Internal Macroinstruction Procedures (IMPs) Reference Manual

Version 5.2

Bob Lordi

Thinking Machines Corporation
245 First St., Cambridge MA 02142 USA

ABSTRACT

This is the Reference Manual for using the IMP (Internal Macroinstruction Procedure) mechanism. It describes the IMP mechanism, and gives a detailed specification for the macroinstructions which are provided with it.

An overview is presented in section 1.

Internal software versions: IMPS-F5202, IMPS-UC-F5203, CMIS-UC-F5204

September 22, 1989

Internal Macroinstruction Procedures (IMPs) Reference Manual

Version 5.2

Bob Lordi

Thinking Machines Corporation
245 First St., Cambridge MA 02142 USA

Internal users note: A machine-readable copy of this document can be found in /cm/cmis/doc/imps-ref-man.out You can get a typeset copy with the UNIX command:

```
/cm/cmis/doc/pdoc -Plaserprinter /cm/imps/doc/imps-ref-man.ms
```

Table of Contents

1. Overview of the IMP Architecture

The IMP mechanism provides the ability to execute sequences of CM2 macroinstructions internally in the CM2 sequencer. Internal sequencing, by eliminating host/CM2 communication overhead, allows the macroinstruction set to be more atomic, i.e. to consists of lower-level primitives from which larger operations can be constructed.

The IMP mechanism provides a set of tools which allow the construction of sequences of macroinstructions called "Internal Macroinstruction Procedures", or **IMPs**. These tools define a programming model wherein IMPs can manipulate sequencer registers, maintain a stack, perform simple branching and looping, and, importantly, control the movement and computation of data within the CM2.

Specifically, the tools allow you program IMPs in a symbolic source language, to download the IMPs into the memory of the CM2 sequencer, and to "execute" them at will from the front end. Input data or parameters to an IMP may be transmitted from the front end to the IMP as it executes, and result data can be sent by the IMP to the front end.

1.1. IMP Format - Steps

An IMP is a sequence of macroinstruction invocations or *steps* stored in the CM2 scratch memory. The first memory word of a step is the call address of the macroinstruction and the remaining words contain optional arguments. The formatting and encoding of argument data is specific to the macroinstruction.

```
STEP N
 argument 1 for step N
 argument 2 for step N
 ...
STEP N+1
 argument for step N+1
 ...
```

Note: The actual stored format of an IMP varies from the above in several ways. You need not concern yourself with these details, but you may consult Appendix C for a discussion of the internal format of an IMP as loaded in SRAM, if your are interested.

Once an IMP has been loaded into the scratch memory of the CM2 microcontroller it can be executed by a microcode routine called the IMP Executor. The IMP Executor executes the IMP by calling the specified macroinstructions in the programmed order, passing the given arguments to the macroinstruction.

1.2. Control Macroinstructions

Supplied with the IMP mechanism is a set of built-in macroinstructions which provide a the basic controls over the microsequencer itself, and the execution of IMPs. These macroinstructions perform these basic functions:

1. Manipulation of the uController registers. This includes some useful specialized functions for setting up arguments in the correct registers before invoking macroinstructions,
2. Basic looping and branching functions to support non-sequential IMP sequencing, and
3. Special functions to support Vector Operations (multiple per-processor calculations, such as is used for VP looping).

1.3. Registers

As a part of the IMP programming model, the registers of the CM2 sequencer have been divided into functional groups. The registers of each group are used in certain predefined ways. For example, the looping functions require the use of one of four "counter registers".

1.4. The IMP Executor

IMPs are executed from the front-end by the function EXECUTE-IMP-ID, which takes an IMP Identifier or "IMP ID" as its argument. This function loads the IMP automatically into the CM2 (if not already loaded), and executes it once.

Macroinstructions are provided for passing data from the front-end to the uRegisters through the IFIFO, thus IMPs can be parameterized from the front-end. Similarly, result data can be passed by the IMP back to the front end through the OFIFO.

1.5. IMP Programming

IMPS are coded using a syntax specific to the application language. Currently, the programming and execution of IMPs from Common LISP and from C/UNIX is supported.

Under Common LISP (either from Symbolics LISP Machines or under SUN/LUCID LISP), IMPs are coded as LISP forms (lists), and processed by the function OPEN-IMP, which assembles the IMP, and makes it ready for subsequent loading and execution.

As an example, the following IMP, coded in LISP, transfers a 32-bit value from the Input FIFO to the register named %P1, and adds 3 to it:

```
(IMP-RECEIVE-%P1)
(IMP-ADD-CONST-TO-%P1
   3)                                    ;note use of argument
(IMP-EXIT)
```

Under C/UNIX, IMPs are programmed in files and translated (assembled) by the IMP Assembler (IMPASS). IMPASS takes symbolic macroinstruction calls and their arguments and generates the vector of numbers that forms the body of the IMP. The assembler also provides some syntactic help in using the macroinstructions, e.g. labels, relative branch calculations, and symbolic names for special purpose uRegisters. Lexical help is also provided, including conditional assembly and macros. The previous example coded for IMPASS under UNIX is as follows:

```
IMP_RECEIVE_%P1                          ;note the use of underscores
IMP_ADD_CONST_TO_%P1
   3
IMP_EXIT
```

Two manuals are provided which describe the IMP tools. Common LISP users should consult the "*Common LISP IMP Tools Reference Manual*". C/UNIX users should consult the *UNIX and VAX/VMS IMP Tools Reference Manual*". VAX/VMS is not currently supported, but will be in the near future, using the same tool set as C/UNIX.

1.6. CMIS

The IMP mechanism is specifically designed to support CMIS. CMIS is a set of macroinstructions, callable from IMPs, which manipulate the state of the Connection Machine hardware, i.e. the bit serial processors, the SPRINT and Floating Point Units, and the communication system.

In keeping with the IMPs philosophy, CMIS is a set of low-level atomic operations from which larger algorithms can be constructed. CMIS macroinstructions are typically the fundamental microcode sequences - pipelines and basic data transfer operations - which comprise the basic architectural repertoire of the hardware.

Architecturally, CMIS is considered to be layered on top of the IMP mechanism, and as such is documented and maintained separately. Consult the document "*CMIS Reference Manual*" for details on the CMIS instruction set.

1.7. Relationship with Existing Microcode

The IMP mechanism comfortably coexists with all the existing microcode, including PARIS, the CM Math Library, and so forth. The current dispatch table and code is used without modification in this scheme, and is used for invoking the IMP Executor from the front end host. Front end calls to execute IMPs and to execute PARIS instructions can be freely intermixed, providing the IMPs are programmed so as not to disturb machine state required by PARIS.

You should consult Appendix D for details on coding IMPs that can be executed along with PARIS instructions.

Furthermore, the IMP mechanism provides a method for calling existing front-end-callable microcode routines (e.g. PARIS instruction microcode) from within IMPs. Some specialized knowledge of the CM2 microcode is required in order to effectively use this feature, and it should be generally avoided.

2. A Note on Meta-Syntax

In this document and all IMP/CMIS documents, the following meta-syntax is used to describe syntactic structure:

{a,b,c,...}	A choice of exactly one of a, b, or c
{i-j}	One value in range i through j (usually integers)
[a,b,c]	A choice of a, b, c, or nothing
[i-j]	One value in range i through j, or nothing .

This syntax can be arbitrarily nested, as in "{A,B,P[1-4]}". At the topmost level, the full syntax is taken to represent the set of all possible expansions of the meta-syntax. For example, the syntax

CMIS-FRTB-%P{1-4}[-DYN]

specifies the 8 CMIS instructions CMIS-FRTB-%P*n* and CMIS-FRTB-%P*n*-DYN where *n* is 1, 2, 3, or 4.

3. The IMP Executor

An IMP is executed by invoking an IMP Executor. When invoked, the IMP Executor sets an internal *step counter* to point to the first step of the IMP (step 0), does a uCode call to the macroinstruction it points to. Upon return, it does the same for step 1, and so forth. Some of the control macroinstructions affect the state of the IMP executor to effect loops, jumps, etc.

In all programming environments, IMPs are executed by calling the function **execute-imp-id**, which takes an IMP "id" as its argument, and which causes the specified IMP to execute. Execution completes when the IMP executes the IMP-EXIT macroinstruction (described later). You should consult the IMPS Tools manual specific to your programming environment for instructions on calling the IMP Executor.

IMPs can also invoke the macroinstruction IMP-CALL-IMP, which allows one IMP to execute (in effect, call) another IMP. Such calls may be arbitrarily nested, and sufficient stack primitives are provided for creating lexical scope.

4. Register Model

Specific (optional) allocation of the microcontroller registers supports the IMP model. Generally, allocations are done to limit certain functions to a small set of registers for the sake of saving control store space. If a function which implies the use of certain registers is not used, those registers can be used instead for any purpose.

The nomenclature %N is used to indicate microcontroller register N. The IMP symbolic names for uRegisters are of the form "%name". Appendix A includes a table of the allocation of the 32 uRegisters.

Twelve *argument registers*, %A1 through %A12, are used as general registers. Instructions are provided for moving data between these registers and most registers, performing some simple arithmetic, and sending and receiving data between these registers and the front end.

Four *pointer registers*, %P1 through %P4 hold memory addresses of fields in CM memory. For arithmetic operations, it is conventional to use %P1 for the destination pointer and %P2 through %P4 for source operands.

Five registers are allocated for performing Vector Operations. These are the *increment registers*, %I1 through %I4, and the *vector length* register %V. %V holds the vector length, i.e. the loop count for doing vector operations. %I1 though %I4 hold the stride values to be added to argument registers %P1 through %P4. Macroinstructions are provided to perform %Pi = %Pi + %Ii. This feature is used, amongst other possibilities, for VP looping. PARIS uses these register for the same purpose, and expects them not to change. Therefore if mixing IMPs with PARIS calls, *do not clobber %I1 through %I4 or %V* (see Appendix D).

Four *counter registers*, %C1 through %C4 are provided for counted loops, and macroinstructions are provided for incrementing, decrementing, and testing these.

Eight *flag registers* are allocated for controlling the CM flags. Registers %CTX-BASE, %CAR-BASE, %OFL-BASE, %TST-BASE hold the base addresses for the four CM flags. Registers %CTX-INC, %CAR-INC, %OFL-INC, %TST-INC hold the increments (strides) to be used for virtual processor looping. Macroinstructions are provided which add the increments to the base addresses.

A register is also used as a stack pointer to the IMP control stack (%IMP-SP). This register is not directly accessible, but rather is controlled implicitly by the control stack macroinstructions. The IMP control stack is described in the next section.

No registers allocated to existing mechanisms in PARIS or the dispatch loop are otherwise allocated.

4.1. Preservation of Registers

All registers are preserved within the execution of a single IMP. The execution of an IMP destroys the contents of register %A1, thus it is not preserved from one IMP execution to the next, but all other registers are preserved.

Execution of PARIS instructions destroy various registers. In general you cannot expect PARIS to preserve any of the %A, %I, %P, or %C registers.

5. The Control Stack

Because the argument registers are limited, and because it is useful to allow one IMP to call another, a control stack is provided. This stack is located in scratch-ram and should not be confused with the stack defined by PARIS, located in CM memory. The control stack grows upward in memory. A reserved register %IMP-SP points to the top element on the stack. The word on the top of the stack (the last value pushed) is located at *stack offset* of zero. The word pushed before it (second from the top) is at stack offset 1, and so forth.

The control stack is used by IMPs for two purposes. First, it is used for saving and restoring the IMP Executor step counter so that IMPs can be nested. Second, it is used by built-in macroinstructions provided to push and pop selected registers.

Macroinstructions are required not to clobber %IMP-SP, but otherwise are free (subject to rules discussed later) to use the control stack and scratch memory. See Appendix B for a scratch memory map.

6. The IMP Control Macroinstructions

A set of macroinstructions are provided which implement the control mechanisms of the IMP programming model. Some of these take arguments, supplied in-line. In-line arguments are documented with each instruction.

Each individual argument takes one word of SRAM. In some cases, arguments consist of multiple data packed into a single SRAM word.

6.1. Register Control Macroinstructions

The following IMP control macroinstructions manipulate the IMP register set, including loading with constant values, performing simple arithmetic, transfers from register to register, and setup of arguments.

6.1.1. IMP-INC-{%C{1-4},%P{1-4}}

This instruction increments a specified counter register or pointer register by one.

6.1.2. IMP-DEC-{%C{1-4},%P{1-4}}

This instruction decrements a specified counter register or pointer register by one.

6.1.3. IMP-MOVE-%A{1-12}-TO-{%A{1-12},%P{1-4},%I{1-4},%C{1-4}}

These instructions move the 32-bit value from one of the %A registers to any selected register, %A*n*, %P*n*, %I*n*, or %C*n*.

6.1.4. IMP-MOVE-{%A{1-12},%P{1-4},%I{1-4},%C{1-4}}-TO-%A{1-12}

These instructions move the contents of a selected register %A*n*, %P*n*, %I*n*, or %C*n* register to a selected register %A*m*.

6.1.5. IMP-MOVE-%P{1-4}-TO-%P{1-4}

These instructions move the contents of a selected pointer register to another pointer register.

6.1.6. IMP-MOVE-%I{1-4}-TO-%I{1-4}

These instructions move the contents of a selected increment register to another increment register.

6.1.7. IMP-MOVE-%C{1-4}-TO-%C{1-4}

These instructions move the contents of a selected counter register to another counter register.

6.1.8. IMP-MOVE-%{CTX,OFL,TST,CAR}-BASE-TO-%P{1-4}

These instructions move the contents of a selected flag base register to a selected pointer register. This is useful for preparing to do calculations via the Beta Chip instructions (LLS etc.) that involve the values of the flags.

6.1.9. IMP-LOAD-{%P{1-4},%I{1-4},%A{1-12},%C{1-4}}

Argument 1:	Value

IMP registers. The constant is argument 1 of the macroinstruction.

6.1.10. IMP-RECEIVE-{reg}

These instructions receive a single 32-bit register value from the next available word in the IFIFO. *Reg* may be any of the following:

%A1 through %A12
%P1 through %P4
%I1 through %I4
%C1 through %C4
%V
%CAR-BASE %CAR-INC
%OFL-BASE %OFL-INC
%TST-BASE %TST-INC
%CTX-BASE %CTX-INC

These instructions require no arguments.

6.1.11. IMP-RECEIVE-REGS-{A-D}

Argument 1:	Register Mask

These macroinstructions read 32-bit values from the IFIFO and load then into selected registers. Selection is done using one of four 16-bit in-line masks. This is the mechanism for accepting data from the front end. There are four instructions, one for each of four masks, A, B, C, or D. Bits in the bit masks are associated with registers as shown below. The first register in the list is associated with bit 0 (LSB) and so forth, and the registers are passed in that order.

Mask A:
  %P1 %P2 %P3 %P4 %I1 %I2 %I3 %I4 %C1 %C2 %C3 %C4 %R15 %R18 %R19 %R20

Mask B:
  %R0 %A1 %A2 %A3 %A4 %A5 %A6 %A7 %A8 %A9 %A10 %A11 %A12 %R21 %R22 %R23

Mask C:
  %R39 %R38 %R37 %R36 %R35 %R34 %R33 %R32 %R31 %R30 %R29 %R28 %R27 %R26 %R25 %R24

Mask D:
  %R40 %V %ctx-base %ctx-inc %tst-base %tst-inc %car-base %car-inc %ofl-base %ofl-inc
    %IMP-SP %R59 %R60 %R61 %R62 %R63

The registers %R*n* are included for completeness, and fill in the remaining hardware registers which exist on the microcontroller. There is really no reason to care much about their contents, and you most definitely must *not* change their contents.

You should note that these instructions are very slow. They are provided for completeness in accessing the microcontroller's registers, and are used internally by the IMP tools. As a result, you will probably want to avoid these in favor of the much faster individual sends, e.g. IMP-SEND-%A1.

6.1.12. IMP-SEND-{reg}

These instructions send a single 32-bit register value to the front end via the OFIFO. *Reg* may be any of the following:

%A1 through %A12
%P1 through %P4
%I1 through %I4
%C1 through %C4
%V
%CAR-BASE %CAR-INC
%OFL-BASE %OFL-INC
%TST-BASE %TST-INC
%CTX-BASE %CTX-INC

These instructions require no arguments.

As a convenience in programming or debugging IMPs, these routines are additionally callable from the front end. For example, in lisp, the following code will return the value in %C2:

```
(progn (cmi::imp-send-%C2) (cmi::read-fifo-32))
```

Similarly, in C the equivalent code is:

```
_CMI_imp_send__C1();        /* note double underscore */
value = _CMI_read_fifo_32();
```

6.1.13. IMP-SEND-REGS-{A-D}

Argument 1:	Register Mask

These instructions work like their "RECEIVE" counterparts, but in the output direction, i.e. they send 32-bit register contents to the front end via the OFIFO.

The instructions IMP-SEND-REGS-{A,B,C,D} all clobber the contents of %A1. As a result, the instruction IMPS-SEND-REGS-B will not correctly send the contents of %A1.

You should note that these instructions are very slow. They are provided for completeness in accessing the microcontroller's registers, and are used internally by the IMP tools. As a result, you will probably want to avoid these in favor of the much faster individual sends, e.g. IMP-SEND-%A1.

6.1.14. IMP-ADD-%A{1-12}-TO-%A{1-12}

These instructions add the contents of any specified %A register to another %A register. The result is left in the second %A register. These instructions take no arguments.

6.2. Control Stack Macroinstructions

The following macroinstructions manipulate the control stack. All IMPs, whether invoked from the front end or called from another IMP *must* leave the stack pointer at the same location as it had upon entry into the IMP.

6.2.1. IMP-{INC,DEC}-SP

These instructions create and collapse control stack frames. %SP is incremented or decremented by the value given in argument 1. Since the stack grows upward, INC adds a frame, DEC removes it.

Argument 1:	Adjustment

6.2.2. IMP-{PUSH,POP}-REGS-{A-D}

Argument 1:	Register Mask

These macroinstructions push and pop a selected set of registers onto the control stack. Selection is done via a 16-bit mask given in argument 1. The mask is processed from LSB to MSB. For the PUSH instructions, registers are associated with the mask bits in the same order as previously described for IMP-RECEIVE-REGS-x and so forth. The **impass** constructor functions **reg-mask-{a-d}** construct this mask correctly. For example, register group A consists of the following ordered set of registers:

%P1 %P2 %P3 %P4 %I1 %I2 %I3 %I4 %C1 %C2 %C3 %C4 %R15 %R18 %R19 %R20

The IMP step (IMP-PUSH-REGS-A (reg-mask-a '%P1 '%P2 '%P3)) will properly push the three registers in that order. The order comes from the ordering within the register group, not from the ordering of the arguments to **reg-mask-a**, thus the IMP step (IMP-PUSH-REGS-A (reg-mask-a '%P2 '%P1 '%P3)) is identical in function.

For the POP instructions, the mask is still processed from LSB to MSB, but the mask bits are associated with the registers in the group in *reversed order*. The **impass** constructor functions **reg-mask-{a-d}-reversed** build masks with the bits reversed. Thus the IMP step (IMP-POP-REGS-A (reg-mask-a-reversed '%P1 '%P2 '%P3)) will pop the registers pushed in the above example in reverse order, restoring them to their original values. Again, the order that registers are specified in the call to **reg-mask-a-reversed** is inconsequential, popping will always be done in descending order within the register group.

The push/pop macroinstructions have several uses. First, they can be used as a save-and-restore mechanism around calls to other IMPs, allowing the nesting of context.

These instructions take about 40 cycles to execute, regardless of the bit mask, thus should only be used to push a large number of registers. You should investigate the possibility of using one of the more specialized PUSH/POP instructions below for improved performance.

Also, you should note that using these instructions to either push or pop the stack pointer register, %IMP-SP, will corrupt the stack, and so must be avoided.

6.2.3. IMP-{PUSH,POP}-{%A{1-12},%I{1-4},%C{1-4},%P{1-4}}

These instructions each push or pop a single register, one of the general, pointer, counter, or increment registers, on to or off of the control stack. Each of these instructions takes 5 cycles. These instructions take no arguments.

6.2.4. IMP-{PUSH,POP}-%A1-THROUGH-%A{2-12}

These instructions push or pop the block of registers %A1 through %A*n* where *n* is 2 through 12. These instructions provide an efficient way of saving and restoring general registers around calls or other code which may alter these registers. These instructions are about twice as fast as using IMP-PUSH-REGS-B.

6.2.5. IMP-{PUSH,POP}-%I1-THROUGH-%I4

These instructions push or pop the four of registers %I1 through %4. These instructions provide an efficient way of saving and restoring the increment registers. This is especially useful when coding IMPs which will be called along with PARIS instructions, since PARIS requires the preservation of the increment registers from instruction to instruction (see Appendix D).

6.2.6. IMP-{READ,WRITE}-STACK-A{1-12}

Argument 1:	Stack Offset (in words)

These instructions transfer the Nth element from the top of the control stack to (READ) or from (WRITE) %A*n*. N is a signed 32-bit value given in argument 1 and is a positive offset into the stack. Thus, %IMP-SP points to the top element (offset zero), the second stack element is at offset 1 etc. Negative offsets may be used to access unused stack for temporary storage, etc.

6.3. Flow Control Macroinstructions

The following macroinstructions implement the ability to do looping and conditional branching within an IMP.

6.3.1. IMP-JUMP

Argument 1:	Jump Offset

This macroinstruction implements a relative jump within an IMP. Its argument is a signed 32-bit number which is added to the current IMP step counter, which points to the NEXT step. Execution continues from the resulting step address. No bounds checking is performed.

Jump offsets are generated using the argument syntax:

LISP:

```
    (IMP-JUMP  (OFFSET label))
```

IMPASS:

```
    IMP_JUMP
      OFFSET label
```

6.3.2. IMP-JUMP-{Z,NZ}-%C{1-4}

Argument 1:	Jump Offset

This macroinstruction performs a relative jump conditionally. The jump is taken if the specified register either contains zero (the "-Z-" instructions) or a non-zero value (the "-NZ-" instructions).

The *Jump Offset* argument is constructed using the "OFFSET" argument syntax (See **IMP-JUMP**, above).

6.3.3. IMP-DEC-%C{1-4}-JUMP-{Z,NZ}

Argument 1:	Jump Offset

This macroinstruction is the same as IMP-JUMP-NZ-%C{1-4} except that the specified counter register %C*n* is decremented before the test. This instruction can be used to implement counter loops using a counter register as the loop count variable.

The *Jump Offset* argument is constructed using the "OFFSET" argument syntax (See **IMP-JUMP**, above).

6.3.4. IMP-CALL-IMP

The IMP-CALL-IMP macroinstruction invokes the IMP Executor from within an IMP. It has one argument, the SRAM address of the imp to be executed. When the execution of the called imp completes, control is returned to the next step in the calling imp.

Argument 1:	IMP Address in Scratch RAM

The **impass** argument syntax "IMP *name*" is used to generate the load address of an IMP. Note that this is not known at assembly time. Rather, **impass** generates a *linkage* to the IMP which is resolved at load time.

6.3.5. IMP-CALL-IMP-IND-%A{11,12}

This macroinstruction is like IMP-CALL-IMP except that it executes the IMP whose address (in SRAM) is contained in the designated register %A11 or %A12.

This macroinstruction requires no arguments.

The IMP Tools (LISP or C) provide specific support for this instruction. This is because it is necessary to assure that when the instruction is executed, that all IMPs which may possibly be called are currently loaded. This is accomplished by using a special technique to execute any IMP which contains IMP-CALL-IMP-IND-%A*n* calls. When executing such an IMP, you must do so using the LISP function **imp::execute-imp-requiring-imps** or the C function **IMP_execute_imp_requiring_imps**. Consult the

LISP IMP Tools Reference Manual or the *UNIX IMP Tools Reference Manual* for a description of these functions and examples of their use.

6.3.6. IMP-EXEC-COROUTINE-%A{9-12} and IMP-COROUTINE-EXIT-%A{9-12}

These instructions provide a much faster way for an IMP to call another IMP. This requires, however, that the IMP(s) being called are no longer callable directly from the front end via **execute-imp-id**.

The mechanism uses a specified %A register as a linkage register in which the return address (address of the instruction after the CALL) is saved. Registers %A9 through %A12 are used as linkage register, thus up to four call levels are possible, more if the %A registers are saved around calls.

This call takes 3 cycles, plus 3 cycles for the return making it much faster than IMP-CALL-IMP.

As an example, we can code a coroutine KILL-TIME as follows:

```
(IMP-LOAD-%C3 100)
(label LOOP)
(IMP-DEC-%C4-JUMP-NZ (offset LOOP))
(IMP-COROUTINE-EXIT-%A3)
```

which can be called with the instruction:

```
(IMP-EXEC-COROUTINE-%A3)
```

Note that KILL-TIME must preserve %A3, since the return address is stored there. Also, KILL-TIME may *not* be executed from the front end using **execute-imp-id**

6.3.7. IMP-EXEC-COROUTINE-%A{9-12}-IND-%A{9-12}

This is the indirect version of IMP-EXEC-COROUTINE-%A*n*, i.e. it is identical except that it executes the IMP whose address is contained in the specified (second) %A register. The linkage register and the address register may not be the same register.

As with IMP-CALL-IMP-IND-%A*n*, the IMP Tools (LISP or C) provide specific support for this instruction. This is because it is necessary to assure that when the instruction is executed, that all IMPs which may possibly be called are currently loaded. This is accomplished by using a special technique to execute any IMP which contains IMP-CALL-IMP-IND-%A*n* calls. When executing such an IMP, you must do so using the LISP function **imp::execute-imp-requiring-imps** or the C function **IMP_execute_imp_requiring_imps**. Consult the *LISP IMP Tools Reference Manual* or the *UNIX IMP Tools Reference Manual* for a description of these functions and examples of their use.

6.3.8. IMP-LOOP-TOP-%A*n*, IMP[-DEC-%C*n*]-JUMP-%A*m*

These instructions, used together, implement a fast unterminated loop facility. Conditional looping is not possible using this method.

To code such a loop, the instruction preceding the loop is the IMP-LOOP-TOP-%A*n* instruction. This saves the address of the next instruction in the specified %A*n* register.

The body of the loop follows, using the IMP-[DEC-%C*n*]JUMP-%A*n* instructions at the bottom to loop back to the top of the loop body. These are the only JUMP instruction which may be used. Note that the optional decrement of %C*n* does *not* affect the jump, it is only provided as a convenience since it does not make the instruction any slower.

Example:

```
(IMP-LOAD-%P1 0)
(IMP-LOAD-%C1 5)                ;Loop count
(IMP-LOOP-TOP-%A5)

(IMP-JUMP-Z-%C1)                ;Loop exit test (2 cycles if not taken)
(IMP-INC-%P1)                   ;Loop Body, executed 5 times
 ...

(IMP-DEC-%C1-JUMP-%A5)
 ...
```

6.3.9. IMP-CALL-PARIS[-IND-%A{11,12}]

These instructions allow the IMP programmer to call internal microcode routines coded via DEF-MIN-MIC or DEF-MIN-MIC-CALLABLE. If you are not familiar with the internal CM2 microcode, you should not be considering using this instruction.

For microcode routines coded with DEF-MIN-MIC, arguments must be setup in %A1 through %An before the call is made.

Microcode routines coded via DEF-MIN-MIC-CALLABLE may also be called, except that the arguments must be set up in %R39 downward. This can be done by setting up the arguments in %A1 through %An and transferring them to %R39 downward using the macroinstruction IMP-MOVE-CALL-ARGS-{1-12}.

The function of this macroinstruction is to preserve the current contents of the Dispatch Register and the Scratch RAM Counter Register, both of which are used by the imp executor, and both of which can be clobbered by older DEF-MIN-MIC code.

The IMP-CALL-PARIS instruction takes the call address from the single inline argument. This address is the address of the microcode routine's "-AFTER-FIFO-POPS" entry point. **Impass** will automatically generate this address for you if you use the "MI *name*" syntax for the argument (see the Common LISP or UNIX Tools Manual for details on source syntax).

Argument 1:	MIN-MIC Entry Point (After FIFO Pops)

The IMP-CALL-PARIS-IND-%A*n* instructions perform the same action, but get the entry address of the PARIS microcode routine from the specified register, either %A11 or %A12.

Routines coded via the newer version of **def-min-mic** which have *only* inline entry points (and not front-end dispatch table style entry points) are not eligible for use with this method of call. *Impass* does not currently catch this error, so you must be careful. To be safe, follow this rule: If the routine is an IMP or CMIS macroinstruction, call directly in the usual fashion. In all other cases, use the IMP-CALL-PARIS instructions.

You should note that most older style microcode routines not only clobber their arguments, but also clobber additional registers. DEF-MIN-MIC routines called with args in %A1 through %A*n* generally clobber an unspecified number of registers from %A1 upward. DEF-MIN-MIC-CALLABLE routines clobber registers from %R39 downward. The registers defined as part of the IMP mechanism other than the %A*n* registers are preserved through calls to older DEF-MIN-MIC routines. **Note: In the current implementation the two counter registers %C3 and %C4 and the two pointer registers %P3 and %P4 are not preserved through PARIS microcode calls. %P1, %P2, %C1 and %C2 are preserved however.**

6.3.10. IMP-LOAD[-CALLABLE]-ARGS-{1-8}

These instructions load a specified number of constants successively into registers %A1 through %An. IMP-LOAD-CALLABLE-ARGS-{1-8} works the same except that it loads registers %R39 down through %R(39-n+1).

6.3.11. IMP-MOVE-CALL-ARGS-{1-12}

These macroinstructions allow for the invocation of microcode routines written using DEF-MIN-MIC-CALLABLE. They move a selected number of registers from %A1 through %An (%1 through %n) to the temp registers %R39 through %R39-n+1.

6.3.12. IMP-EXIT

The "IMP-EXIT" macroinstruction exits the IMP executor. If the IMP was invoked from another IMP, the calling IMP's execution continues. If the IMP was from invoked from the front end (top-level), control passes to the front-end. This is the ONLY way to terminate execution of an IMP.

6.3.13. IMP-ERROR

Argument 1:	Code

IMP-ERROR signals an error condition to the front end. It uses the "uc-ferror" mechanism to print the error string

IMP Error, Code=nnn

where nnn is the value of the one in-line argument.

6.4. Vector Support Macroinstructions

The IMP mechanism provides support for "vector operations", i.e. operations which perform the same parallel operation over an array of fields in each physical processor. VP Looping is an important example of this kind of operation, but others exist, including support for array (field or slice) operations, shared arrays, etc.

To support vector operations, instructions are provided for executing single operations within a loop wherein the memory addresses are incrementable by a "stride" value. The instructions, along with the looping mechanisms already defined, complete the ability to do VP looping in the microcontroller.

6.4.1. IMP-MOVE-%V-TO-%C{1-4}

These macroinstructions transfer the current vector length register contents, stored in %V, into a given counter register %C*n*. This can then be used as a loop counter for vector looping. The looping control macroinstructions can then be used to effect the vector loop.

6.4.2. IMP-ADD-%I{1-4}-TO-%P{1-4}

These macroinstructions provide a mechanism for vector looping by adding a stride increment (in one of the four increment registers) to any one of the pointer registers. IMP-ADVANCE-POINTERS advances all four pointer registers by their corresponding increments.

6.4.3. IMP-SUBTRACT-%I{1-4}-FROM-%P{1-4}

These macroinstructions provide a mechanism for adjusting any of the pointer registers *downward* by any of the increment registers.

6.4.4. IMP-ADD-%A{i}-TO-%P{n}

This macroinstruction provides a mechanism for adjusting the current contents of %P*n* by an amount stored in one of the %A registers.

6.4.5. IMP-ADD-CONST-TO-%P{1-4}

Argument 1:	Value

This macroinstruction provides a mechanism for adjusting the current contents of %P*n* by an constant amount, taken from argument 1 of the instruction.

6.4.6. IMP-{INC,DEC}-%P{1-4}

These macroinstructions provide a mechanism for adjusting the current contents of %P*n* by one. These are equivalent to IMP-ADD-CONST-TO-%P*n* with an argument of one, but are a cycle faster (one cycle as opposed to two).

6.4.7. IMP-ADVANCE-POINTERS

This instruction adds the current values of %I*n* to %P*n* for each *n* 1 through 4, i.e.

%P1 <— %P1 + %I1,
%P2 <— %P2 + %I2,
%P3 <— %P3 + %I3, and
%P4 <— %P4 + %I4.

6.4.8. IMP-JUMP-NO-SELECTED-PROCESSORS

IMP-JUMP-NO-SELECTED-PROCESSORS allows for skipping blocks of code when all processors are deselected, i.e. when for each physical processor, the physical Context Flag (not the PARIS Context Flag bit field in memory) is zero. This allows vector loops to be constructed which skip vector elements on which computation is not required across the full machine.

Care should be taken to use this instruction only when it is expected that it is likely enough that all processors will be deselected. This instruction costs 6 cycles if the jump is not taken, so can represent a substantial loss in performance if the jump is rarely taken. On the other hand, if taken fairly often to skip substantial computations, that savings can be well worth the cost of the instruction.

Argument 1:	Jump Offset

The *Jump Offset* argument is constructed using the "OFFSET" argument syntax (See **IMP-JUMP**, above).

6.4.9. IMP-ADVANCE-FLAGS-{names}

This macroinstruction provides looping over the CM2 processor flags. For each named flag, it advances the flag pointer *%flag*-BASE by the value in *%flag*-INC. *Names* is any of the 15 combinations of flag names CAR, TST, OFL, and CTX, in that order, separated by dashes. For example, to advance flags

Overflow and Context conditionally, use IMP-ADVANCE-FLAGS-COND-OFL-CTX. Name combinations not in the correct order (e.g. IMP-ADVANCE-FLAGS-COND-CTX-OFL) are not allowed.

6.4.10. IMP-{SAVE,RESTORE}-FLAG-BASES-{names}

These instructions save or restore any of the 15 combinations of flag base registers. *Names* is any of the 16 combinations of flag names CAR, TST, OFL, and CTX, in that order, separated by dashes. For example, to Save the Overflow and Context flag base registers, use IMP-SAVE-FLAG-BASES-OFL-CTX. Name combinations not in the correct order (e.g. IMP-SAVE-FLAG-BASES-CTX-OFL) are not allowed.

Saving is done using registers R39, R38, R37, and R36 in the microsequencer. These registers are generally clobbered by PARIS calls. As a result, these instructions are generally used to save and restore these registers within the context of a single IMP.

6.4.11. IMP-{PUSH,POP}-FLAG-BASES

These macroinstructions push and pop the four flag base registers onto the control stack. The registers are pushed in the following order: Context, Test, Carry, Overflow (thus the overflow flag is at the top of the stack after the push). Popping is done in the reverse order, restoring the flags to their values before they were PUSHed.

These instructions require no arguments.

The instructions IMP-SAVE-FLAG-BASES-{names} and IMP-RESTORE-FLAG-BASES-{names}, described above, are considerably faster than using the stack, and should be considered as an alternative where performance is an issue.

6.5. Instructions for Gathering Timing Statistics

The CM2 microsequencer includes hardware for counting certain events which can occur during microcode execution. In particular, at any one time, any one of the following may be counted:

1. High Speed clock ticks. The clock which is generated directly by the CM2 timing crystal, divided by four, is counted. For an 8 Mhz machine, each count is thus 500ns.
2. Microinstructions, each of which take from 4 to 13 high speed ticks. Four is by far the most common, but 8-tick instructions occur when CM memory is written, a page fault (row strobe) is required for a CM memory read, or when memory refresh cycles occur.
3. Page Faults, which cause a microcycle which does a memory read to take 8 high-speed ticks instead of 4. These occur each time a memory read is performed at address A after a previous read or write to address B and addresses A and B are not on the same dynamic memory "page" (row). In machines with 64k memories, pages are 256 bits, in 1 Megabyte memories, pages are 1024 bits.
4. Memory Refreshes, which occur at regular intervals and cost either 4 or 8 high-speed ticks to perform. Refresh has an overall impact on the execution rate of the machine of 3.5 percent (7 percent in systems with Rev 2 Nexus Boards).

There is a single 40-bit counter and instructions are provided for clearing the counter, enabling (starting) counting of one of the above four events, disabling counting, and reading back the counter value. The counter is disabled and cleared using the IMP-PM-CLEAR instruction. To enable counting of one of the events, the IMP-PM-xxx-COUNT-START instructions are used. To stop the counter, the IMP-PM-STOP instruction is used. The counter can then be restarted to continue the counting, or can be read out into an A register using IMP-PM-MOVE-LOW-COUNTER-TO-%A*n*) or sent directly up the OFIFO using IMP-PM-SEND-COUNTER. Also, while stopped, the counter can be manually incremented by one using the IMP-PM-INC-COUNTER instruction.

You should be aware that these instructions themselves can contribute to the counts. The instructions to enable and disable the counter will themselves contribute 1 cycle, or 4 high-speed clock ticks, no page faults, and at most one memory refresh (rarely).

6.5.1. IMP-PM-CLEAR

This instruction disables counting and clears the counter.

6.5.2. IMP-PM-INC-COUNTER

This instruction increments the counter by one. The counter must be disabled when this instruction is performed or the state of the counter is unpredictable.

6.5.3. IMP-PM-START-TICK-COUNT

This instruction enables counting of high-speed clock ticks. The counter is incremented by oneafter each count of four ticks of the crystal clock. For an 8 Mhz machine, each tick is thus 4 125 ns ticks or 500 ns.

6.5.4. IMP-PM-START-CYCLE-COUNT

This instruction enables counting of microcycles (microinstruction executions).

6.5.5. IMP-PM-START-PAGE-FAULT-COUNT

This instructions enables counting of memory page faults.

6.5.6. IMP-PM-START-REFRESH-COUNT

This instructions enables counting of memory refreshes.

6.5.7. IMP-PM-STOP

This instruction disables counting. Only in the disabled state can the counter be read out or incremented manually.

6.5.8. IMP-PM-SEND-COUNTER

This instruction sends the value of the 40-bit counter up the OFIFO as two FIFO words. The first contains the low 32 bits of the counter, and the second contains the high 8 bits of the counter in the low 8 bits of the FIFO word.

6.5.9. IMP-PM-MOVE-LOW-COUNTER-TO-%A{1-12}

This instruction moves the low 32 bits of the 40-bit counter to the specified %A register. The counter is unaffected by this operations. Note that the upper 8 bits of the counter are not accessible using this instruction. Note that when counting high-speed clock ticks, only 132 seconds of counting can occur before the counter overflows 32 bits. Other counting modes will rarely overflow the counter (for example, when counting microcycles, the counter will not overflow for more than 23 days).

7. Appendix A - Register Allocation

The following registers may be clobbered by calls to PARIS routines:

%R1-%R12	%A1-%A12 (General/Argument registers)
%R13	%C3
%R14	%C4
%R16	%P3
%R17	%P4

The following registers are preserved through PARIS calls

%R41	%V (vector length register)
%R42-%R45	%I1-%I4 (Increment Registers)
%R46	%CTX-BASE
%R47	%CTX-INC
%R48	%TST-BASE
%R49	%TST-INC
%R50	%CAR-BASE
%R51	%CAR-INC
%R52	%OFL-BASE
%R53	%OFL-INC
%R54	%C1 (counter register 1)
%R55	%C2 (counter register 2)
%R56	%P1
%R57	%P2
%R58	%IMP-SP (Control Stack pointer)

8. Appendix B - Scratch Memory Allocation

Scratch memory is allocated as follows. There are 16K words of scratch space.

15000-16767	Reserved for scratch use by microinstructions (some PARIS functions put scratch stack here)
13000-14999	IMP Control Stack (growing upwards)
12000-12999	IMP scratch space (for use with IMP-SCRATCH-LOAD, IMP-SCRATCH-STORE
2000-11999	IMPs

9. Appendix C - IMP Internal Format

You do not need to know the format of IMPs as stored in the microcontroller SRAM. **Impass** will properly format IMPs for you. This section is intended for those who desire a brief understanding of IMP formats.

IMPs are nothing more than sequences of macroinstructions and their arguments. Each STEP consists of the microcode entry point of the macroinstruction followed by the arguments in order. However, due to the method the IMP executor uses to sequentially execute IMPs, arguments are skewed (out of phase) by one step in the forward direction. In other words, the arguments for step N actually follow step N+1. The resulting format is thus:

```
STEP N-1
 in-line argument 1 for step N-2
 in-line argument 2 for step N-2
 ...
STEP N
 in-line argument 1 for step N-1
 in-line argument 2 for step N-1
 ...
STEP N+1
 in-line argument 1 for step N
 in-line argument 1 for step N
 ...
```

Certain macroinstructions are further perturbed in format in that they expect their first argument to appear in place of the next step's call address. This is called the *arg1-counter* feature, and is used to facilitate optimized transfer of the first argument to the internal microsequencer loop counter register. Thus, if STEP N uses the arg1-counter feature, the format becomes:

```
STEP N-1
  in-line argument 1 for step N-2
  in-line argument 2 for step N-2
  ...
STEP N (uses the arg1-counter feature)
  in-line argument 1 for step N-1
  in-line argument 2 for step N-1
  ...
in-line-argument 1 for step N (counter) <-- Next step normally here
  in-line argument 2 for step N
  in-line argument 3 for step N
  ...
STEP N+1
STEP N+2
  in-line argument 1 for step N+1
  in-line argument 2 for step N+1
  ...
```

Moreover, when this feature is used, the value of the counter argument may actually be offset by a constant as it appears in the IMP to account for pipeline lags and uSequencer counter idiosyncrasies.

Again, when using IMPASS to construct IMPs, you need not concern yourself with these issues. Formatting of in-line arguments and offsetting of the counter arguments is done for you automatically by IMPASS.

10. Appendix D - Mixing IMP Executions with PARIS Calls

It is the extremely common case that IMP executions will be mixed with PARIS calls in the font-end application. IMPs can also internally call existing PARIS microcode routines via the IMP-CALL-PARIS or IMP-CALL-PARIS-IN-%An instructions. In both cases, certain rules must be followed when coding IMPs that will be used along with PARIS microcode.

Rule 1: Preserve the register %I1 through %I4 and %V.

PARIS uses the increment register and the vector length register for the same purposes as IMPs do (for vector striding). PARIS avoids loading these registers for each instruction when the are known to contain the desired values (i.e. they are cached). As a result, if your IMP changes the values in these registers, they must be restored before exiting the IMP. The instructions %IMP-MOVE-%I*n*-TO-%A*m* and

%IMP-MOVE-%A*n*-TO-%I*m* are the fastest and easiest way of accomplishing this. %IMP-PUSH-REGS-A and %IMP-POP-REGS-A can be used also, but are much slower.

These same microcode routines expect %I1 through %I4 and %V to be set up upon entry. When calling these functions internally (IMP-CALL-PARIS), you must be aware of what registers must be set up.

Rule 2: Do not rely on all registers being preserved through PARIS calls

All calls to PARIS from the front end or via IMP-CALL-PARIS will clobber the uController registers R1 through R39. This stretch of registers includes the IMP/CMIS registers %A1 through %A12, %C3, %C4, %P3, and %P4. Note that %P1, %P2, %C1, and %C2 are preserved.

Rule 3: Do not rely on the IMP Control Stack being preserved.

A few PARIS functions (very few) create and use an stack at location 12000 and upward. This is the same location as the default location of the IMP Control Stack. IMPs should generally not leave the stack with things pushed on it in any case, but they certainly should not expect those things to still be there after PARIS calls. When PARIS routines are called internally, it is best if the stack is flat.