0      ; L0 := low(L0) * word value
                                          addressed by L1
```

mulul - Multipy unsigned long
Format: qr Flags:

Syntax: mulul lv lr

Semantics: lr := lr * lv

Description: Long register lr is multiplied by long value lv. Both quantities are treated as UNSIGNED integers.

Flags:

Examples:
```
mulul   L2      L1      ; L1 := L1 * L2;
mulul   [sp+2]  L1      ; L1 := L1 * long at sp+2;
mulul   [L1]    L0      ; L0 := L0 * long value addressed
                                  by L1
```

muluw - Multipy unsigned word

Format: qr Flags:

Syntax: muluw wv wr

Semantics: wr := wr * wv

Description: Word register wr is multiplied by word value wv. Both quantities are treated as UNSIGNED integers.

Flags:

Examples:
```
muluw   w2      w1      ; w1 := w1 * w2;
muluw   [sp+2]  w1      ; w1 := w1 * word at sp+2;
muluw   [L1]    w0      ; w0 := w0 * word value addressed
                            by L1
```

muluwl - Multipy unsigned words yielding long

Format: qr Flags:

Syntax: muluwl wv lr

Semantics: lr := lr * wv

Description: The loworder word of long register lr is multiplied by word value wv, and the result is placed in long register lr. Note that the highorder word of lr can have any value before this instruction is executed; it is ignored. After the instruction is executed, the highorder word of lr is part of the product computed by this instruction. All quantities are treated as UNSIGNED integers.

Flags:

Examples:
```
muluwl  w2      L1      ; L1 := low(L1) * w2;
muluwl  [sp+2]  L1      ; L1 := low(L1) * word at sp+2;
muluwl  [L1]    L0      ; L0 := low(L0) * word value
                            addressed by L1
```

```
negd  -  Negate double
                                        Format: qr       Flags:

Syntax:         negd    dv      dr

Semantics:      dr := -dv

Description:    Double register dr is set to the negation of double value
                dv.

Flags:

Examples:       negd    d1      d1      ; d1 := -d1
                negd    d0      d1      ; d1 := -d0
                negd    [L1]    d0      ; d0 := - (double value
                                                   addressed by L1)
```

```
negf  -  Negate float
                                        Format: qr       Flags:

Syntax:         negf    fv      fr

Semantics:      fr := -fv

Description:    Float register fr is set to the negation of float value
                fv.

Flags:

Examples:       negf    f1      f1      ; f1 := -f1
                negf    f0      f1      ; f1 := -f0
                negf    [L1]    f0      ; f0 := - (float value
                                                   addressed by L1)
```

negl - Negate long

Format: qr Flags:

Syntax: negl lv lr

Semantics: lr := -lv

Description: Long register lr is set to the negation of long value lv. The negation of a long is the 2's-complement of the long (so 0x80000000 is not changed by negation).

Flags:

```
Examples:       negl    L1      L1      ; L1 := -L1
                negl    L0      L1      ; L1 := -L0
                negl    [L1]    L0      ; L0 := - (long value
                                                addressed by L1)
```

negw - Negate word

Format: qr Flags:

Syntax: negw wv wr

Semantics: wr := -wv

Description: Word register wr is set to the negation of word value wv. The negation of a word is the 2's-complement of the word (so 0x8000 is not changed by negation).

Flags:

```
Examples:       negw    w1      w1      ; w1 := -w1
                negw    w0      w1      ; w1 := -w0
                negw    [L1]    w0      ; w0 := - (word value
                                                addressed by L1)
```

nop - No operation

Format: n0 Flags:

Syntax: nop

Semantics: No operation

Description: This instruction does nothing.

Flags:

Example: nop ; Kill some time

notl - Not long

Format: qr Flags:

Syntax: notl lv lr

Semantics: lr := ~lv

Description: Long register lr is set to the bitwise negation of long value lv.

Flags:

```
Examples:       notl    L1      L1      ; L1 := ~L1
                notl    L0      L1      ; L1 := ~L0
                notl    [L1]    L0      ; L0 := ~(long value addressed
                                                  by L1)
```

notw - Not word

Format: qr Flags:

Syntax: notw wv wr

Semantics: wr := ~wv

Description: Long register wr is set to the bitwise negation of long value wv.

Flags:

Examples:
```
notw    w1      w1      ; w1 := ~w1
notw    w0      w1      ; w1 := ~w0
notw    [L1]    w0      ; w0 := ~(word value addressed
                                    by L1)
```

orl - Or long

Format: qr Flags:

Syntax: orl lv lr

Semantics: lr := lr | lv

Description: Long value lv is bitwise or-ed to long register lr.

Flags:

Examples:
```
orl     L1      L1      ; L1 := L1 | L1
orl     L0      L1      ; L1 := L1 | L0
orl     [L1]    L0      ; L0 := L0 | long value
                                addressed by L1
```

orw - Or word

Format: qr Flags:

Syntax: orw wv wr

Semantics: wr := wr | wv

Description: Word value wv is bitwise or-ed to word register wr.

Flags:

```
Examples:       orw     w1      w1      ; w1 := w1 | w1
                orw     w0      w1      ; w1 := w1 | w0
                orw     [L1]    w0      ; w0 := w0 | word value
                                                addressed by L1
```

popd - Pop double

Format: qo Flags:

Syntax: popd da

Semantics: da := double addressed by sp
sp := sp + 8

Description: Pop a double off the stack into da.

Flags:

```
Examples:       popd    d1              ; Pop into d1
                popd    [fp+2]          ; Pop into double at fp+2
                popd    [L1]            ; Pop into double addressed by L1
```

popf - Pop float

Format: qo Flags:

Syntax: popf fa

Semantics: fa := float addressed by sp
sp := sp + 4

Description: Pop a float off the stack into fa.

Flags:

Examples:
```
popf    f1              ; Pop into f1
popf    [fp+2]          ; Pop into float at fp+2
popf    [L1]            ; Pop into float addressed by L1
```

popl - Pop long

Format: qo Flags:

Syntax: popl la

Semantics: la := long addressed by sp
sp := sp + 4

Description: Pop a long off the stack into la.

Flags:

Examples:
```
popl    L1              ; Pop into L1
popl    [fp+2]          ; Pop into long at fp+2
popl    [L1]            ; Pop into long addressed by L1
```

popregs - Pop multiple registers

Format: n04 Flags:

Syntax: popregs bmsk

Semantics: Multiple registers are popped.

Description: The bit mask bmsk is scanned from bit 31 to bit 0, and for each bit that is on, the corresponding register is popped (i.e., the top of the stack is popped into the register). See an earlier section for the numbering of the 32 major VMAX registers. Only as many bytes as a register holds are popped for each register. Thus, if bmsk indicates that d1, f1, L1, and w1 are to be popped, then 8-bytes, 4-bytes, 4-bytes, and 2-bytes are popped. Note that popregs and pushregs scan the bit mask in opposite orders, so the masked used to push a group of registers can also be used to pop the registers.

Flags:

Examples:
```
popregs  0x80000003     ; Pop d7, w1, and w0
popregs  0x30000008     ; Pop d5, d4, and w3
popregs  0              ; Pop nothing
```

popw - Pop word

Format: qo Flags:

Syntax: popw wa

Semantics:
```
wa := word addressed by sp
sp := sp + 2
```

Description: Pop a word off the stack into wa.

Flags:

Examples:
```
popw    w1              ; Pop into w1
popw    [fp+2]          ; Pop into word at fp+2
popw    [L1]            ; Pop into word addressed by L1
```

pushd - Push double

Format: qo Flags:

Syntax: pushd dv

Semantics: sp := sp - 8
Move dv to address contained in sp

Description: Push double value dv onto the stack

Flags:

```
Examples:       pushd   d1              ; Push d1
                pushd   [fp+2]          ; Push double at fp+2
                pushd   [L1]            ; Push double addressed by L1
                pushd   3.14159         ; Push double 3.14159
```

pushf - Push float

Format: qo Flags:

Syntax: pushf fv

Semantics: sp := sp - 4
Move fv to address contained in sp

Description: Push float value fv onto the stack

Flags:

```
Examples:       pushf   f1              ; Push f1
                pushf   [fp+2]          ; Push float at fp+2
                pushf   [L1]            ; Push float addressed by L1
                pushf   3.14159         ; Push float 3.14159
```

pushl - Push long

Format: qo Flags:

Syntax: pushl lv

Semantics: sp := sp - 4
Move lv to address contained in sp

Description: Push long value lv onto the stack

Flags:

Examples:
```
pushl   L1              ; Push L1
pushl   [fp+2]          ; Push long at fp+2
pushl   [L1]            ; Push long addressed by L1
pushl   0xaabbccdd      ; Push long 0xaabbccdd
```

pushregs - Push multiple registers

Format: n04 Flags:

Syntax: pushregs bmsk

Semantics: Multiple registers are pushed.

Description: The bit mask bmsk is scanned from bit 0 to bit 31, and for each bit that is on, the corresponding register is pushed onto the stack. See an earlier section for the numbering of the 32 major VMAX registers. Only as many bytes as a register holds are pushed for each register. Thus, if bmsk indicates that w1, L1, f1, and d1 are to be pushed, then 2-bytes, 4-bytes, 4-bytes, and 8-bytes are pushed. Note that pushregs and popregs scan the bit mask in opposite orders, so the masked used to push a group of registers can also be used to pop the registers.

Flags:

Examples:
```
pushregs  0x80000003    ; Push w0, w1, and d7
pushregs  0x30000008    ; Push w3, d4, and d5
pushregs  0             ; Push nothing
```

pushw - Push word

Format: qo Flags:

Syntax: pushw wv

Semantics: sp := sp - 2
Move wv to address contained in sp

Description: Push word value wv onto the stack

Flags:

Examples:
```
pushw   w1              ; Push w1
pushw   [fp+2]          ; Push word at fp+2
pushw   [L1]            ; Push word addressed by L1
pushw   0xeeff          ; Push word 0xeeff
```

remsl - Remainder signed long

Format: qr Flags:

Syntax: remsl lv lr

Semantics: lr := lr % lv

Description: Long register lr is divided by long value lv, and the remainder is place in lr. No quotient is calculated. All quantities are treated as SIGNED integers.

Flags:

Examples:
```
remsl   L2      L1      ; L1 := L1 % L2
remsl   [sp+2]  L1      ; L1 := L1 % long at sp+2
remsl   [L1]    L0      ; L0 := L0 % long value addressed
                                by L1
```

remsw - Remainder signed word

Format: qr Flags:

Syntax: remsw wv wr

Semantics: wr := wr % wv

Description: Word register wr is divided by word value wv, and the remainder is place in wr. No quotient is calculated. All quantities are treated as SIGNED integers.

Flags:

Examples:
```
remsw   w2      w1      ; w1 := w1 % w2
remsw   [sp+2]  w1      ; w1 := w1 % word at sp+2
remsw   [L1]    w0      ; w0 := w0 % word value addressed
                                by L1
```

remul - Remainder unsigned long

Format: qr Flags:

Syntax: remul lv lr

Semantics: lr := lr % lv

Description: Long register lr is divided by long value lv, and the remainder is place in lr. No quotient is calculated. All quantities are treated as UNSIGNED integers.

Flags:

Examples:
```
remul   L2      L1      ; L1 := L1 % L2
remul   [sp+2]  L1      ; L1 := L1 % long at sp+2
remul   [L1]    L0      ; L0 := L0 % long value addressed
                                by L1
```

remuw - Remainder unsigned word

Format: qr Flags:

Syntax: remuw wv wr

Semantics: wr := wr % wv

Description: Word register wr is divided by word value wv, and the remainder is place in wr. No quotient is calculated. All quantities are treated as UNSIGNED integers.

Flags:

Examples:

```
remuw   w2      w1      ; w1 := w1 % w2
remuw   [sp+2]  w1      ; w1 := w1 % word at sp+2
remuw   [L1]    w0      ; w0 := w0 % word value addressed
                            by L1
```

ret - Return from call

Format: b1 Flags:

Syntax: ret stkc

Semantics: Move long addressed by sp to pc
sp := sp + 4
sp := sp + 2*stkc

Description: This instruction is used to return from a subroutine. It pops the return address off the stack into pc, and then increments the stack by 2*stkc to clear parameters to the subroutine out of the stack. Note that stkc is an unsigned quantity, scaled by a factor of 2 (because the smallest entity that can be pushed or popped is a word). Thus, at most 510 bytes can be cleared from the stack by the ret instruction. If more must be cleared, then this must be done by a addl to sp following the call of the function.

Many implementations of C assume that every function can take a variable number of parameters. In such cases, "ret 0" is always generated, and the stack is adjusted by an addl instruction following the call.

Flags:

Examples:
```
ret     22              ; Clear 22 bytes from stack
ret      0              ; Clear no bytes from stack
```

rlil - Rotate left immediate long

Format: ir Flags:

Syntax: rlil sc lr

Semantics: lr := lr rotated left by sc bits

Description: Long register lr is rotated left by sc bits. Bits shifted out of the highorder end of the register are shifted into the loworder end of the register. Note that the maximum value sc can have is 31. This is no problem, since there is never any need to rotate a long register by 32 bits, specified as an immediate value. The sc value is UNSIGNED.

Flags:

```
Examples:       rlil     1      L1      ; Rotate L1 left 1 bit
                rlil    10      L2      ; Rotate L2 left 10 bits
                rlil    31      L3      ; Rotate L3 left 31 bits
```

rliw - Rotate left immediate word

Format: ir Flags:

Syntax: rliw sc wr

Semantics: wr := wr rotated left by sc bits

Description: Word register wr is rotated left by sc bits. Bits shifted out of the highorder end of the register are shifted into the loworder end of the register. If sc >= 16, then the effect is the same as if the number of bits rotated is sc mod 16. The sc value is UNSIGNED.

Flags:

```
Examples:       rliw     1      w1      ; Rotate w1 left 1 bit
                rliw    10      w2      ; Rotate w2 left 10 bits
                rliw    31      w3      ; Rotate w3 left 15 bits
```

rll - Rotate left long

Format: qr Flags:

Syntax: rll bvl lr

Semantics: lr := lr rotated left by bvl bits

Description: Long register lr is rotated left by bvl bits. Bits shifted out of the highorder end of the register are shifted into the loworder end of the register. If bvl >= 32, then the effect is the same as if the number of bits rotated is bvl mod 32. The bvl value is UNSIGNED.

Flags:

```
Examples:       rll     L2      L1      ; Rotate L1 left by lob(L2) bits
                rll     [sp+2]  L2      ; Rotate L2 left by no. bits spec-
                                            ified by byte at sp+2
                rll     [L1]    L3      ; Rotate L3 left by no. bits spec-
                                            ified by byte addressed by L1
```

rlw - Rotate left word

Format: qr Flags:

Syntax: rlw bvw wr

Semantics: wr := wr rotated left by bvw bits

Description: Word register wr is rotated left by bvw bits. Bits shifted out of the highorder end of the register are shifted into the loworder end of the register. If bvw >= 16, then the effect is the same as if the number of bits rotated is bvw mod 16. The bvw value is UNSIGNED.

Flags:

```
Examples:       rlw     w2      w1      ; Rotate w1 left by lob(w2) bits
                rlw     [sp+2]  w2      ; Rotate w2 left by no. bits spec-
                                            ified by byte at sp+2
                rlw     [L1]    w3      ; Rotate w3 left by no. bits spec-
                                            ified by byte addressed by L1
```

rril - Rotate right immediate long

Format: ir Flags:

Syntax: rril sc lr

Semantics: lr := lr rotated right by sc bits

Description: Long register lr is rotated right by sc bits. Bits shifted out of the loworder end of the register are shifted into the highorder end of the register. Note that the maximum value sc can have is 31. This is no problem, since there is never any need to rotate a long register by 32 bits, specified as an immediate value. The sc value is UNSIGNED.

Flags:

Examples:
```
rril     1      L1      ; Rotate L1 right 1 bit
rril    10      L2      ; Rotate L2 right 10 bits
rril    31      L3      ; Rotate L3 right 31 bits
```

rriw - Rotate right immediate word

Format: ir Flags:

Syntax: rriw sc wr

Semantics: wr := wr rotated right by sc bits

Description: Word register wr is rotated right by sc bits. Bits shifted out of the loworder end of the register are shifted into the highorder end of the register. If sc >= 16, then the effect is the same as if the number of bits rotated is sc mod 16. The sc value is UNSIGNED.

Flags:

Examples:
```
rriw     1      w1      ; Rotate w1 right 1 bit
rriw    10      w2      ; Rotate w2 right 10 bits
rriw    31      w3      ; Rotate w3 right 15 bits
```

rrl - Rotate right long

Format: qr Flags:

Syntax: rrl bvl lr

Semantics: lr := lr rotated right by bvl bits

Description: Long register lr is rotated right by bvl bits. Bits shifted out of the loworder end of the register are shifted into the highorder end of the register. If bvl >= 32, then the effect is the same as if the number of bits rotated is bvl mod 32. The bvl value is UNSIGNED.

Flags:

```
Examples:       rrl     L2      L1      ; Rotate L1 right by lob(L2) bits
                rrl     [sp+2]  L2      ; Rotate L2 right by no. bits spe-
                                            cifed by byte at sp+2
                rrl     [L1]    L3      ; Rotate L3 right by no. bits spe-
                                            cifed by byte addressed by L1
```

rrw - Rotate right word

Format: qr Flags:

Syntax: rrw bvw wr

Semantics: wr := wr rotated right by bvw bits

Description: Word register wr is rotated right by bvw bits. Bits shifted out of the loworder end of the register are shifted into the highorder end of the register. If bvw >= 16, then the effect is the same as if the number of bits rotated is bvw mod 16. The bvw value is UNSIGNED.

Flags:

```
Examples:       rrw     w2      w1      ; Rotate w1 right by lob(w2) bits
                rrw     [sp+2]  w2      ; Rotate w2 right by no. bits spe-
                                            cifed by byte at sp+2
                rrw     [L1]    w3      ; Rotate w3 right by no. bits spe-
                                            cifed by byte addressed by L1
```

set0l - Store condition(0) in long

Format: qc Flags:

Syntax: set0l c0 la

Semantics: la := (condition c0 is TRUE)

Description: The long addressed by la is set to the integer value 1 if the condition specified by c0 is TRUE. Otherwise, it is set to 0. Note that only conditions 1 through 7 can be specified by c0; see the set1l instruction for conditions 8 through 14. (See the description of the i-operand of the ij-format for a list of all conditions.)

Flags:

Examples:
```
set0l   cNE     L2      ; L2 := (EF==0)
set0l   cG      [sp+2]  ; Long at sp+2 := (GF==1)
set0l   cLU     [L1]    ; Long addressed by L1 := (LUF==1)
```

set0w - Store condition(0) in word

Format: qc Flags:

Syntax: set0w c0 wa

Semantics: wa := (condition c0 is TRUE)

Description: The word addressed by wa is set to the integer value 1 if the condition specified by c0 is TRUE. Otherwise, it is set to 0. Note that only conditions 1 through 7 can be specified by c0; see the set1w instruction for conditions 8 through 14. (See the description of the i-operand of the ij-format for a list of all conditions.)

Flags:

Examples:
```
set0w   cNE     w2      ; w2 := (EF==0)
set0w   cG      [sp+2]  ; Word at sp+2 := (GF==1)
set0w   cLU     [L1]    ; Word addressed by L1 := (LUF==1)
```

set1l - Store condition(1) in long

Format: qc Flags:

Syntax: set1l c0 la

Semantics: la := (condition c1+8 is TRUE)

Description: The long addressed by la is set to the integer value 1 if the condition specified by c1+8 is TRUE. Otherwise, it is set to 0. Note that only conditions 8 through 14 can be specified by c1; see the set0l instruction for conditions 1 through 7. (See the description of the i-operand of the ij-format for a list of all conditions.)

Flags:

```
Examples:       set1l   cL      L2      ; L2 := (LF==1)
                set1l   cE      [sp+2]  ; Long at sp+2 := (EF==1)
                set1l   cGU     [L1]    ; Long addressed by L1 := (GUF==1)
```

set1w - Store condition(1) in word

Format: qc Flags:

Syntax: set1w c0 wa

Semantics: wa := (condition c1+8 is TRUE)

Description: The word addressed by wa is set to the integer value 1 if the condition specified by c1+8 is TRUE. Otherwise, it is set to 0. Note that only conditions 8 through 14 can be specified by c1; see the set0w instruction for conditions 1 through 7. (See the description of the i-operand of the ij-format for a list of all conditions.)

Flags:

```
Examples:       set1w   cL      w2      ; w2 := (LF==1)
                set1w   cE      [sp+2]  ; Word at sp+2 := (EF==1)
                set1w   cGU     [L1]    ; Word addressed by L1 := (GUF==1)
```

slil - Shift left immediate long

Format: ir Flags:

Syntax: slil sc lr

Semantics: lr := lr shifted left by sc bits

Description: Long register lr is shifted left by sc bits. Bits shifted out of the highorder end of the register are lost, and zero bits are shifted into the loworder end of the register. Note that the maximum value sc can have is 31. This is no problem, since there is never any need to shift a long register by 32 bits, specified as an immediate value. The sc value is UNSIGNED.

Flags:

Examples:
```
slil     1      L1      ; Shift L1 left 1 bit
slil    10      L2      ; Shift L2 left 10 bits
slil    31      L3      ; Shift L3 left 31 bits
```

sliw - Shift left immediate word

Format: ir Flags:

Syntax: sliw sc wr

Semantics: wr := wr shifted left by sc bits

Description: Word register wr is shifted left by sc bits. Bits shifted out of the highorder end of the register are lost, and zero bits are shifted into the loworder end of the register. If sc >= 16, then wr is set to zero. The sc value is UNSIGNED.

Flags:

Examples:
```
sliw     1      w1      ; Shift w1 left 1 bit
sliw    10      w2      ; Shift w2 left 10 bits
sliw    16      w3      ; w3 := 0
```

sll - Shift left long

Format: qr Flags:

Syntax: sll bvl lr

Semantics: lr := lr shifted left by bvl bits

Description: Long register lr is shifted left by bvl bits. Bits shifted out of the highorder end of the register are lost, and zero bits are shifted into the loworder end of the register. If bvl >= 32, then lr is set to zero. The bvl value is UNSIGNED.

Flags:

```
Examples:       sll     L2      L1      ; Shift L1 left by lob(L2) bits
                sll     [sp+2]  L2      ; Shift L2 left by no. bits spec-
                                            ifed by byte at sp+2
                sll     [L1]    L3      ; Shift L3 left by no. bits spec-
                                            ifed by byte addressed by L1
```

slw - Shift left word

Format: qr Flags:

Syntax: slw bvw wr

Semantics: wr := wr shifted left by bvw bits

Description: Word register wr is shifted left by bvw bits. Bits shifted out of the highorder end of the register are lost, and zero bits are shifted into the loworder end of the register. If bvw >= 16, then wr is set to zero. The bvw value is UNSIGNED.

Flags:

```
Examples:       slw     w2      w1      ; Shift w1 left by lob(w2) bits
                slw     [sp+2]  w2      ; Shift w2 left by no. bits spec-
                                            ifed by byte at sp+2
                slw     [L1]    w3      ; Shift w3 left by no. bits spec-
                                            ifed by byte addressed by L1
```

sqrtd - Square root of double

Format: qr Flags:

Syntax: sqrtd dv dr

Semantics: dr := sqrt(dv)

Description: The square root of double dv is stored in double register dr.

QUESTION: What if dv < 0?

Flags:

Examples:
```
sqrtd   d1      d1      ; d1 := sqrt(d1)
sqrtd   d0      d1      ; d1 := sqrt(d0)
sqrtd   [L1]    d0      ; d0 := sqrt(double value
                                addressed by L1)
```

sqrtf - Square root of float

Format: qr Flags:

Syntax: sqrtf fv fr

Semantics: fr := sqrt(fv)

Description: The square root of float fv is stored in float register fr.

QUESTION: What if fv < 0?

Flags:

Examples:
```
sqrtf   f1      f1      ; f1 := sqrt(f1)
sqrtf   f0      f1      ; f1 := sqrt(f0)
sqrtf   [L1]    f0      ; f0 := sqrt(float value
                                addressed by L1)
```

srail - Shift right arithmetic immediate long
Format: ir Flags:

Syntax: srail sc lr

Semantics: lr := lr shifted right arithmetically by sc bits

Description: Long register lr is shifted right arithmetically by sc bits. Bits shifted out of the loworder end of the register are lost. The sign bit is shifted into the highorder bits of the register. Note that the maximum value sc can have is 31. This is not a problem, because after a shift of 31 bits, lr is completely filled with the sign bit, i.e., lr is either 0 or -1. The sc value is UNSIGNED.

Flags:

Examples:
```
srail    1      L1      ; Shift L1 right arith. 1 bit
srail   10      L2      ; Shift L2 right arith. 10 bits
srail   31      L3      ; Shift L3 right arith. 31 bits
```

sraiw - Shift right arithmetic immediate word
Format: ir Flags:

Syntax: sraiw sc wr

Semantics: wr := wr shifted right arithmetically by sc bits

Description: Word register wr is shifted right arithmetically by sc bits. Bits shifted out of the loworder end of the register are lost. The sign bit is shifted into the highorder bits of the register. If sc >= 15, then wr is completely filled with the sign bit, i.e., wr is either 0 or -1. The sc value is UNSIGNED.

Flags:

Examples:
```
sraiw    1      w1      ; Shift w1 right arith. 1 bit
sraiw   10      w2      ; Shift w2 right arith. 10 bits
sraiw   15      w3      ; w3 := 0 or -1
```

sral - Shift right arithmetic long

Format: qr Flags:

Syntax: sral bvl lr

Semantics: lr := lr shifted right arithmetically by bvl bits

Description: Long register lr is shifted right arithmetically by bvl bits. Bits shifted out of the loworder end of the register are lost. The sign bit is shifted into the highorder bits of the register. If bvl >= 31, then lr is completely filled with the sign bit, i.e., lr is either 0 or -1. The bvl value is UNSIGNED.

Flags:

```
Examples:       sral    L2      L1      ; Shift L1 right arith. by lob(L2)
                                            bits
                sral    [sp+2]  L2      ; Shift L2 right arith. by no.
                                            bits specifed by byte at sp+2
                sral    [L1]    L3      ; Shift L3 right arith. by no.
                                            bits specifed by byte address-
                                            ed by L1
```

sraw - Shift right arithmetic word

Format: qr Flags:

Syntax: sraw bvw wr

Semantics: wr := wr shifted right arithmetically by bvw bits

Description: Word register wr is shifted right arithmetically by bvw bits. Bits shifted out of the loworder end of the register are lost. The sign bit is shifted into the highorder bits of the register. If bvw >= 15, then wr is completely filled with the sign bit, i.e., wr is either 0 or -1. The bvw value is UNSIGNED.

Flags:

```
Examples:       sraw    w2      w1      ; Shift w1 right arith. by lob(w2)
                                            bits
                sraw    [sp+2]  w2      ; Shift w2 right arith. by no.
                                            bits specifed by byte at sp+2
                sraw    [L1]    w3      ; Shift w3 right arith. by no.
                                            bits specifed by byte address-
                                            ed by L1
```

srlil - Shift right logical immediate long
Format: ir Flags:

Syntax: srlil sc lr

Semantics: lr := lr shifted right logically by sc bits

Description: Long register lr is shifted right logically by sc bits. Bits shifted out of the loworder end of the register are lost, and zero bits are shifted into the highorder end of the register. Note that the maximum value sc can have is 31. This is not a problem, because a shift of 32 bits causes lr to be set to zero. The sc value is UNSIGNED.

Flags:

```
Examples:       srlil    1      L1      ; Shift L1 right logical 1 bit
                srlil   10      L2      ; Shift L2 right logical 10 bits
                srlil   31      L3      ; Shift L3 right logical 31 bits
```

srliw - Shift right logical immediate word
Format: ir Flags:

Syntax: srliw sc wr

Semantics: wr := wr shifted right logically by sc bits

Description: Word register wr is shifted right logically by sc bits. Bits shifted out of the loworder end of the register are lost, and zero bits are shifted into the highorder end of the register. If sc >= 16, then wr is set to zero. The sc value is UNSIGNED.

Flags:

```
Examples:       srliw    1      w1      ; Shift w1 right logical 1 bit
                srliw   10      w2      ; Shift w2 right logical 10 bits
                srliw   16      w3      ; w3 := 0
```

srll - Shift right logical long

Format: qr Flags:

Syntax: srll bvl lr

Semantics: lr := lr shifted right logically by bvl bits

Description: Long register lr is shifted right logically by bvl bits. Bits shifted out of the loworder end of the register are lost, and zero bits are shifted into the highorder end of the register. If bvl >= 32, then lr is set to zero. The bvl value is UNSIGNED.

Flags:

```
Examples:       srll    L2      L1      ; Shift L1 right logical by
                                            lob(L2) bits
                srll    [sp+2]  L2      ; Shift L2 right logical by no.
                                            bits specifed by byte at sp+2
                srll    [L1]    L3      ; Shift L3 right logical by no.
                                            bits specifed by byte address-
                                            ed by L1
```

srlw - Shift right logical word

Format: qr Flags:

Syntax: srlw bvw wr

Semantics: wr := wr shifted right logically by bvw bits

Description: Word register wr is shifted right logically by bvw bits. Bits shifted out of the loworder end of the register are lost, and zero bits are shifted into the highorder end of the register. If bvw >= 16, then wr is set to zero. The bvw value is UNSIGNED.

Flags:

```
Examples:       srlw    w2      w1      ; Shift w1 right logical by
                                            lob(w2) bits
                srlw    [sp+2]  w2      ; Shift w2 right logical by no.
                                            bits specifed by byte at sp+2
                srlw    [L1]    w3      ; Shift w3 right logical by no.
                                            bits specifed by byte address-
                                            ed by L1
```

stod - Store double

Format: qr Flags:

Syntax: stod dr da

Semantics: da := dr

Description: Store double register dr at double address da.

Flags:

```
Examples:       stod    d0      d1      ; d1 := d0
                stod    d1      [sp+2]  ; double at sp+2 := d1
                stod    d0      [L1]    ; double addressed by L1 := d0
```

stof - Store float

Format: qr Flags:

Syntax: stof fr fa

Semantics: fa := fr

Description: Store float register fr at float address fa.

Flags:

```
Examples:       stof    f0      f1      ; f1 := f0
                stof    f1      [sp+2]  ; float at sp+2 := f1
                stof    f0      [L1]    ; float addressed by L1 := f0
```

stoflags - Store flags register into word

Format: qo Flags:

Syntax: stoflags wa

Semantics: wa := flags

Description: Store the flags register at word address wa.

Flags:

```
Examples:       stoflags  w1            ; w1 := flags
                stoflags  [sp+2]        ; word at sp+2 := flags
                stoflags  [L1]          ; word addressed by L1 := flags
```

stol - Store long

Format: qr Flags:

Syntax: stol lr la

Semantics: la := lr

Description: Store long register lr at long address la.

Flags:

```
Examples:       stol    L0      L1      ; L1 := L0
                stol    L1      [sp+2]  ; long at sp+2 := L1
                stol    L0      [L1]    ; long addressed by L1 := L0
```

stolb - Store loworder byte of long into byte
Format: qr Flags:

Syntax: stolb lr bal

Semantics: bal := lob(lr)

Description: The loworder byte of long register lr is stored at byte address bal. Only a single byte of the destination is changed.

Flags:

Examples:
```
stolb   L0      L1      ; lob(L1) := lob(L0)
stolb   L1      [sp+2]  ; byte at sp+2 := lob(L1)
stolb   L0      [L1]    ; byte addressed by L1 := lob(L0)
```

stolw - Store loworder word of long into word
Format: qr Flags:

Syntax: stolw lr wa

Semantics: wa := low(lr)

Description: The loworder word of long register lr is stored at word address wa. Only a single word of the destination is changed.

Flags:

Examples:
```
stolw   L0      w1      ; w1 := low(L0)
stolw   L1      [sp+2]  ; word at sp+2 := low(L1)
stolw   L0      [L1]    ; word addressed by L1 := low(L0)
```

stow - Store word

Format: qr Flags:

Syntax: stow wr wa

Semantics: wa := wr

Description: Store word register wr at word address wa.

Flags:

Examples:
```
stow    w0      w1      ; w1 = w0
stow    w1      [sp+2]  ; word at sp+2 := w1
stow    w0      [L1]    ; word addressed by L1 := w0
```

stowb - Store loworder byte of word into byte

Format: qr Flags:

Syntax: stowb wr baw

Semantics: baw := lob(wr)

Description: The loworder byte of word register wr is stored at byte address baw. Only a single byte of the destination is changed.

Flags:

Examples:
```
stowb   w0      w1      ; lob(w1) := lob(w0)
stowb   w1      [sp+2]  ; byte at sp+2 := lob(w1)
stowb   w0      [L1]    ; byte addressed by L1 := lob(w0)
```

subcl - Subtract long with carry

Format: qr Flags:

Syntax: subcl lv lr

Semantics: lr := lr - lv - carry bit

Description: Long value lv and the carry bit are subtracted from long register lr. This instruction makes it possible to write multiple-precision arithmetic.

Flags: NOTE: The carry bit is not yet defined, so this instruction is not yet available.

```
Examples:       subcl   L1      L1      ; L1 := L1 - L1 - carry bit
                subcl   L0      L1      ; L1 := L1 - L0 - carry bit
                subcl   [L1]    L0      ; L0 := L0 - long value
                                                addressed by L1 - carry bit
```

subd - Subtract double

Format: qr Flags:

Syntax: subd dv dr

Semantics: dr := dr - dv

Description: Double value dv is subtracted from double register dr.

Flags:

```
Examples:       subd    d1      d1      ; d1 := d1 - d1
                subd    d0      d1      ; d1 := d1 - d0
                subd    [L1]    d0      ; d0 := d0 - double value
                                                addressed by L1
```

subf - Subtract float

Format: qr Flags:

Syntax: subf fv fr

Semantics: fr := fr - fv

Description: Float value fv is subtracted from float register fr.

Flags:

Examples:
```
subf    f1      f1      ; f1 := f1 - f1
subf    f0      f1      ; f1 := f1 - f0
subf    [L1]    f0      ; f0 := f0 - float value
                                addressed by L1
```

subl - Subtract long

Format: qr Flags:

Syntax: subl lv lr

Semantics: lr := lr - lv

Description: Long value lv is subtracted from long register lr.

Flags:

Examples:
```
subl    L1      L1      ; L1 := L1 - L1
subl    L0      L1      ; L1 := L1 - L0
subl    [L1]    L0      ; L0 := L0 - long value
                                addressed by L1
```

subswl - Subtract signed word from long
Format: qr Flags:

Syntax: subswl wv lr

Semantics: lr := lr - sign-extend(wv)

Description: Word value wv is sign-extended to 32 bits, and then subtracted from long register lr.

Flags:

Examples:
```
subswl  w1      L1      ; L1 := L1 - sign-extend(w1)
subswl  w0      L1      ; L1 := L1 - sign-extend(w0)
subswl  [L1]    L0      ; L0 := L0 - sign-extend(word value
                                     addressed by L1)
```

subuwl - Subtract unsigned word from long
Format: qr Flags:

Syntax: subuwl wv lr

Semantics: lr := lr - zero-extend(wv)

Description: Word value wv is zero-extended to 32 bits, and then subtracted from long register lr.

Flags:

Examples:
```
subuwl  w1      L1      ; L1 := L1 - zero-extend(w1)
subuwl  w0      L1      ; L1 := L1 - zero-extend(w0)
subuwl  [L1]    L0      ; L0 := L0 - zero-extend(word value
                                     addressed by L1)
```

subw - Subtract word

Format: qr Flags:

```
Syntax:         subw    wv      wr

Semantics:      wr := wr - wv

Description:    Word value wv is subtracted from word register wr.

Flags:

Examples:       subw    w1      w1      ; w1 := w1 - w1
                subw    w0      w1      ; w1 := w1 - w0
                subw    [L1]    w0      ; w0 := w0 - word value
                                                addressed by L1
```

xorl - Exclusive or long

Format: qr Flags:

```
Syntax:         xorl    lv      lr

Semantics:      lr := lr ^ lv

Description:    Long value lv is bitwise exclusive-or-ed to long register
                lr.

Flags:

Examples:       xorl    L1      L1      ; L1 := L1 ^ L1
                xorl    L0      L1      ; L1 := L1 ^ L0
                xorl    [L1]    L0      ; L0 := L0 ^ long value
                                                addressed by L1
```

xorw - Exclusive or word

Format: qr Flags:

Syntax: xorw wv wr

Semantics: wr := wr ^ wv

Description: Word value wv is bitwise exclusive-or-ed to word register wr.

Flags:

Examples:

```
xorw    w1      w1      ; w1 := w1 ^ w1
xorw    w0      w1      ; w1 := w1 ^ w0
xorw    [L1]    w0      ; w0 := w0 ^ word value
                                addressed by L1
```

APPENDIX A: Instructions Grouped by Format

This appendix contains a list of all VMAX instructions, organized by format:

format	number operands	first operand	second operand	examples
a3	1	3-byte adr	-	jump, call
b1	1	1-byte int	-	ret
b14	2	1-byte int	4-byte msk	entersav
ij	2	cond. code	jump adr	jump, call
ir	2	imm. int	register	shift, rotate
mr	2	general	register	gmov, gsto
n0	0	-	-	halt, nop
n04	1	4-byte msk	-	pushregs
qc	2	general	cond. code	store cond.
qo	1	general	-	push, pop
qr	2	general	register	add, move

a3 Format

Opcode	Operands	Instruction	Format	Function
callb	ma3	Call backward	a3	jump
callf	ma3	Call forward	a3	jump
jumpb	ma3	Jump backward	a3	jump
jumpf	ma3	Jump forward	a3	jump

b1 Format

Opcode	Operands	Instruction	Format	Function
enter	stkc	Enter function	b1	stack
leave	stkc	Leave function	b1	jump
ret	stkc	Return from call	b1	jump

b14 Format

Opcode	Operands	Instruction	Format	Function
entersav	stkc bmsk	Enter func. and save regs	b14	stack
leaveres	stkc bmsk	Leave func. and restore regs	b14	jump

ij Format

Opcode	Operands	Instruction	Format	Function
call	cc ma	Call	ij	jump
jump	cc ma	Jump	ij	jump

ir Format

Opcode	Operands		Instruction	Format	Function
rlil	sc	lr	Rotate left immediate long	ir	shift
rliw	sc	wr	Rotate left immediate word	ir	shift
rril	sc	lr	Rotate right immediate long	ir	shift
rriw	sc	wr	Rotate right immediate word	ir	shift
slil	sc	lr	Shift left immediate long	ir	shift
sliw	sc	wr	Shift left immediate word	ir	shift
srail	sc	lr	Shift right arithmetic imm long	ir	shift
sraiw	sc	wr	Shift right arithmetic imm word	ir	shift
srlil	sc	lr	Shift right logical imm long	ir	shift
srliw	sc	wr	Shift right logical imm word	ir	shift

mr Format

Opcode	Operands		Instruction	Format	Function
gmov	ga	gr	General move	mr	move
gsto	ga	gr	General store	mr	store

n0 Format

Opcode	Operands		Instruction	Format	Function
halt			Halt the VMAX machine	n0	misc
nop			No operation	n0	misc

n04 Format

Opcode	Operands	Instruction	Format	Function
pushregs	bmsk	Push multiple registers	n04	stack
popregs	bmsk	Pop multiple registers	n04	stack

qc Format

Opcode	Operands		Instruction	Format	Function
set0l	la	c0	Store condition(0) in long	qc	flags
set1l	la	c1	Store condition(1) in long	qc	flags
set0w	wa	c0	Store condition(0) in word	qc	flags
set1w	wa	c1	Store condition(1) in word	qc	flags

qo Format

Opcode	Operands	Instruction	Format	Function
movflags	wv	Move word to flags reg	qo	flags
popd	da	Pop double	qo	stack
popf	fa	Pop float	qo	stack
popl	la	Pop long	qo	stack
popw	wa	Pop word	qo	stack

qo Format (continued)

Opcode	Operands	Instruction	Format	Function
pushd	dv	Push double	qo	stack
pushf	fv	Push float	qo	stack
pushl	lv	Push long	qo	stack
pushw	wv	Push word	qo	stack
stoflags	wa	Store flags reg into word	qo	flags

qr Format

Opcode	Operands		Instruction	Format	Function
absd	dv	dr	Absolute value of double	qr	otherarith
absf	fv	fr	Absolute value of float	qr	otherarith
absl	lv	lr	Absolute value of long	qr	otherarith
absw	wv	wr	Absolute value of word	qr	otherarith
addcl	lv	lr	Add long with carry	qr	add
addd	dv	dr	Add double	qr	add
addf	fv	fr	Add float	qr	add
addl	lv	lr	Add long	qr	add
addswl	wv	lr	Add signed word to long	qr	add
adduwl	wv	lr	Add unsigned word to long	qr	add
addw	wv	wr	Add word	qr	add
andl	lv	lr	And long	qr	logical
andw	wv	wr	And word	qr	logical
cmpd	dv	dr	Compare double	qr	compare
cmpf	fv	fr	Compare float	qr	compare
cmpl	lv	lr	Compare long	qr	compare
cmplb	bvl	lr	Compare lob(long) to byte	qr	compare
cmpw	wv	wr	Compare word	qr	compare
cmpwb	bvw	wr	Compare lob(word) to byte	qr	compare
cvtbsl	bvl	lr	Convert byte sign-ext to long	qr	convert
cvtbsw	bvw	wr	Convert byte sign-ext to word	qr	convert
cvtbzl	bvl	lr	Convert byte zero-ext to long	qr	convert

qr Format (continued)

Opcode	Operands		Instruction	Format	Function
cvtbzw	bvw	wr	Convert byte zero-ext to word	qr	convert
cvtdf	dv	fr	Convert double to float	qr	convert
cvtfd	fv	dr	Convert float to double	qr	convert
cvtsld	lv	dr	Convert signed long to double	qr	convert
cvtslf	lv	fr	Convert signed long to float	qr	convert
cvttdsl	dv	lr	Cnvrt trunc double to sgned lng	qr	convert
cvttdul	dv	lr	Cnvrt trunc doub to unsgned lng	qr	convert
cvttfsl	fv	lr	Cnvrt trunc float to signed lng	qr	convert
cvttful	fv	lr	Cnvrt trunc float to unsgnd lng	qr	convert
cvtuld	lv	dr	Convert unsigned long to double	qr	convert
cvtulf	lv	fr	Convert unsigned long to float	qr	convert
cvtwsl	wv	lr	Convert word sign-ext to long	qr	convert
cvtwzl	wv	lr	Convert word zero-ext to long	qr	convert
divd	dv	dr	Divide double	qr	divide
divf	fv	fr	Divide float	qr	divide
divrsl	lv	lr	Divide with rem signed long	qr	divide
divrslw	wv	lr	Div with rem sgned long by word	qr	divide
divrsw	wv	wr	Divide with rem signed word	qr	divide
divrul	lv	lr	Divide with rem unsigned long	qr	divide
divrulw	wv	lr	Div with rem unsgnd lng by word	qr	divide
divruw	wv	wr	Divide with rem unsigned word	qr	divide
divsl	lv	lr	Divide signed long	qr	divide
divsw	wv	wr	Divide signed word	qr	divide
divul	lv	lr	Divide unsigned long	qr	divide
divuw	wv	wr	Divide unsigned word	qr	divide
leal	lv	lr	Load effective address	qr	load
movbl	bvl	lr	Move byte to lob(long)	qr	move
movbw	bvw	wr	Move byte to lob(word)	qr	move
movd	dv	dr	Move double	qr	move
movf	fv	fr	Move float	qr	move
movl	lv	lr	Move long	qr	move
movw	wv	wr	Move word	qr	move
movwl	wv	lr	Move word to low(long)	qr	move
muld	dv	dr	Multiply double	qr	multiply
mulf	fv	fr	Multiply float	qr	multiply
mulsl	lv	lr	Multiply signed long	qr	multiply
mulsw	wv	wr	Multiply signed word	qr	multiply
mulswl	wv	lr	Multiply signed words -> long	qr	multiply
mulul	lv	lr	Multiply unsigned long	qr	multiply
muluw	wv	wr	Multiply unsigned word	qr	multiply

qr Format (continued)

Opcode	Operands		Instruction	Format	Function
muluwl	wv	lr	Multiply unsigned words -> long	qr	multiply
negd	dv	dr	Negate double	qr	subtract
negf	fv	fr	Negate float	qr	subtract
negl	lv	lr	Negate long	qr	subtract
negw	wv	wr	Negate word	qr	subtract
notl	lv	lr	Not long	qr	logical
notw	wv	wr	Not word	qr	logical
orl	lv	lr	Or long	qr	logical
orw	wv	wr	Or word	qr	logical
remsl	lv	lr	Remainder signed long	qr	divide
remsw	wv	wr	Remainder signed word	qr	divide
remul	lv	lr	Remainder unsigned long	qr	divide
remuw	wv	wr	Remainder unsigned word	qr	divide
rll	bvl	lr	Rotate left long	qr	shift
rlw	bvw	wr	Rotate left word	qr	shift
rrl	bvl	lr	Rotate right long	qr	shift
rrw	bvw	wr	Rotate right word	qr	shift
sll	bvl	lr	Shift left long	qr	shift
slw	bvw	wr	Shift left word	qr	shift
sqrtd	dv	dr	Square root of double	qr	otherarith
sqrtf	fv	fr	Square root of float	qr	otherarith
sral	bvl	lr	Shift right arithmetic long	qr	shift
sraw	bvw	wr	Shift right arithmetic word	qr	shift
srll	bvl	lr	Shift right logical long	qr	shift
srlw	bvw	wr	Shift right logical word	qr	shift
stod	da	dr	Store double	qr	store
stof	fa	fr	Store float	qr	store
stol	la	lr	Store long	qr	store
stolb	bal	lr	Store lob(long) into byte	qr	store
stolw	wa	lr	Store low(long) into word	qr	store
stow	wa	wr	Store word	qr	store
stowb	baw	wr	Store lob(word) into byte	qr	store
subcl	lv	lr	Subtract long with carry	qr	subtract
subd	dv	dr	Subtract double	qr	subtract
subf	fv	fr	Subtract float	qr	subtract
subl	lv	lr	Subtract long	qr	subtract
subswl	wv	lr	Subtract signed word from long	qr	subtract
subuwl	wv	lr	Subtract unsigned word from lng	qr	subtract

qr Format (continued)

Opcode	Operands		Instruction	Format	Function
subw	wv	wr	Subtract word	qr	subtract
xorl	lv	lr	Exclusive or long	qr	logical
xorw	wv	wr	Exclusive or word	qr	logical

APPENDIX B: Instructions Grouped by Function

This appendix contains a list of all VMAX instructions, organized into these functional groups:

data movement		
	move	memory-to-register moves
	store	register-to-memory moves
	load	load effective address
	flags	move and store flags register
arithmetic		
	add	add words, longs, ...
	subtract	subtract words, longs, ...
	multiply	multiply words, longs, ...
	divide	divide longs, ...
	other arith	abs, sqrt
shift		shift and rotate
logical		and, or, not, ...
convert		convert from one data type to another
compare		compare bytes, words, longs, ...
jump		jump and call
stack		pop, push, ...
misc		miscellaneous

Data Movement Instructions

Opcode	Operands		Instruction	Format	Function
gmov	ga	gr	General move	mr	move
movbl	bvl	lr	Move byte to lob(long)	qr	move
movbw	bvw	wr	Move byte to lob(word)	qr	move
movd	dv	dr	Move double	qr	move
movf	fv	fr	Move float	qr	move
movl	lv	lr	Move long	qr	move
movw	wv	wr	Move word	qr	move
movwl	wv	lr	Move word to low(long)	qr	move

Data Movement Instructions (continued)

Opcode	Operands		Instruction	Format	Function
gsto	ga	gr	General store	mr	store
stod	da	dr	Store double	qr	store
stof	fa	fr	Store float	qr	store
stol	la	lr	Store long	qr	store
stolb	bal	lr	Store lob(long) into byte	qr	store
stolw	wa	lr	Store low(long) into word	qr	store
stow	wa	wr	Store word	qr	store
stowb	baw	wr	Store lob(word) into byte	qr	store
leal	lv	lr	Load effective address	qr	load
movflags	wv		Move word to flags reg	qo	flags
stoflags	wa		Store flags reg into word	qo	flags
set0l	la	c0	Store condition(0) in long	qc	flags
set1l	la	c1	Store condition(1) in long	qc	flags
set0w	wa	c0	Store condition(0) in word	qc	flags
set1w	wa	c1	Store condition(1) in word	qc	flags

Arithmetic Instructions

Opcode	Operands		Instruction	Format	Function
addcl	lv	lr	Add long with carry	qr	add
addd	dv	dr	Add double	qr	add
addf	fv	fr	Add float	qr	add
addl	lv	lr	Add long	qr	add
addswl	wv	lr	Add signed word to long	qr	add
adduwl	wv	lr	Add unsigned word to long	qr	add
addw	wv	wr	Add word	qr	add
negd	dv	dr	Negate double	qr	subtract
negf	fv	fr	Negate float	qr	subtract
negl	lv	lr	Negate long	qr	subtract
negw	wv	wr	Negate word	qr	subtract

Arithmetic Instructions (continued)

Opcode	Operands		Instruction	Format	Function
subcl	lv	lr	Subtract long with carry	qr	subtract
subd	dv	dr	Subtract double	qr	subtract
subf	fv	fr	Subtract float	qr	subtract
subl	lv	lr	Subtract long	qr	subtract
subswl	wv	lr	Subtract signed word from long	qr	subtract
subuwl	wv	lr	Subtract unsigned word from lng	qr	subtract
subw	wv	wr	Subtract word	qr	subtract
muld	dv	dr	Multiply double	qr	multiply
mulf	fv	fr	Multiply float	qr	multiply
mulsl	lv	lr	Multiply signed long	qr	multiply
mulsw	wv	wr	Multiply signed word	qr	multiply
mulswl	wv	lr	Multiply signed words -> long	qr	multiply
mulul	lv	lr	Multiply unsigned long	qr	multiply
muluw	wv	wr	Multiply unsigned word	qr	multiply
muluwl	wv	lr	Multiply unsigned words -> long	qr	multiply
divd	dv	dr	Divide double	qr	divide
divf	fv	fr	Divide float	qr	divide
divrsl	lv	lr	Divide with rem signed long	qr	divide
divrslw	wv	lr	Div with rem sgned long by word	qr	divide
divrsw	wv	wr	Divide with rem signed word	qr	divide
divrul	lv	lr	Divide with rem unsigned long	qr	divide
divrulw	wv	lr	Div with rem unsgnd lng by word	qr	divide
divruw	wv	wr	Divide with rem unsigned word	qr	divide
divsl	lv	lr	Divide signed long	qr	divide
divsw	wv	wr	Divide signed word	qr	divide
divul	lv	lr	Divide unsigned long	qr	divide
divuw	wv	wr	Divide unsigned word	qr	divide
remsl	lv	lr	Remainder signed long	qr	divide
remsw	wv	wr	Remainder signed word	qr	divide
remul	lv	lr	Remainder unsigned long	qr	divide
remuw	wv	wr	Remainder unsigned word	qr	divide
absd	dv	dr	Absolute value of double	qr	otherarith
absf	fv	fr	Absolute value of float	qr	otherarith
absl	lv	lr	Absolute value of long	qr	otherarith
absw	wv	wr	Absolute value of word	qr	otherarith
sqrtd	dv	dr	Square root of double	qr	otherarith
sqrtf	fv	fr	Square root of float	qr	otherarith

Shift Instructions

Opcode	Operands		Instruction	Format	Function
rlil	sc	lr	Rotate left immediate long	ir	shift
rll	bvl	lr	Rotate left long	qr	shift
rril	sc	lr	Rotate right immediate long	ir	shift
rrl	bvl	lr	Rotate right long	qr	shift
rliw	sc	wr	Rotate left immediate word	ir	shift
rlw	bvw	wr	Rotate left word	qr	shift
rriw	sc	wr	Rotate right immediate word	ir	shift
rrw	bvw	wr	Rotate right word	qr	shift
slil	sc	lr	Shift left immediate long	ir	shift
sll	bvl	lr	Shift left long	qr	shift
srail	sc	lr	Shift right arithmetic imm long	ir	shift
sral	bvl	lr	Shift right arithmetic long	qr	shift
srlil	sc	lr	Shift right logical imm long	ir	shift
srll	bvl	lr	Shift right logical long	qr	shift
sliw	sc	wr	Shift left immediate word	ir	shift
slw	bvw	wr	Shift left word	qr	shift
sraiw	sc	wr	Shift right arithmetic imm word	ir	shift
sraw	bvw	wr	Shift right arithmetic word	qr	shift
srliw	sc	wr	Shift right logical imm word	ir	shift
srlw	bvw	wr	Shift right logical word	qr	shift

Logical Instructions

Opcode	Operands		Instruction	Format	Function
andl	lv	lr	And long	qr	logical
notl	lv	lr	Not long	qr	logical
orl	lv	lr	Or long	qr	logical
xorl	lv	lr	Exclusive or long	qr	logical
andw	wv	wr	And word	qr	logical
notw	wv	wr	Not word	qr	logical
orw	wv	wr	Or word	qr	logical
xorw	wv	wr	Exclusive or word	qr	logical

Conversion Instructions

Opcode	Operands		Instruction	Format	Function
cvtbzw	bvw	wr	Convert byte zero-ext to word	qr	convert
cvtbsw	bvw	wr	Convert byte sign-ext to word	qr	convert
cvtbzl	bvl	lr	Convert byte zero-ext to long	qr	convert
cvtbsl	bvl	lr	Convert byte sign-ext to long	qr	convert
cvtwzl	wv	lr	Convert word zero-ext to long	qr	convert
cvtwsl	wv	lr	Convert word sign-ext to long	qr	convert
cvtulf	lv	fr	Convert unsigned long to float	qr	convert
cvtslf	lv	fr	Convert signed long to float	qr	convert
cvtuld	lv	dr	Convert unsigned long to double	qr	convert
cvtsld	lv	dr	Convert signed long to double	qr	convert
cvttful	fv	lr	Cnvrt trunc float to unsgnd lng	qr	convert
cvttfsl	fv	lr	Cnvrt trunc float to signed lng	qr	convert
cvttdul	dv	lr	Cnvrt trunc doub to unsgned lng	qr	convert
cvttdsl	dv	lr	Cnvrt trunc double to sgned lng	qr	convert
cvtdf	dv	fr	Convert double to float	qr	convert
cvtfd	fv	dr	Convert float to double	qr	convert

Compare Instructions

Opcode	Operands		Instruction	Format	Function
cmpd	dv	dr	Compare double	qr	compare
cmpf	fv	fr	Compare float	qr	compare
cmpl	lv	lr	Compare long	qr	compare
cmplb	bvl	lr	Compare lob(long) to byte	qr	compare
cmpw	wv	wr	Compare word	qr	compare
cmpwb	bvw	wr	Compare lob(word) to byte	qr	compare

Jump Instructions

Opcode	Operands		Instruction	Format	Function
call	cc	ma	Call	ij	jump
callb	ma3		Call backward	a3	jump
callf	ma3		Call forward	a3	jump
jump	cc	ma	Jump	ij	jump
jumpb	ma3		Jump backward	a3	jump
jumpf	ma3		Jump forward	a3	jump
leave	stkc		Leave function	b1	jump
leaveres	stkc	bmsk	Leave func. and restore regs	b14	jump
ret	stkc		Return from call	b1	jump

Stack Instructions

Opcode	Operands		Instruction	Format	Function
enter	stkc		Enter function	b1	stack
entersav	stkc	bmsk	Enter func. and save regs	b14	stack
pushregs	bmsk		Push multiple registers	n04	stack
popregs	bmsk		Pop multiple registers	n04	stack
popd	da		Pop double	qo	stack
pushd	dv		Push double	qo	stack
popf	fa		Pop float	qo	stack
pushf	fv		Push float	qo	stack
popl	la		Pop long	qo	stack
pushl	lv		Push long	qo	stack
popw	wa		Pop word	qo	stack
pushw	wv		Push word	qo	stack

Miscellaneous Instructions

Opcode	Operands	Instruction	Format	Function
halt		Halt the VMAX machine	n0	misc
nop		No operation	n0	misc

APPENDIX C: Instructions Ordered Alphabetically by Opcode

Opcode	Operands		Instruction	Format	Function
absd	dv	dr	Absolute value of double	qr	otherarith
absf	fv	fr	Absolute value of float	qr	otherarith
absl	lv	lr	Absolute value of long	qr	otherarith
absw	wv	wr	Absolute value of word	qr	otherarith
addcl	lv	lr	Add long with carry	qr	add
addd	dv	dr	Add double	qr	add
addf	fv	fr	Add float	qr	add
addl	lv	lr	Add long	qr	add
addswl	wv	lr	Add signed word to long	qr	add
adduwl	wv	lr	Add unsigned word to long	qr	add
addw	wv	wr	Add word	qr	add
andl	lv	lr	And long	qr	logical
andw	wv	wr	And word	qr	logical
call	cc	ma	Call	ij	jump
callb	ma3		Call backward	a3	jump
callf	ma3		Call forward	a3	jump
cmpd	dv	dr	Compare double	qr	compare
cmpf	fv	fr	Compare float	qr	compare
cmpl	lv	lr	Compare long	qr	compare
cmplb	bvl	lr	Compare lob(long) to byte	qr	compare
cmpw	wv	wr	Compare word	qr	compare
cmpwb	bvw	wr	Compare lob(word) to byte	qr	compare
cvtbsl	bvl	lr	Convert byte sign-ext to long	qr	convert
cvtbsw	bvw	wr	Convert byte sign-ext to word	qr	convert
cvtbzl	bvl	lr	Convert byte zero-ext to long	qr	convert
cvtbzw	bvw	wr	Convert byte zero-ext to word	qr	convert
cvtdf	dv	fr	Convert double to float	qr	convert
cvtfd	fv	dr	Convert float to double	qr	convert
cvtsld	lv	dr	Convert signed long to double	qr	convert
cvtslf	lv	fr	Convert signed long to float	qr	convert
cvttdsl	dv	lr	Cnvrt trunc double to sgned lng	qr	convert
cvttdul	dv	lr	Cnvrt trunc doub to unsgned lng	qr	convert
cvttfsl	fv	lr	Cnvrt trunc float to signed lng	qr	convert
cvttful	fv	lr	Cnvrt trunc float to unsgnd lng	qr	convert
cvtuld	lv	dr	Convert unsigned long to double	qr	convert
cvtulf	lv	fr	Convert unsigned long to float	qr	convert
cvtwsl	wv	lr	Convert word sign-ext to long	qr	convert
cvtwzl	wv	lr	Convert word zero-ext to long	qr	convert
divd	dv	dr	Divide double	qr	divide
divf	fv	fr	Divide float	qr	divide
divrsl	lv	lr	Divide with rem signed long	qr	divide
divrslw	wv	lr	Div with rem sgned long by word	qr	divide
divrsw	wv	wr	Divide with rem signed word	qr	divide
divrul	lv	lr	Divide with rem unsigned long	qr	divide
divrulw	wv	lr	Div with rem unsgnd lng by word	qr	divide

Opcode	Operands		Instruction	Format	Function
divruw	wv	wr	Divide with rem unsigned word	qr	divide
divsl	lv	lr	Divide signed long	qr	divide
divsw	wv	wr	Divide signed word	qr	divide
divul	lv	lr	Divide unsigned long	qr	divide
divuw	wv	wr	Divide unsigned word	qr	divide
enter	stkc		Enter function	b1	stack
entersav	stkc	bmsk	Enter func. and save regs	b14	stack
gmov	ga	gr	General move	mr	move
gsto	ga	gr	General store	mr	store
halt			Halt the VMAX machine	n0	misc
jump	cc	ma	Jump	ij	jump
jumpb	ma3		Jump backward	a3	jump
jumpf	ma3		Jump forward	a3	jump
leal	lv	lr	Load effective address	qr	load
leave	stkc		Leave function	b1	jump
leaveres	stkc	bmsk	Leave func. and restore regs	b14	jump
movbl	bvl	lr	Move byte to lob(long)	qr	move
movbw	bvw	wr	Move byte to lob(word)	qr	move
movd	dv	dr	Move double	qr	move
movf	fv	fr	Move float	qr	move
movflags	wv		Move word to flags reg	qo	flags
movl	lv	lr	Move long	qr	move
movw	wv	wr	Move word	qr	move
movwl	wv	lr	Move word to low(long)	qr	move
muld	dv	dr	Multiply double	qr	multiply
mulf	fv	fr	Multiply float	qr	multiply
mulsl	lv	lr	Multiply signed long	qr	multiply
mulsw	wv	wr	Multiply signed word	qr	multiply
mulswl	wv	lr	Multiply signed words -> long	qr	multiply
mulul	lv	lr	Multiply unsigned long	qr	multiply
muluw	wv	wr	Multiply unsigned word	qr	multiply
muluwl	wv	lr	Multiply unsigned words -> long	qr	multiply
negd	dv	dr	Negate double	qr	subtract
negf	fv	fr	Negate float	qr	subtract
negl	lv	lr	Negate long	qr	subtract
negw	wv	wr	Negate word	qr	subtract
nop			No operation	n0	misc
notl	lv	lr	Not long	qr	logical
notw	wv	wr	Not word	qr	logical

Opcode	Operands		Instruction	Format	Function
orl	lv	lr	Or long	qr	logical
orw	wv	wr	Or word	qr	logical
popd	da		Pop double	qo	stack
popf	fa		Pop float	qo	stack
popl	la		Pop long	qo	stack
popregs	bmsk		Pop multiple registers	n04	stack
popw	wa		Pop word	qo	stack
pushd	dv		Push double	qo	stack
pushf	fv		Push float	qo	stack
pushl	lv		Push long	qo	stack
pushregs	bmsk		Push multiple registers	n04	stack
pushw	wv		Push word	qo	stack
remsl	lv	lr	Remainder signed long	qr	divide
remsw	wv	wr	Remainder signed word	qr	divide
remul	lv	lr	Remainder unsigned long	qr	divide
remuw	wv	wr	Remainder unsigned word	qr	divide
ret	stkc		Return from call	b1	jump
rlil	sc	lr	Rotate left immediate long	ir	shift
rliw	sc	wr	Rotate left immediate word	ir	shift
rll	bvl	lr	Rotate left long	qr	shift
rlw	bvw	wr	Rotate left word	qr	shift
rril	sc	lr	Rotate right immediate long	ir	shift
rriw	sc	wr	Rotate right immediate word	ir	shift
rrl	bvl	lr	Rotate right long	qr	shift
rrw	bvw	wr	Rotate right word	qr	shift
set0l	la	c0	Store condition(0) in long	qc	flags
set0w	wa	c0	Store condition(0) in word	qc	flags
set1l	la	c1	Store condition(1) in long	qc	flags
set1w	wa	c1	Store condition(1) in word	qc	flags
slil	sc	lr	Shift left immediate long	ir	shift
sliw	sc	wr	Shift left immediate word	ir	shift
sll	bvl	lr	Shift left long	qr	shift
slw	bvw	wr	Shift left word	qr	shift
sqrtd	dv	dr	Square root of double	qr	otherarith
sqrtf	fv	fr	Square root of float	qr	otherarith
srail	sc	lr	Shift right arithmetic imm long	ir	shift
sraiw	sc	wr	Shift right arithmetic imm word	ir	shift
sral	bvl	lr	Shift right arithmetic long	qr	shift
sraw	bvw	wr	Shift right arithmetic word	qr	shift
srlil	sc	lr	Shift right logical imm long	ir	shift
srliw	sc	wr	Shift right logical imm word	ir	shift
srll	bvl	lr	Shift right logical long	qr	shift
srlw	bvw	wr	Shift right logical word	qr	shift

Opcode	Operands		Instruction	Format	Function
stod	da	dr	Store double	qr	store
stof	fa	fr	Store float	qr	store
stoflags	wa		Store flags reg into word	qo	flags
stol	la	lr	Store long	qr	store
stolb	bal	lr	Store lob(long) into byte	qr	store
stolw	wa	lr	Store low(long) into word	qr	store
stow	wa	wr	Store word	qr	store
stowb	baw	wr	Store lob(word) into byte	qr	store
subcl	lv	lr	Subtract long with carry	qr	subtract
subd	dv	dr	Subtract double	qr	subtract
subf	fv	fr	Subtract float	qr	subtract
subl	lv	lr	Subtract long	qr	subtract
subswl	wv	lr	Subtract signed word from long	qr	subtract
subuwl	wv	lr	Subtract unsigned word from lng	qr	subtract
subw	wv	wr	Subtract word	qr	subtract
xorl	lv	lr	Exclusive or long	qr	logical
xorw	wv	wr	Exclusive or word	qr	logical

APPENDIX D: Differences Between VMAX v1.00 and v2.00

Version History

Version 1.00: 1990 April 26

Version 2.00: 1990 July 16

Document Changes from Version 1.00 to Version 2.00

The following sections are new or have been changed:

VMAX Memory and Address Space	CHANGED
VMAX Registers	CHANGED
Addressing Modes	NEW
Overview of Formats	EXPANDED
The qr Format	CHANGED
The mr Format	NEW
The b14 Format	NEW
The n04 Format	NEW
Instruction Set Summary by Function	NEW
Appendices D, E, and F	NEW
Instruction Descriptions	NEW INSTRUCTIONS

Each of the major changes is described in the following.

VMAX Memory and Address Space (CHANGED)

VMAX memory is now mapped directly onto the PCMAX2 Vram. The VMAX stack grows from high addresses to low addresses, so the organization of Code Space, Data Space, Heap, and Stack is now reversed in VMAX memory.

VMAX Registers (CHANGED)

There are now 8 of each type of register, and no registers overlap. For the time being, VMAX registers will be stored in the PCMAX2 DataRam rather than in PCMAX2 registers.

The flags register has been changed so that instead of the traditional overflow flag, carry flag, zero flag, and sign flag, the following flags are used: less than unsigned flag, less than signed flag, equal flag, greater than signed flag, and greater than unsigned flag. Only compare instructions change the flag bits, so the VMAX interpreter need not deal with flags for every arithmetic instruction.

Addressing Modes (NEW)

Most of the original addressing modes have been omitted and replaced with based, indexed, and based-indexed modes. The indexed modes allow a scale factor of 1, 2, 4, or 8 to be used. Displacements of various lengths are allowed with all the new addressing modes.

Overview of Formats (EXPANDED)

This section of the document now includes a brief discussion of each instruction format.

The qr Format (CHANGED)

The qr format was changed to allow for the new based, indexed, and based-indexed addressing modes.

The mr, b14, and n04 Formats (NEW)

The mr format allows 1, 2, 4, or 8 bytes to be moved from memory to any register type (and vice versa), as well as from any register type to any register type. Thus, for example, a double register can be moved to 4 contiguous word registers. In version 1.00 this could only be done by storing the double register into memory, and then moving it in pieces to word registers, using 4 move word instructions. -- All the addressing modes of the qr format are available in the mr format except for immediate operands.

The b14 and n04 are simple extensions of the b1 and n0 formats, used by the new instructions entersav, leaveres, pushregs, and popregs.

Instruction Set Summary by Function (NEW)

This section describes groups of instructions by the functions they perform. Several subsections contain important information not found elsewhere, e.g., the subsections on divide, convert, and compare instructions.

Appendices D, E, and F (NEW)

These appendices include the one you are now reading, a discussion of possible future directions for VMAX, and a set of diagrams for all the instruction formats.

Instruction Descriptions

The opcode mnemonics for the following instructions have been changed:

new name	instruction	old name
cmplb	Compare loworder byte of long to byte	cmpbl
cmpwb	Compare loworder byte of word to byte	cmpbw
cvtsld	Convert signed long to double	cvtld
cvttdsl	Convert truncated double to signed long	cvtdl
divrsl	Divide with rem signed long	divsl
divrsw	Divide with rem signed word	divsw
divrul	Divide with rem unsigned long	divul
divruw	Divide with rem unsigned word	divuw
stolb	Store loworder byte of long into byte	stobl
stolw	Store loworder word of long into word	stowl
stowb	Store loworder byte of word into byte	stobw

The following instructions are new:

new name	instruction
addcl	Add long with carry
addswl	Add signed word to long
adduwl	Add unsigned word to long
cvtslf	Convert signed long to float
cvttdul	Convert truncated double to unsigned long
cvttfsl	Convert truncated float to signed long
cvttful	Convert truncated float to unsigned long
cvtuld	Convert unsigned long to double
cvtulf	Convert unsigned long to float
divrslw	Divide with rem signed long by word
divrulw	Divide with rem unsigned long by word
divsl	Divide signed long
divsw	Divide signed word
divul	Divide unsigned long
divuw	Divide unsigned word
entersav	Enter function and save registers
gmov	General move
gsto	General store
leal	Load effective address into long register
leaveres	Leave function and restore registers
movflags	Move word to flags register
mulswl	Multiply signed words yielding long
muluwl	Multiply unsigned words yielding long
popregs	Pop multiple registers
pushregs	Push multiple registers
remsl	Remainder signed long
remsw	Remainder signed word
remul	Remainder unsigned long
remuw	Remainder unsigned word
stoflags	Store flags register into word
subcl	Subtract long with carry
subswl	Subtract signed word from long
subuwl	Subtract unsigned word from long

The following paragraphs briefly discuss the rationale behind these new instructions:

addcl and subcl instructions allow for multiple-precision arithmetic.

addswl, adduwl, divrslw, divrulw, mulswl, muluwl, subswl, and subuwl allow arithmetic on operands of different sizes, namely words and longs.

cvtslf, and the the other new conversion instructions make it possible to convert floating to unsigned integers as well as to signed integers. Also, these new conversions operations have been carefully chosen to satisfy all of GCC's requirements.

divsl, and other new division and remainder instructions separate the calculation of quotients and remainders, so just the one desired in a given instance need be computed. This also conforms to what GCC wants.

entersav and leaveres make it possible to code C function prologues and epilogues with fewer bytes.

gmov and gsto allow greater freedom of data movement when data type is not a consideration. Also, these instructions are more or less required by GCC.

leal allows address calculations to be shorter and faster.

movflags and stoflags allow the flags register to be read, changed, and written in a reasonable easy manner.

popregs and pushregs allow any number of registers to be saved and restored with a single instruction.

APPENDIX E: Ideas and Notes for Future Versions

Ultimately we will probably want two versions of the VMAX interpreter, one which runs as fast as possible, and one which does as much error checking as possible. The latter version should include stack checking, as discussed in the next section.

Stack Checking

A nagging problem with hardware stacks on machines like the 8086 is that there is no hardware checking of stack overflow or underflow. This can be especially irksome when the heap and the stack compete for memory: When one grows into the other the results are usually disasterous.

Thus, we define two new VMAX registers, spmin and spmax. These are 32-bit registers containing unsigned long values which define the extent of the stack. Each time sp is changed (either explicitly or implicitly via push, pop, call, ret, etc.), this check is made:

```
spmin <= sp <= spmax
```

If sp is out of bounds, a trap occurs. (Exactly what it means for a trap to occur on VMAX will not be dealt with at this time.)

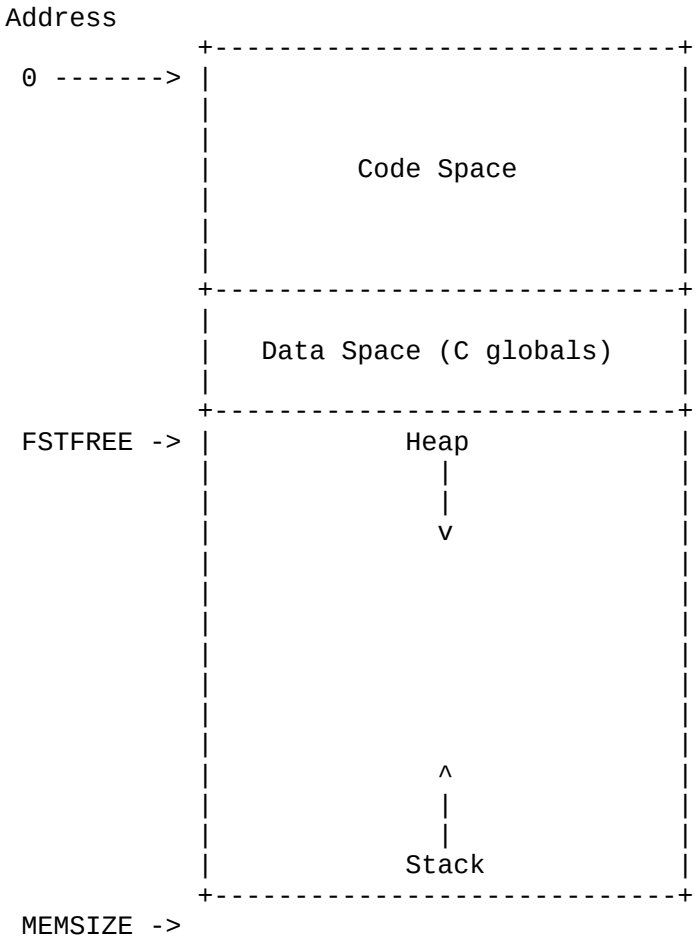
The following new qo-format instructions are used to operate on spmin and spmax:

```
movspmin  lv          Move q-operand to spmin
movspmax  lv          Move q-operand to spmax
stospmin  la          Store spmin in q-operand
stospmax  la          Store spmax in q-operand
```

Note that movspmin, movspmax, stospmin, and stospmax do no checking at all. These instructions simply move values into and out of spmin and spmax. All checking is done when sp changes or is operated on.

[Are stospmin and stospmax really needed?]

To illustrate how these instructions are used, consider the following memory map for a compiled C program:



Before the loader transfers control to the loaded program, it executes these instructions:

```
movspmin  FSTFREE              ; spmin = FSTFREE
movspmax  MEMSIZE-4            ; spmax = MEMSIZE-4
movl      MEMSIZE-4, sp        ; sp = MEMSIZE-4
```

Thus, as the program executes, sp is guaranteed to always lie in the interval [FSTFREE, MEMSIZE-4].

Whenever a malloc call needs to increase the size of the heap, the following code is executed, where HeapTop is the address of the first free byte after the top of the heap, and NewHeapTop is the new value we wish to assign to HeapTop:

```
if (NewHeapTop < sp)
        spmin = HeapTop = NewHeapTop;
else
        out of memory (or need to garbage collect)
```

Thus, spmin always contains the address of the top of the heap, so that the stack cannot overflow into the heap without a trap. (The variable HeapTop is not really needed since its value is always equal to the contents of the spmin register.)

In some instances strict stack checking can get in the way, so we have a new instruction to turn stack checking on and off:

```
stkchk  0/1             Turn stack checking on or off
```

This is a b1-format instruction whose operand is either 0 (off) or 1 (on).

[NOTE: Perhaps there should be a bit in the flags register which tells if stack checking is on or off? If so, then we probably don't want a stkchk instruction, because stoflags and movflags can be used to turn the bit on or off.]

Summary of Stack Checking Instructions

format	opcode	operands	instruction
qo	movspmin	lv	Move address to spmin register
qo	movspmax	lv	Move address to spmax register
qo	stospmin	la	Store spmin register
qo	stospmax	la	Store spmax register
b1	stkchk	0/1	Turn stack checking off/on

Loose Ends

Following is a list of loose ends:

1. At present, numeric values have not been assigned to VMAX opcodes. This can be done at anytime; it is a question of what works best for the VMAX interpreter.

2. No instructions have been defined for I/O, in particular for communication with the host PC. This is an important area which will be dealt with later.

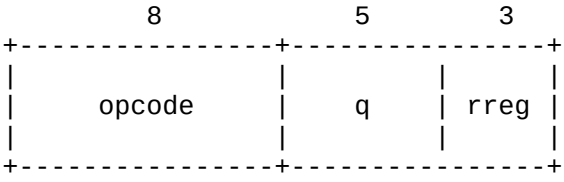
3. Thought needs to be given to how a C program running on the PCMAX2 accesses XMEM, the extended memory available to both the PC and PCMAX2.

4. The effect of each instruction on the flags register is not yet defined and documented in the instruction descriptions. At present only compare instructions change the flags register, but (as discussed elsewhere in this document), at some point we need to decide what to do about arithmetic overflow.

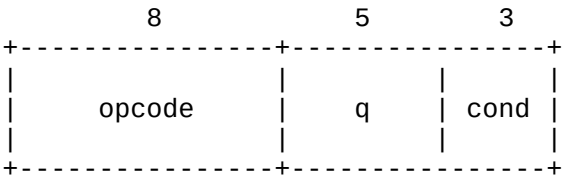
APPENDIX F: Diagrams of Instruction Formats

Overview of Formats

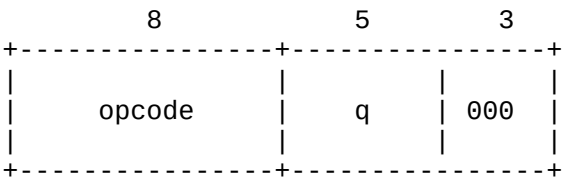
qr format (2, 4, 6, 8, or 10 bytes)



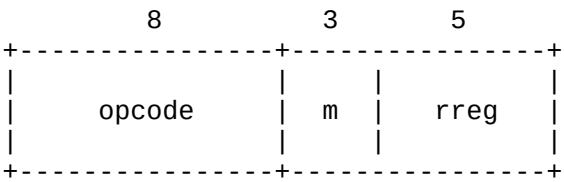
qc format (2, 4, 6, 8, or 10 bytes)



qo format (2, 4, 6, 8, or 10 bytes)



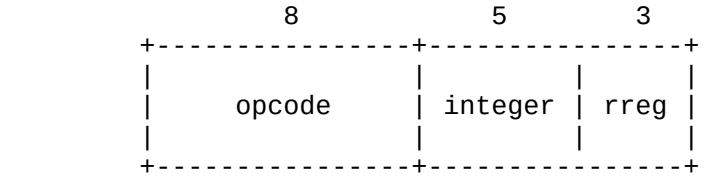
mr format (4, 6, or 8 bytes)



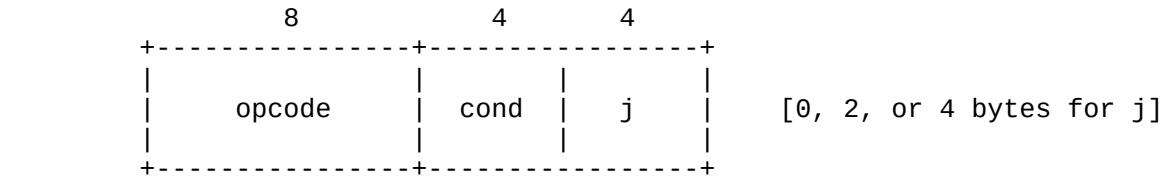
APPENDIX F: Diagrams of Instruction Formats (continued)

Overview of Formats (continued)

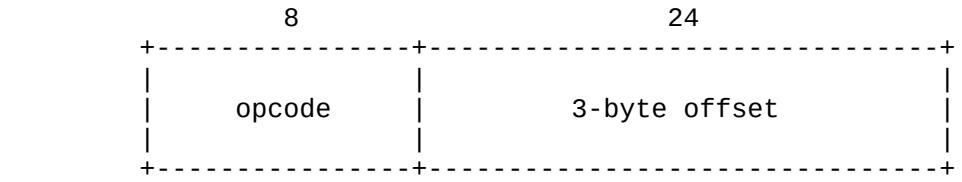
ir format (2 bytes)



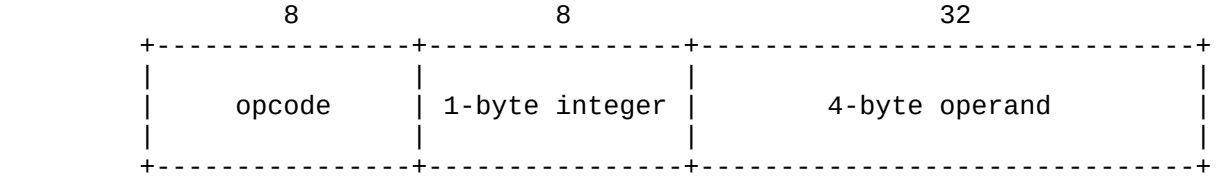
ij format (2, 4, or 6 bytes)



a3 format (4 bytes)



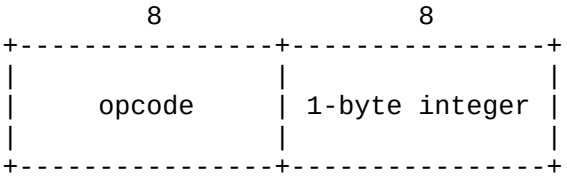
b14 format (6 bytes)



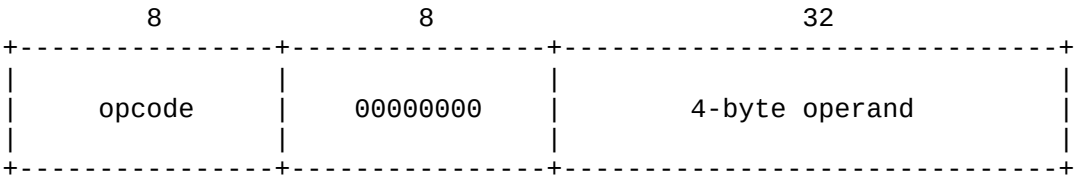
APPENDIX F: Diagrams of Instruction Formats (continued)

Overview of Formats (continued)

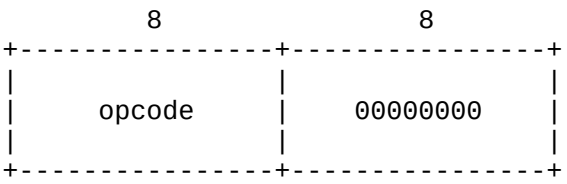
b1 format (2 bytes)



n04 format (6 bytes)



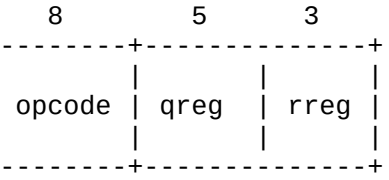
n0 format (2 bytes)



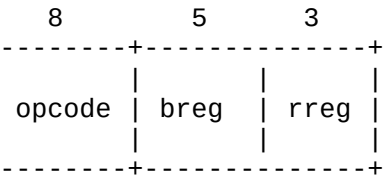
APPENDIX F: Diagrams of Instruction Formats (continued)

The qr Format: Basic q-Operands

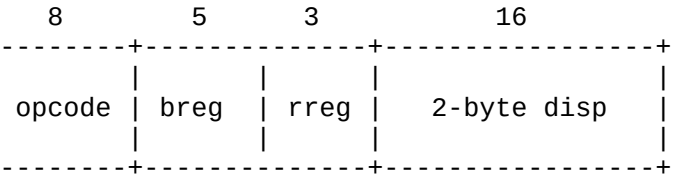
reg (2 bytes)



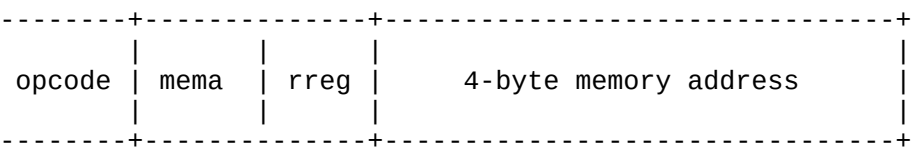
b (2 bytes)



bd2 (4 bytes)



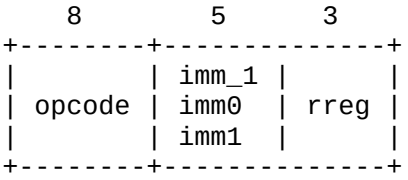
mema (6 bytes)



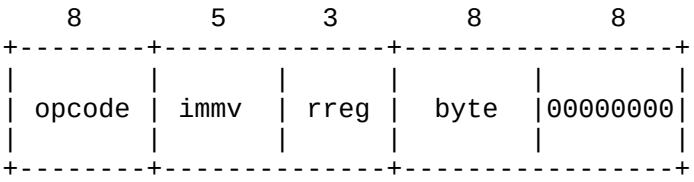
APPENDIX F: Diagrams of Instruction Formats (continued)

The qr Format: Immediate q-Operands

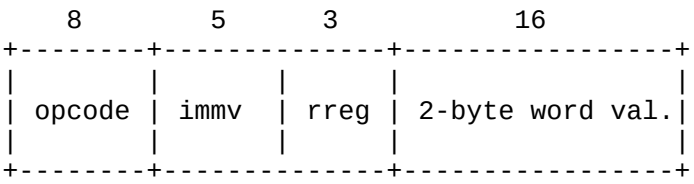
imm_1, imm0, imm1 (2 bytes)



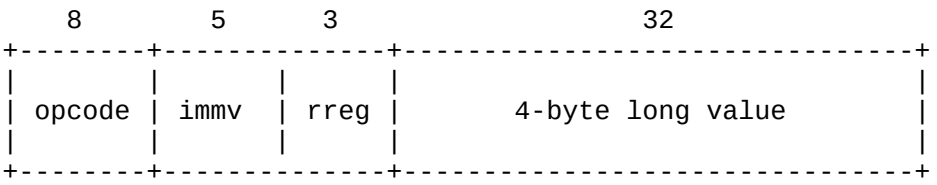
immv byte (4 bytes)



immv word (4 bytes)



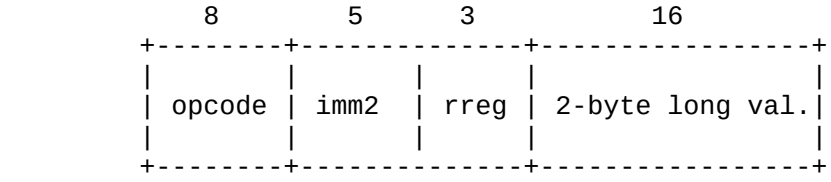
immv long (6 bytes)



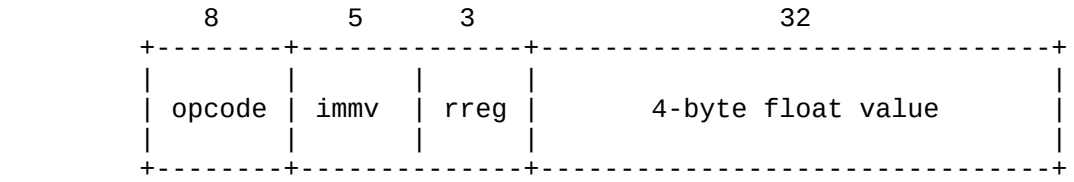
APPENDIX F: Diagrams of Instruction Formats (continued)

The qr Format: Immediate q-Operands (continued)

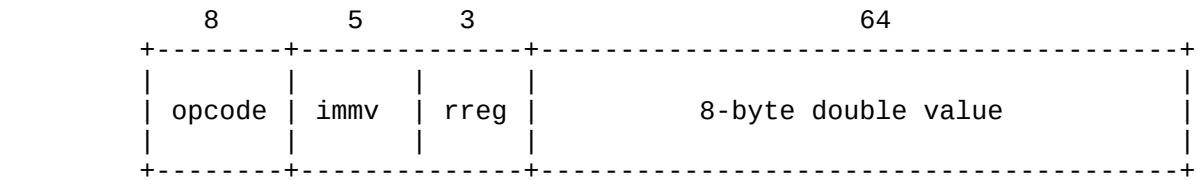
imm2 long (4 bytes)



immv float (6 bytes)



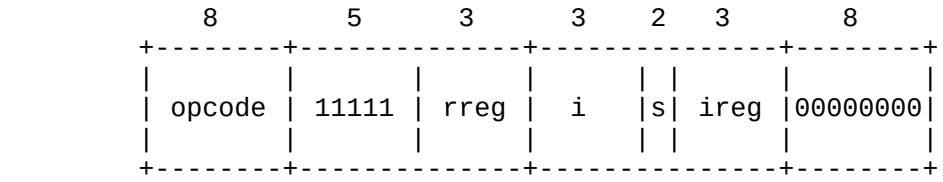
immv double (10 bytes)



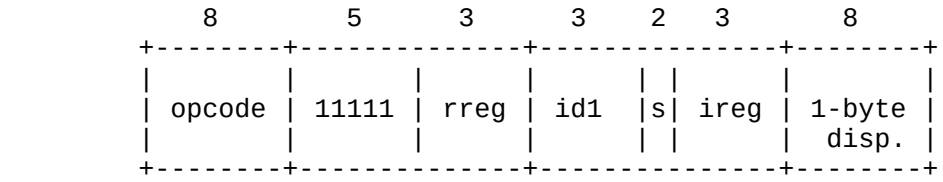
APPENDIX F: Diagrams of Instruction Formats (continued)

The qr Format: Indexed q-Operands

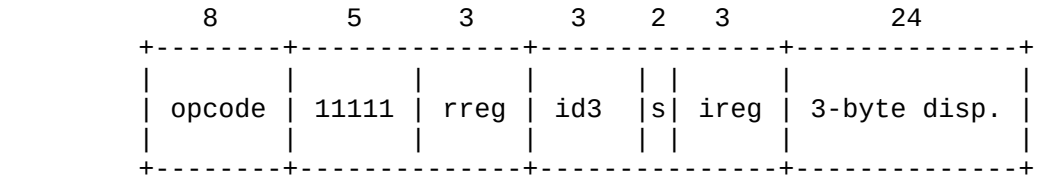
i (4 bytes)



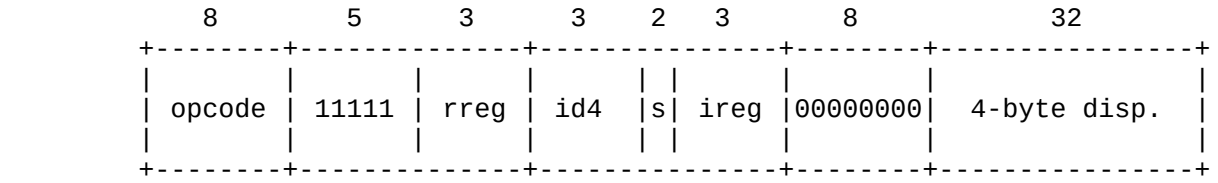
id1 (4 bytes)



id3 (6 bytes)



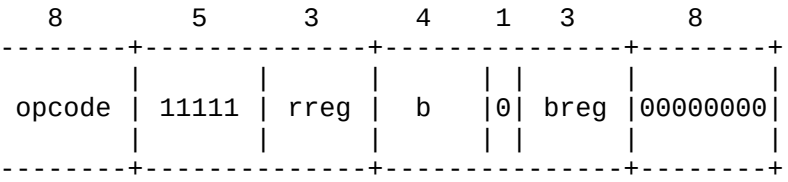
id4 (8 bytes)



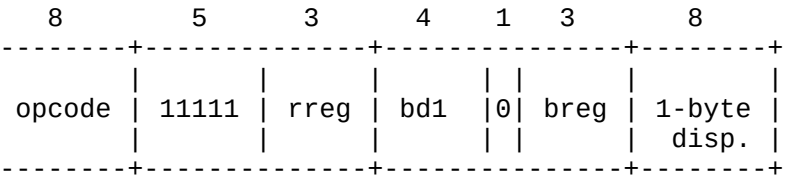
APPENDIX F: Diagrams of Instruction Formats (continued)

The qr Format: Based q-Operands

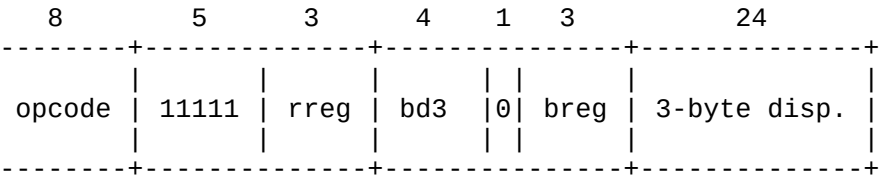
b (4 bytes)



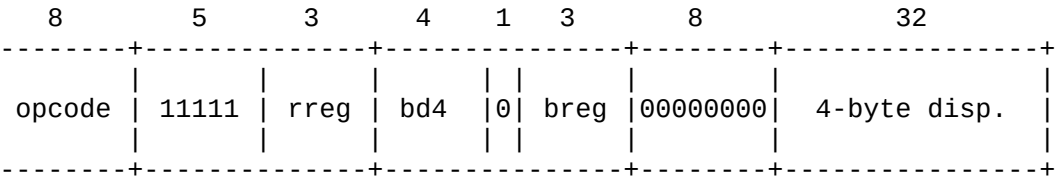
bd1 (4 bytes)



bd3 (6 bytes)



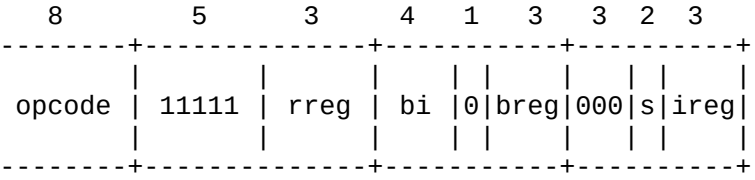
bd4 (8 bytes)



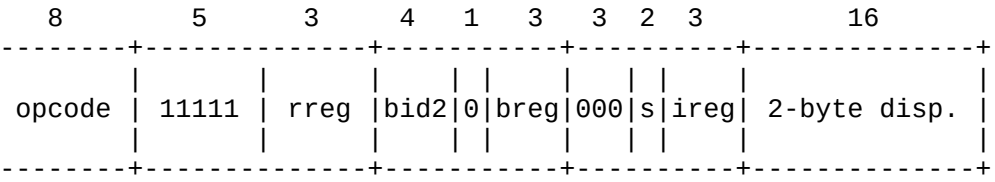
APPENDIX F: Diagrams of Instruction Formats (continued)

The qr Format: Based-Indexed q-Operands

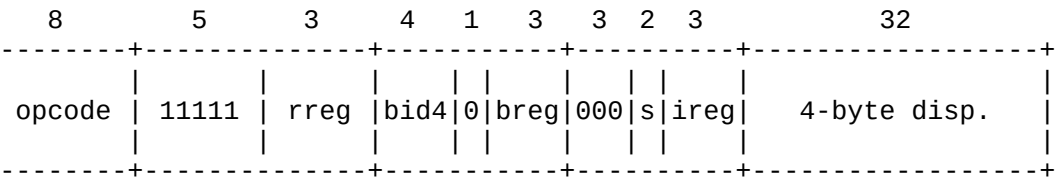
bi (4 bytes)



bid2 (6 bytes)

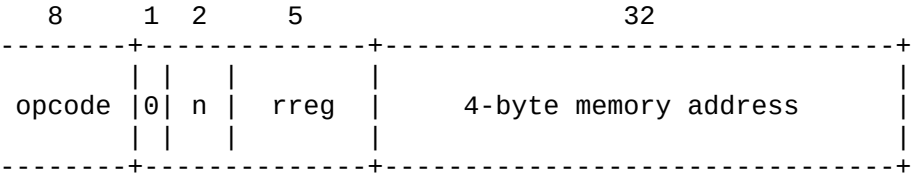


bid4 (8 bytes)

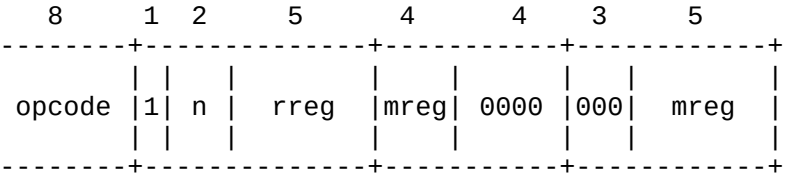


The mr Format

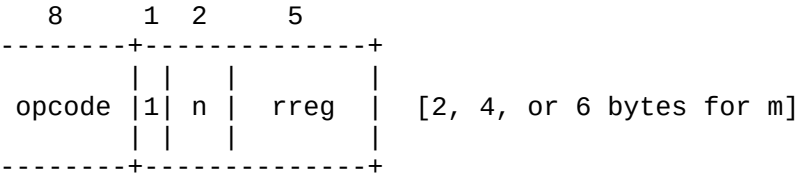
mmema (6 bytes)



mreg (4 bytes)



i, b, bi (4, 6, or 8 bytes)

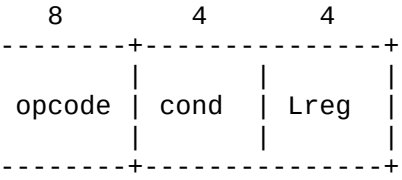


The m-operand of an mr instruction can be specified using any of the indexed, based, or based-indexed addressing modes used to specify q-operands.

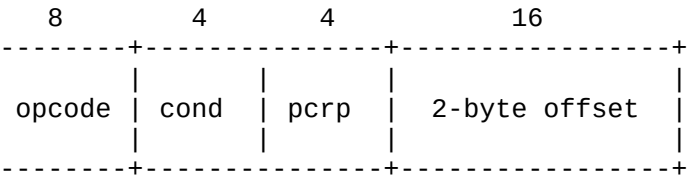
APPENDIX F: Diagrams of Instruction Formats (continued)

The ij Format

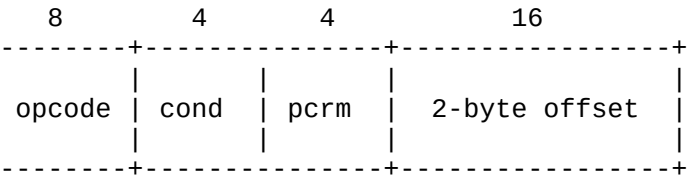
jregi (2 bytes)



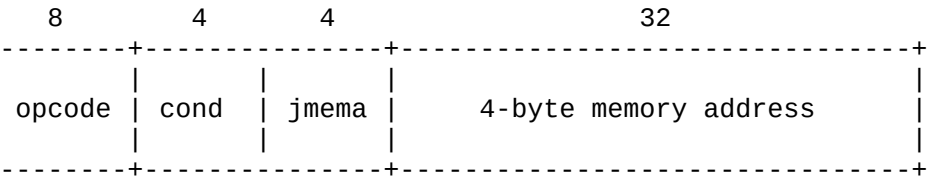
pcrp (4 bytes)



pcrm (4 bytes)



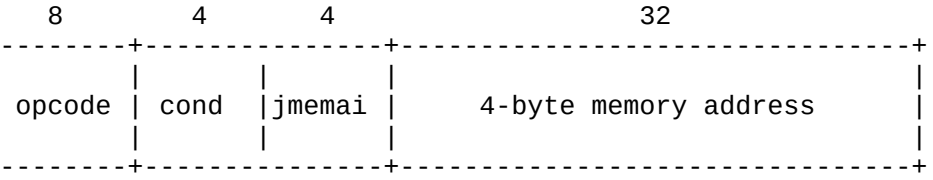
jmema (6 bytes)



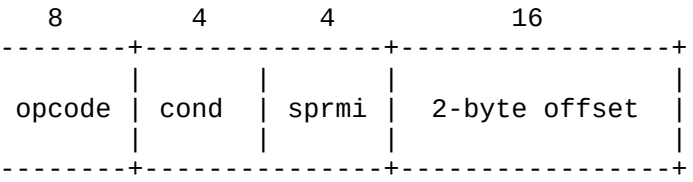
APPENDIX F: Diagrams of Instruction Formats (continued)

The ij Format (continued)

jmemai (6 bytes)



sprmi (4 bytes)



<END DOCUMENT>