CMIS Reference Manual

Version 5.2

Bob Lordi

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

ABSTRACT

This document contains a description of the CMIS macroinstruction set. This information is layered on the IMPs Mechanism which is documented in the "*Internal Macroinstruction Procedures Reference Manual*" which should be read before this.

Internal software versions at time of publication: PARIS 5.2 (beta), IMPS-F5202, IMPS-UC-F5203, CMIS-UC-F5204.

October 2, 1989

CMIS Reference Manual

Version 5.2

Bob Lordi

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

Internal Users Note:

A typeset copy of this document can be printed with the UNIX command

/cm/cmis/doc/pdoc -P*laserprinter* /cm/cmis/doc/cmis-ref-man.ms

Table of Contents

1. Overview

The IMP mechanism (see *IMPs Reference Manual*) offers a way of executing sequences of microcode routine calls ("macroinstructions") from a very tight dispatching loop running in the microcontroller. You should be thoroughly familiar with the IMP mechanism before reading this document.

The IMP mechanism includes a number of macroinstructions used to control the CM2 microcontroller and to control basic IMP execution. Included are simple looping, nesting and register manipulation macroinstructions.

CMIS completes the picture with a set of macroinstructions to control the CM2 itself. It includes instructions to control the CM processor chips themselves as well as the SPRINT/FPU hardware. The philosophy is to provide the basic low-level sequences, which can be pieced together to implement arbitrarily complex algorithms.

You will also have to be familiar with the low-level architecture. of the CM2. This manual covers this briefly in the following section but really is not sufficient to train the novice. You should seek help in understanding the CM2 architecture from your TMC applications engineer, or contact TMC Customer Support to arrange training.

1.1. 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-FRTA-%P{1-4}[-DYN]

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

2. CM2 Architecture

To use the CMIS instruction set, you must be very familiar with the low-level architecture of the Connection Machine. A concise discussion of the architecture of the CM2 as viewed by the CMIS programmer follows.

2.1. Sections

The basic unit of replication of the CM2 is what this document will refer to as a *SPRINT Section*, or *Section* for short. Figure 1 shows a block diagram of a CM2 section. The basic parts are the 32 CM2 Processors (2 CM2 chips), the memory, which is 32 bits wide by 65536 or 262144 (256K) bits in length, the SPRINT chip and the optional Floating Point Unit (FPU) chip.

The 32 bit wide memory is addressed by a 20-bit bit address bus. Most often, the address is driven by the microcontroller, as is the same value for all sections. It is possible, however, for the SPRINT chip to drive the address bus with a value that differs from section to section. This is called "SPRINT Indirect Addressing" and is discussed later. Memory data is read or written from two sources, either the 32 processors working in parallel, or the SPRINT chip which reads and writes 32-bit *slices*.

The 32 *Processors* each access one bit of memory at the currently addressed location. On any single cycle, they can read a bit, or write a bit, and perform one-bit calculations using memory data and the

state of four internal flags.

The SPRINT chip reads and write 32-bit slices, transferring them to or from one of its internal devices. The devices are the Transposers A through C (D in some new models), the Bypass Register, and the Conditionalization Register.

2.2. The CM2 Chip

The CM2 Chip contains 16 1-bit processors. There are two CM2 chips in each section, for a total of 32 processors. The 32 processors operate in lockstep, on each cycle performing an operation that typically reads or writes a bit in memory. In concert, they access a 32-bit slice of memory, one bit per processor. See Figure 2 for a block diagram of each processor.

There are five basic operations supported in CMIS, the LOADA cycle, the LOADI cycle, the LOADB cycle, the STORE cycle, and the RUG-W-A cycle.

The LOADA cycle reads a bit from memory and loads it into the A and C latches of each physical processor, sets the CARRY function for the ALU, and specifies the Read Flag to be used as input to the ALU.

The LOADI cycle loads a constant bit into the A and C latches of each processor.

The LOADB cycle reads a bit from memory into the B latch of each processor, sets the SUM function for the ALU, and specifies the Contextualization Flag to be used to contextualize the storing of the result and whether or not to invert this flag before using it as such.

The RUG-W-A cycle reads a bit from memory into the C latch of each processor.

The STORE cycle calculates the SUM and CARRY functions using the A and B latches and the Read Flag as input, stores the SUM function result to a bit in memory, and stores the CARRY function result into a specified Write Flag. The C latch is stored instead for processors whose Contextualization Flag is off.

There are four flags which may be specified as the Read, Write and Contextualization flags. Any of the four can be used for any purpose, but to support the PARIS programming model, these flags are named TEST, OVERFLOW, CARRY, and CONTEXT. A fifth flag, the ZERO or SINK flag, may also be used for reading, writing or contextualization - it always reads as a zero, and writing it has no effect.

2.3. The SPRINT Chip

The SPRINT chip is essentially a data path between the memory and the FPU, and is supported by CMIS instructions which provide common pipelines and machine sequences. Data is moved through the chip via its two external busses, the Memory Bus (Mbus) and the Float Bus (Fbus). The Mbus transfers 32-bit slices between memory and one of the four internal SPRINT devices, Data transposers A through C, and the Bypass Register. Similarly, the Fbus transfers between these devices and the data I/O bus of the FPU. A fourth data transposer, D, is available on later model SPRINT chips. See Figure 1 for the internal structure of the SPRINT chip.

The transposers are 32-bit by 32-bit matrices which are written with 32-bit values row-wise, and from which 32-bit values are read column-wise. This has the effect of rotating or "transposing" the data. This allows 32 32-bit PARIS fields to be read from the Mbus, then passed to the Fbus as 32-bit values. It is also possible to write these transposed values back into memory, after which they are said to be stored *slicewise*. Transposers are read or written using a single *transposer pointer*, which is generally incremented after each read or write, but may be loaded directly for irregular data access.

The SPRINT bypass register is a single 32-bit latch, into which data from the Fbus or Mbus may be read or written. Such data is not transposed or altered in any way. Using the bypass register, data stored slicewise in memory can be moved directly to and from the FPU.

Transfers between the Fbus and transposers may be conditionalized. The SPRINT *condition register* is used for this purpose and is loaded via the CMIS-FPU-MWC[-%P*n*] instructions.

An additional transposer, the Status Transposer, can optionally be used to collect 5 status bits from the FPU (two are unused in systems with WTL3132 FPUs). These status bits are passed directly, and in an 8-bit partially decoded form, making a total of 13 bits of status information collected into the Status Transposer. This data can then be stored as 13-bit fields in memory from which status information can be computed.

Conditional operation of the SPRINT chip works as follows:

The bit controlling whether an operation is selected or deselected is the N'th bit of the SPRINT Condition Register, where N is the current value of the transposer pointer for the transposer being accessed.

On Float Bus transposer reads (data transferred from SPRINT transposer to FPU), the data transferred will be taken instead from the Bypass Register if the condition bit is 0 (deselected).

On Float Bus transposer writes (data transfer from the Float Bus to transposer *i*), if the condition bit is 0, the data will be taken instead from the *companion* of transposer *i*. The companion to transposer A is transposer C (and vice versa); the companion to transposer B is transposer D (and vice versa) if transposer D exists on the given revision of the SPRINT chip. As a result, whenever transferring data from the FPU to a transposer, the transposer pointers of the destination transposer *and its companion* are advanced together. Both transposer pointers are advanced, even when conditional operation is disabled (i.e. after a CMIS-FPU-ALWAYS instruction).

2.4. The FPU (WTL3132 or WTL3164)

Your machine may have one of two available Floating Point Units (FPUs), or it may have no FPU at all. If you do not have an FPU, the CMIS instructions having names CMIS-FPU-*xxx*-DYN[AMIC] or CMIS-FPU-STATIC-*xxx* cannot be used. Instructions that use the SPRINT chip only may be used without any change in functionality.

If your system has an FPU, the chip used is either the Weitek WTL3132 or the Weitek WTL3164. Before programming either FPU, you should read the Weitek manual for the chip you have (supplied with the CMIS documentation). These manuals are complete, though you will have to read them carefully as they contain a great deal of information. You should also consult the examples at the end of this manual to see the basic techniques for programming the two FPU's.

The WTL3132 is capable of only single precision. It performs ADD, SUBTRACT, and MULTIPLY operations which are compliant with IEEE Standard 754. Divide is possible on this chip through use of an on-chip inverse seed table which gives an initial 7-bit approximation from which an inverse can be obtained by two iterations of Newton-Raphson algorithm. The chip is capable of an operation every three cycles, and can be pipelined to start a new operation on every cycle. A MULTIPLY/ADD operation is also available (3 cycles) but does not round correctly between the multiply and the add, so can give answers off by as much as two LSBs.

The WTL3164 is fully IEEE compliant and includes the DIVIDE and SQUARE ROOT operations. Operations are also included to convert between single precision, double-precision and 32-bit integer formats. For integers, add, subtract, multiply, and several logical operations are also provided. The chip has separate ADD and MULTIPLY pipes, and operations can be chained to get IEEE-compliant MULTIPLY/ADD operations.

Both chips have on-chip register files having 32 general registers and sufficient temporary registers to perform accumulating polynomial evaluations at full chip bandwidth.

Instructions are given to the FPU's in two halves. The first half is a 9-bit *Static Instruction*, which roughly specifies the function to be performed. The second half is a 24-bit *Dynamic Instruction*, which specifies the source operand and destination registers. Execution of the Dynamic Instruction causes an operation to be started on the chip. It is common to execute N of the same operation (e.g. ADD) by issuing one Static Instruction followed by N Dynamic Instructions.

The FPU's read and write external data across a single 32-bit data bus which is connected to the SPRINT chip. Double precision numbers on the WTL3164 must be passed in two cycles. It is possible to pass a value across the data bus and simultaneously start an operation using that value as input. This

is called *Input Bypass*, and is controlled differently on the two chips.

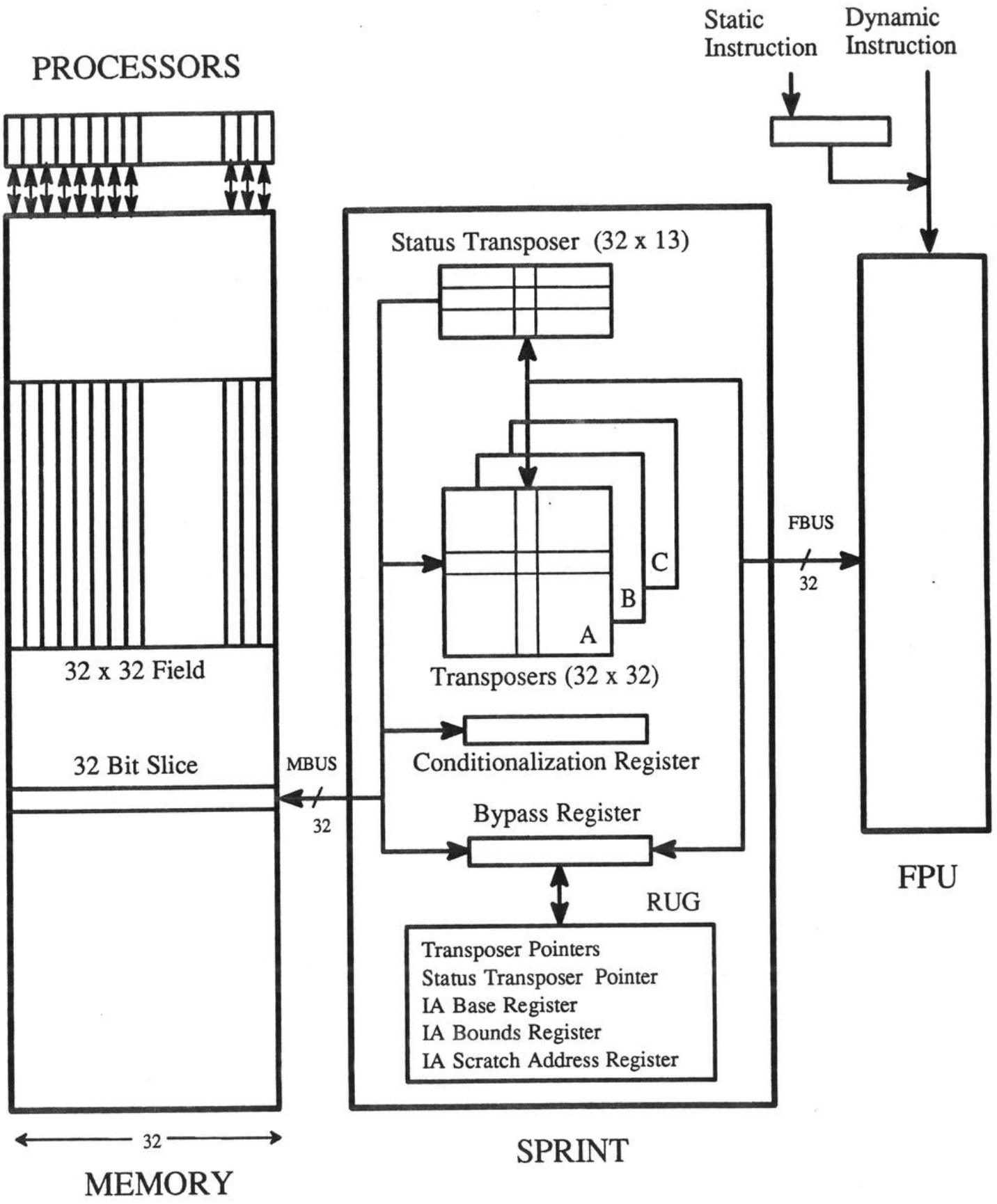


Figure 1: One Section

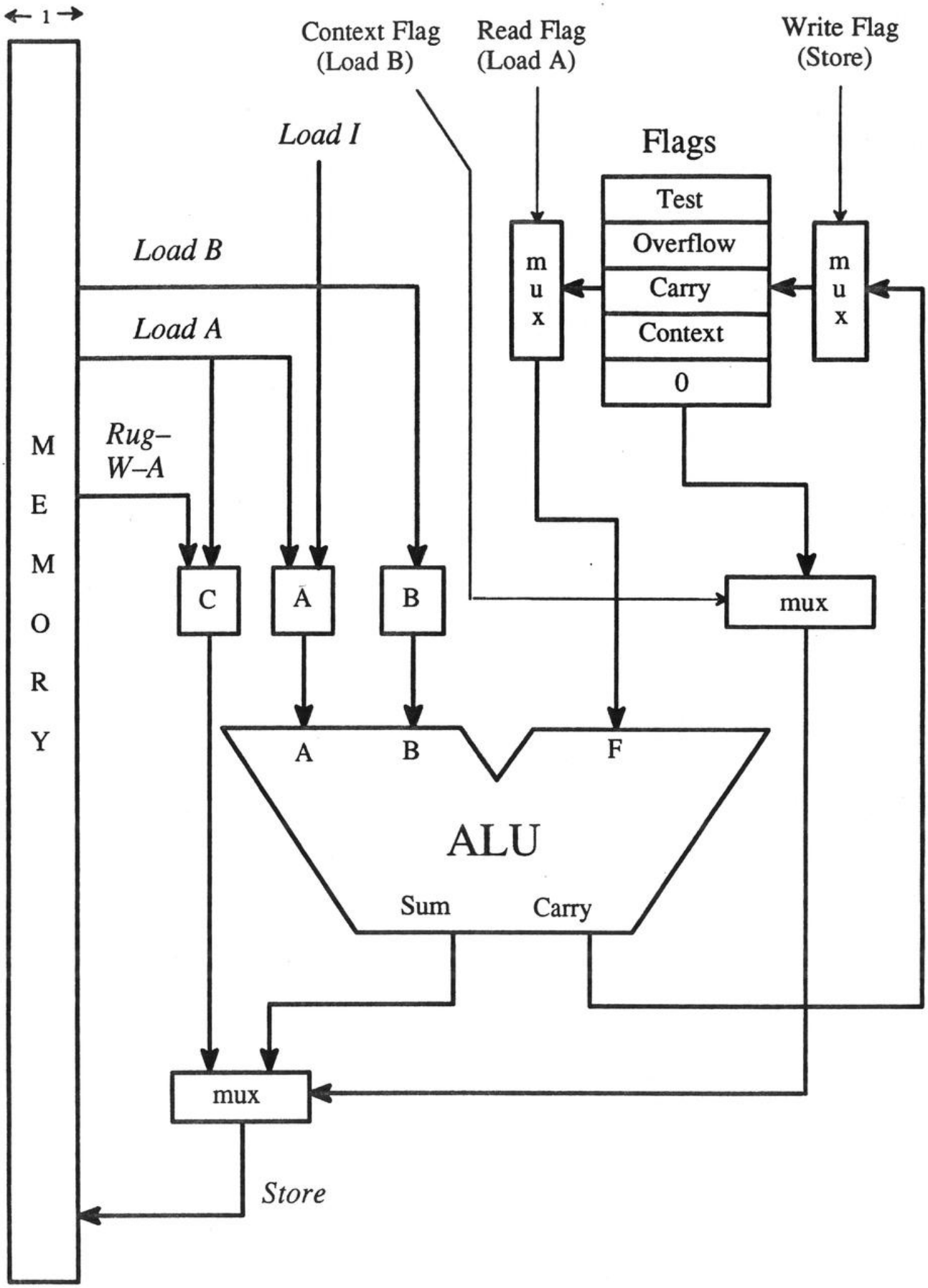


Figure 2: CM2 Chip

3. Registers

CMIS uses the registers defined by the IMP mechanism. In particular, it uses:

Register	*Usage*
%A1 through %A12	General data registers
%P1 though %P4	Field addresses (physical)
%I1 though %I4	Increments (strides)
%V	Vector length
%C1 through %C4	Counters
%{CAR,TST,OFL,CTX}-BASE	Flag Base Registers
%{CAR,TST,OFL,CTX}-INC	Flag Increment Registers

4. Instruction Set

The following sections describe the CMIS instructions that are available.

Unless specified otherwise, all instructions are unconditional. Conditionalization is not a part of the CMIS model. Rather, the mechanisms for controlling conditionalization are provided by CMIS instructions. For example, floating point operations, conditionalized by the SPRINT Configuration RUG contents, are controlled by the CMIS-FPU-ALWAYS and CMIS-FPU-CONDITIONAL instructions which setup this register.

Some examples of the *argument constructor functions* are given to help explain the usage of certain of the macroinstructions. You should consult the appropriate IMP Tools manual (Common Lisp or UNIX) for full definitions of the constructor functions and their syntax in the language you are using. LISP syntax is shown in this manual.

4.1. CM Chip Operations

The following macroinstructions do the basic sequences for controlling the CM Beta chip itself. Generally, addresses are taken from the pointer registers (%P*n*). The current version of CMIS supports only those cycles which support integer arithmetic, as described at the beginning of this manual under "CM2 Architecture".

4.1.1. CMIS-{LOAD,STORE}-FLAGS-{COND,ALWAYS}-{names}

These macroinstructions provide the ability to transfer CM flags to or from CM memory. The -LOAD instructions transfer the memory bits whose locations are contained in the Flag Base Registers for the selected flags into the physical flag latches in the CM2 chip. For example, loading the TEST flag is done by transferring the bit from the memory location addressed by the %TST-BASE register to the physical TEST flag latch.

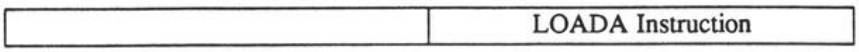
The STORE instructions do the opposite in that they transfer the bits in the specified physical flag latches out to the memory bits addressed by the corresponding Flag Base Registers.

Names is any of the 16 combinations of flag names CAR, TST, OFL, and CTX, in that order, separated by dashes. For example, to read flags Carry and Overflow conditionally, use the instruction CMIS-LOAD-FLAGS-COND-CAR-OFL. Name combinations not in the correct order (for example CMIS-LOAD-FLAGS-COND-OFL-CAR) are not allowed.

The -COND versions work conditionally on the context flag (before being loaded), the -ALWAYS- versions transfer the flag data unconditionally.

4.1.2. CMIS-LA

A single LOADA cycle is performed. %P1 contains the address of the memory location used to load the A and C latches. The inline arguments are formatted as follows:

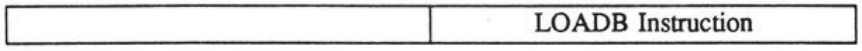


The argument can be constructed using the **IMPASS** argument constructor **loada-instr** as follows (first arg shown is default):

```
(loada-instr
  :carry-func ':f
  :read-flag {:ZERO-FLAG,:OVERFLOW-FLAG,...}
  :bsel {nil,t}
  :pmode {:ecc,:parity}
  :beta-instr-parity {0,1})
```

4.1.3. CMIS-LB

A single LOADB cycle is performed. %P2 contains the address used to load the B latch from memory. The inline arguments are formatted as follows:

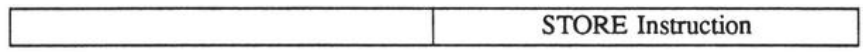


The argument can be constructed using the **IMPASS** argument constructor **loadb-instr** as follows (first arg shown is default):

```
(loadb-instr
  :sum-func ':a
  :condition-flag {:CONTEXT-FLAG,:TEST-FLAG,...}
  :condition-invert {nil,t}
  :pmode {:ecc,:parity}
  :beta-instr-parity {0,1})
```

4.1.4. CMIS-STORE

A single STORE cycle is performed. %P1 contains the address of the memory location used to STORE the current ALU function result (or C latch if conditionalization is enabled). The inline argument is formatted as follows:



The argument can be constructed using the **IMPASS** argument constructor **store-instr** as follows (first arg shown is default):

```
(store-instr
  :cube 0
  :edge {nil,t}
  :write-flag {:SINK-FLAG,:OVERFLOW-FLAG,...}
  :pmode {:ecc,:parity}
  :beta-instr-parity {0,1})
```

4.1.5. CMIS-LAS

A LOADA/STORE sequence is performed. %P1 contains the address used as the A and STORE address. The inline argument is constructed as follows:

Argument 1:

STORE Instruction	LOADA Instruction

The argument can be constructed using the **IMPASS** argument constructor **las-arg** as follows (first arg shown is default):

```
(las-arg
  :carry-func ':f
  :read-flag :ZERO-FLAG
  :bsel {nil,t}
  :cube 0
  :edge {nil,t}
  :write-flag {:SINK-FLAG,:OVERFLOW-FLAG,...}
  :pmode {:ecc,:parity}
  :beta-instr-parity {0,1})
```

4.1.6. CMIS-LAS-{INC,DEC}

These instructions are the same as CMIS-LAS except that they post increment or post decrement the pointer register %P1.

4.1.7. CMIS-LAS-{UP,DOWN}-%C{1-4}

The sequence LOADA, STORE is performed a counted number of times with the %P1 register post-incremented (or post-decremented) after each repetition. In-line arguments are formatted as in CMIS-LAS. %Cn contains the loop count (field size) and is not altered by the instruction.

4.1.8. CMIS-LBS

A LOADB/STORE sequence is performed. %P2 contains the address used as the B address, and %P1 contains the STORE address. The inline argument is constructed as follows:

Argument 1:

STORE Instruction	LOADB Instruction

The argument can be constructed using the **IMPASS** argument constructor **lbs-arg** as follows (first arg shown is default):

```
(lbs-arg :sum-func ':a
                 :condition-flag :CONTEXT-FLAG
                 :condition-invert {nil,t}
                 :cube 0
                 :edge {nil,t}
                 :write-flag {:SINK-FLAG,...}
                 :pmode {:ecc,:parity}
                 :beta-instr-parity {0,1})
```

4.1.9. CMIS-LBS-{INC,DEC}

These instructions are the same as CMIS-LBS except that they post increment or post decrement the pointer registers %P1 and %P2.

4.1.10. CMIS-LBS-{UP,DOWN}-%C{1-4}

The sequence LOADA, STORE is performed a counted number of times with the %P1 and %P2 registers each post-incremented (or post-decremented) after each repetition. In-line arguments are formatted as in CMIS-LBS. %Cn contains the loop count (field size) and is not altered by the instruction.

4.1.11. CMIS-LLS

This instruction executes a LOADB/LOADA/STORE sequence. The contents of %P1 are used as the A and STORE address. The contents of %P2 are used as the B address. Both %P1 and %P2 are unaffected by this operation. The instructions are taken from the inline arguments in the following format:

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

IMPASS argument constructors are provided for building the combined arguments. **IMPASS/LISP** syntax is as follows:

```
(lls-args :sum-func ':a
          :condition-flag {:CONTEXT-FLAG,...}
          :condition-invert {nil,t}
          :carry-func ':f
          :read-flag {:ZERO-FLAG,...}
          :bsel {nil,t}
          :cube 0
          :edge {nil,t}
          :write-flag {:SINK-FLAG,...}
          :pmode {:ecc,:parity}
          :beta-instr-parity {0,1})
```

For example, and ADD-WITH-CARRY is implemented with the LLS sequence:

LISP:

```
(lls-args
  :sum-func '(xor :a :b :f)
  :carry-func '(majority :a :b :f)
  :read-flag :carry-flag
  :write-flag :carry-flag)
```

C (impass):

```
lls_args sum_func=A^B^F, carry_func=(A+B)>>1, read_flag=carry_flag, write_flag=carry_flag
```

Note: The XOR and MAJORITY functions are provided in the IMP package (they are not common LISP).

4.1.12. CMIS-LLS-{INC,DEC}

These instructions are the same as LLS except that the %P1 and %P2 registers are incremented or decremented by one following the operation.

4.1.13. CMIS-LLS-{UP,DOWN}

These instructions perform repeated executions of CMIS-LLS-INC or CMIS-LLS-DEC with the memory addresses (in %P1 and %P2 as in the LLS macroinstruction) incremented by one each time. The number of iterations is taken from the first argument. The address registers are left incremented by the loop count at the end.

Argument 1:		Count
Argument 2:	LOADA Instruction	LOADB Instruction
Argument 3:		STORE Instruction

4.1.14. CMIS-LLS-{UP,DOWN}-%C{1-4}

These instructions are identical to CMIS-LLS-{UP,DOWN}, except that the number of iterations performed is taken from the specified Counter Register, rather than from an argument.

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

4.1.15. CMIS-LLS3

This instruction executes a LOADB/LOADA/STORE sequence, much like the CMIS-LLS instructions, except that 3 different memory addresses are used rather than using the same pointer for the A and STORE addresses. The contents of %P1 are used as the A address. The contents of %P2 are used as the B address. The contents of %P3 are used as the STORE address. None of the three pointers are affected by this operation. The instructions are taken from the inline arguments in the following format:

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

The **IMPASS** argument constructors provided for the the CMIS-LLS-*xxx* family of instructions are used for this instruction in like fashion.

Note that for deselected processors, the A source data is stored to the STORE location, rather than the ALU result. This does *not* achieve the effect of disabling deselected processors. As a result, this sequence is most often used for "always" operations, although there are interesting uses of this functionality with contextualization. If true contextualization is desired, the CMIS-LLLS-*xxx* instructions must be used instead.

4.1.16. CMIS-LLS3-{INC,DEC}

These instructions are the same as CMIS-LLS3 except that the %P1, %P2, and %P3 registers are incremented or decremented by one following the operation.

4.1.17. CMIS-LLS3-{UP,DOWN}

These instructions perform repeated executions of CMIS-LLS3-INC or CMIS-LLS3-DEC with the memory addresses (in %P1, %P2, and %P3 as in the LLS3 macroinstruction) incremented by one each time. The number of iterations is taken from the first argument. The address registers are left incremented by the loop count at the end.

Argument 1:		Count
Argument 2:	LOADA Instruction	LOADB Instruction
Argument 3:		STORE Instruction

4.1.18. CMIS-LLS3-{UP,DOWN}-%C{1-4}

These instructions are identical to CMIS-LLS3-{UP,DOWN}, except that the number of iterations performed is taken from the specified Counter Register, rather than from an argument.

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

4.1.19. CMIS-LLLS

This instruction executes a LOADB/LOADA/RUG-W-A/STORE sequence. This sequence is the proper way to do conditional, 3-operand bit serial operations, i.e. operations where the result is not stored back to one of the operands, but rather to a third location. This is a special case because it is necessary to load the *C* latch with the prior value of the result field so that the result field remains unchanged for deselected processors. Usage of this instruction is identical to the CMIS-LLS3 instruction. In particular, the usage of the pointer registers is the same. The contents of %P1 are used as the A address. The contents of %P2 are used as the B address. The contents of %P3 are used as the STORE address and for loading the *C* latch prior to the STORE cycle. None of the three pointers are affected by this operation. The instructions are taken from the inline arguments in the following format:

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

The **IMPASS** argument constructors provided for the the CMIS-LLS-*xxx* family of instructions are used for this instruction in like fashion.

4.1.20. CMIS-LLLS-{INC,DEC}

These instructions are the same as CMIS-LLLS except that the %P1, %P2, and %P3 registers are incremented or decremented by one following the operation.

4.1.21. CMIS-LLLS-{UP,DOWN}

These instructions perform repeated executions of CMIS-LLLS-INC or CMIS-LLLS-DEC with the memory addresses (in %P1, %P2, and %P3 as in the LLLS macroinstruction) incremented by one each time. The number of iterations is taken from the first argument. The address registers are left incremented by the loop count at the end.

Argument 1:		Count
Argument 2:	LOADA Instruction	LOADB Instruction
Argument 3:		STORE Instruction

4.1.22. CMIS-LLLS-{UP,DOWN}-%C{1-4}

These instructions are identical to CMIS-LLLS-{UP,DOWN}, except that the number of iterations performed is taken from the specified Counter Register, rather than from an argument.

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

4.1.23. CMIS-LILS-%A{1-12}

This instruction executes a LOADA/LOADI/LOADB/STORE sequence. This sequence is the proper way to do conditional or unconditional 3-operand bit serial operations where one of the operands is a constant. The constant is taken from a specified %A register, which is not affected. Usage of this instruction is similar to the CMIS-LLS3 instruction. The contents of %P1 are used as the STORE address, and the contents of %P2 are used as the B address. There is no A address because the A and C latches are loaded from the least significant bit of the constant in the %A register. None of the three pointers are affected by this operation. The instructions are taken from the inline arguments in the following format:

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

The **IMPASS** argument constructors provided for the the CMIS-LLS-*xxx* family of instructions are used for this instructions in like fashion.

4.1.24. CMIS-LILS-INC-%A{1-12}

These instructions are the same as CMIS-LILS-%A*n* except that the %P1 and %P2 registers are incremented by one following the operation.

4.1.25. CMIS-LILS-UP-%A{1-12}

These instructions perform repeated executions of CMIS-LILS-INC with the memory addresses (in %P1 and %P2 as in the CMIS-LILS-%A*n* macroinstruction) incremented by one each time. The number of iterations is taken from the first argument. The A operand bits are taken from successive bits of %A*n*, starting from the least significant bit and proceeding upward. If more than 32 iterations are performed, all further A bits are taken as zero. The address registers are left incremented by the loop count at the end.

Argument 1:		Count
Argument 2:	LOADA Instruction	LOADB Instruction
Argument 3:		STORE Instruction

4.1.26. CMIS-LILS-UP-%A{1-12}-%C{1-4}

These instructions are identical to CMIS-LILS-UP-%A*n*, except that the number of iterations performed is taken from the specified Counter Register, rather than from an argument.

Argument 1:	LOADA Instruction	LOADB Instruction
Argument 2:		STORE Instruction

4.2. Fieldwise Data Reading and Writing

The following CMIS macroinstructions access fieldwise data. Memory addresses for these instructions are always physical addresses.

Several of these instructions read or write a specific processor. In such instructions, the processor number (a physical processor number in the range 0 through 65535) is given in the %P4 register.

4.2.1. CMIS-WRITE-CONST-{ALWAYS,COND}-%A{1-12}[-%C4]

These instructions transfer a specified constant currently in %A*n* into a field in memory. The %P1 register specifies the address at which the constant will be written. %P1 is left advanced to the next location after the field (i.e. is advanced by the *length* argument) after the instruction completes.

The -ALWAYS version writes the field to all processors. The -COND version only writes to those processor whose physical Context Flag contains a one. The CMIS-LOAD-FLAGS-xxx instructions can be used to load the Context Flag, or the CMIS-SELECT-PROCESSOR can be used to select a single processor.

The length of the field is taken from an argument, except in the -%C4 instructions in which the length is taken from the %C4 register. This length must be in the range 1 through 32 inclusive.

4.2.2. CMIS-SELECT-PROCESSOR

The CMIS-SELECT-PROCESSOR instruction sets the physical context flag of the one selected processor whose processor number is in %P4 and clears the physical context flags of all other processors in the machine, contextualizing that processor alone.

This instruction can be used along with the CMIS-WRITE-CONST-COND-%A{1-12}[-%C4] instructions to implement a "write constant to processor" operation, as shown in the example:

Example: Load the 10 bit constant 299 into a 10-bit field at location 1000 in processor 65

```
(IMP-LOAD-%P4 64)                  ;Processor number
(CMIS-SELECT-PROCESSOR)            ;set Context Flag in processor 64 only

(IMP-LOAD-%A5 299)                 ;Constant
(IMP-LOAD-%P1 1000)                ;Memory address for constant
(CMIS-WRITE-CONST-COND-%A5 10)     ;Write the constant conditionally
```

4.2.3. CMIS-{U,S}-READ-PROC-%P{1,2,3}-%A{1,2,3}-%C4

This instruction reads a single fieldwise value from a specified physical processor and places the result in a specified register %A*n*. The physical processor number is taken from %P4. %P*n* gives the maddr, and the result is written to %A*n*, thus %A1 is written with a value accessed using %P1 and so forth. The length of the field transferred is taken from %C4, and must be 32 or less.

The CMIS-U-*xxx* versions treat the number as unsigned, and zero fill the remaining bits in the %A register.

The CMIS-S-*xxx* versions treat the number as signed, and fill the remaining bits with a copy of the sign bit (MSB) of the value.

As a side effect, this routine destroys the contents of transposer A, and clears all transposer pointers.

4.3. Slicewise Data Reading, Writing, and Moving

The following CMIS macroinstructions access slicewise data. Support is provided for 32-bit slicewise data only.

Memory addresses for these instructions are always physical addresses.

The CMIS-READ-CHIP-SLICE, CMIS-SEND-CHIP-SLICE, and CMIS-WRITE-CHIP-SLICE instructions read or write to a specific *SPRINT section.* The section is specified by the *Sprint chip section number*. This is the number of the SPRINT chip within the currently attached microsequencers. Section 0 is the section containing physical processors 0 through 31, section 1 contains physical processors 32 through 63 and so on. The number of available SPRINT sections is always the number of physical processors divided by 32.

Note: In versions of CMIS prior to 1.1, SPRINT sections were numbered absolutely through the entire CM, and it was necessary to calculate the absolute SPRINT section number from the relative section number. This is no longer required in version 1.1 and later, and programs written for prior versions which use the instructions in this section will no longer function correctly.

4.3.1. CMIS-READ-CHIP-SLICE-%P{1-3}-%A{1-3}

This instruction reads a 32-bit slice from a given SPRINT section in the system. %P*n* contains the memory address from which the slice is to be read. %P4 contains the SPRINT section number as described at the top of this section. The data is placed in %A*n*. Note that *n* is constant, i.e. when reading into %A1, the pointer register %P1 must be used, and so on.

This instruction requires no arguments.

4.3.2. CMIS-SEND-CHIP-SLICE-%P{1-3}

This instruction reads a 32-bit slice from a given SPRINT section in the system, and sends it up the OFIFO to the front end. %P*n* contains the memory address from which the slice is to be read. %P4 contains the SPRINT section number as described at the top of this section. The data is sent as a 32-bit value up the OFIFO to the front end, where it can be read via the functions **imp:receive-imp-data** (LISP) or **IMP_receive_imp_data** (C).

Note that this instruction is considerably faster than CMIS-READ-CHIP-SLICE, and is the correct choice when the data is to be moved up the OFIFO.

This instruction requires no arguments.

4.3.3. CMIS-WRITE-CHIP-SLICE-%P{1-3}-%A{1-3}

The 32-bit value in the specified %A register is transferred to the memory slice whose address is in the specified %P register. This transfer is only performed on the SPRINT section whose Sprint Section Number is given in %P4.

This instruction requires no arguments.

4.3.4. CMIS-MOVE-SLICES-%P1-TO-%P2-STRIDE[-%C4]

This instruction moves a vector of slices from one memory location to another. The source address of the vector to be moved is taken from register %P1. The destination address is taken from register %P2. The offset between elements in the source vector (the *stride*) is taken from register %I1. The destination stride is taken from register %I2.

Care must be taken when the vectors overlap. Notably, if the vectors overlap the destination is at a higher memory address, the move must be done from the top down. This can be accomplished by initializing the pointers to point to the last element of the source and destination vectors and using a negative stride.

Following execution, the pointers will point to the address of the last element moved plus the corresponding stride.

4.4. FPU/SPRINT Instructions

This group of instructions controls the CM2 FPU, which consists of a floating point chip and the CM2 SPRINT chip.

You will need to understand the architecture of the SPRINT and FPU chips before using these instructions. A brief summary of the SPRINT architecture follows. The programming of the FPU varies depending on the FPU used in your system. Programming manuals for the various FPU's are available from Thinking Machines.

4.4.1. Instruction Naming

Instructions generally have names derived from a concatenation of the things they do within the pipe. "MRTA-%P2" means "Memory Reads Transposer A using pointer register %P2 for the address". "FWTC" means "Float Bus writes transposer C". "DYN" means do dynamic FPU instructions, generally taken from in-line arguments to the macroinstruction. STATUS-L3 means the status transposer is written with the FPU status information resulting from each dynamic instruction at a latency of 3 cycles from the invocation of each dynamic instruction.

You should also be aware of the difference between *striding* instructions and *incrementing* instructions. In instructions which take a memory address in a pointer register (%P*n*) and do a counted number of operations using memory, the pointer register is advanced after each operation by one (incrementing) or the contents of the corresponding increment register, %I*n* (striding). Importantly, on completion, the incrementing instructions always leave the pointer register unchanged from its original value. In contrast, the striding instructions always leave the pointer register advanced past the last memory datum, i.e. effectively advance the pointer register by the count multiplied by the stride in %I*n*.

4.4.2.1. CMIS-FPU-WRITE-CONFIGURATION

This instruction transfers the current contents of the BYPASS register on all SPRINT chips to the SPRINT Configuration RUG Register. The BYPASS register must be preloaded with the desired value using one of the CMIS-FPU-LOAD-BYPASS-xxx instructions. See the SPRINT Chip Specification for details on the use of the Configuration RUG Register. The format of this register follows:

Bits	*Meaning*	*Values*
0	Error enable	0=disable, 1=enable
1	Global enable	0=disable, 1=enable
2	Error on single bit error	0=disable, 1=enable
3	Error on double bit error	0=disable, 1=enable
4	Error on VP error	0=disable, 1=enable
5	Error on float exception	0=disable, 1=enable
6	Signal float exception	0=disable, 1=enable
7-8	Memory size	0=64K Bits
9	Functional test mode	0=disable, 1=enable
10	Parametric test mode	0=disable, 1=enable
11	Even/odd	0=even, 1=odd
12-15	SPRINT serial number	(read only)
16	AB conditional scratch	0=disable, 1=enable
17	FB write conditional	0=disable, 1=enable
18	FB read conditional	0=disable, 1=enable

Note that the AB-CONDITIONAL-SCRATCH, FB-WRITE-CONDITIONAL, and FB-READ-CONDITIONAL fields are also affected managed by the CMIS-FPU-ALWAYS, CMIS-FPU-CONDITIONAL, and CMIS-FPU-CONDITIONAL-WRITE instructions.

4.4.2.2. CMIS-FPU-READ-CONFIGURATION

This instruction transfers the value of the current SPRINT Configuration RUG register to the SPRINT Bypass register in each SPRINT section. The Configuration RUG register is not modified by this operation.

4.4.2.3. CMIS-FPU-STATIC

This instruction performs a single static instruction. The argument is the instruction to be executed:

Argument 1: | Static Instruction |

which can be constructed using the **wtl3132-static-instruction** constructor function (first argument choice is the default):

```
(wtl3132-static-instruction (&optional &key
   :c-port {:ENABLE,:DISABLE}
   :wtl3132-function {:MISCELLANEOUS, :FLOAT-NEGATE-AND-ADD, :FLOAT-SUBTRACT,
                      :FLOAT-ADD, :FLOAT-MULT-NEGATE-AND-ADD,
                      :FLOAT-MULT-NEGATE-AND-SUB, :FLOAT-MULT-AND-ADD})
```

4.4.2.4. CMIS-FPU-STATIC-%A{1-12}

This instruction performs a single static instruction. It functions like CMIS-FPU-STATIC except that the static instruction is taken from the specified %A register. This value can be loaded within the IMP as in:

```
(IMP-LOAD-%A4
  (imp:wtl3132-static-instruction ...))
(CMIS-FPU-STATIC-%A4)
```

or may be calculated on the front end and sent to %A*n* via the IFIFO.

4.4.2.5. CMIS-FPU-DYNAMIC

This instruction performs a single dynamic operation. No status information is collected. The argument is the instruction to be executed:

Argument 1: | Dynamic Instruction |

which can be constructed using the **wtl3132-dynamic-instruction** constructor function:

```
(wtl3132-dynamic-instruction
   :a-addr 0
   :b-addr 0   {n, :LOAD-STATUS-REG, :READ-STATUS-REG, :LOAD-MODE-REG, :FABS
               :FIX-TO-FLOAT, :FLOAT-TO-FIX, :FLUT}
   :c-addr 0
   :mult-b-input :BBUS   {:BBUS, :CBUS}
   :condition-register :DISABLE  {:DISABLE, :ENABLE}
   :io-direction :NOP   {:LOAD, :STORE, :NOP}
   :alu-destination :CBUS   {:TEMP{1-3}-AND-CBUS, :CBUS}
   :alu-b-input :ZERO)   {:CBUS, :BBUS, :TEMP{1-3}, :TWO, :ZERO}
```

4.4.2.6. CMIS-FPU-DYNAMIC-%A{1-12}

This instruction performs a single dynamic instruction. It functions like CMIS-FPU-DYNAMIC except that the dynamic instruction is taken from the specified %A register. This value can be loaded within the IMP as in:

```
(IMP-LOAD-%A4
  (imp:wtl3132-dynamic-instruction ...))
(CMIS-FPU-DYNAMIC-%A4)
```

or may be calculated on the front end and sent to %A*n* via the IFIFO.

4.4.2.7. Loading the WTL3132 Mode Register

The **CMIS-FPU-DYNAMIC** instruction can be used to load the mode register of the WTL3132 FPU using the following **IMPASS** constructor:

```
(wtl3132-load-mode-reg-dynamic-instruction
   :internal-a-c-bypass 0
   :fix-rounds-to-neg-inf 0
   :input-bypass 1                    ;Required by PARIS
   :output-bypass 0
   :fpex-output 1                     ;Required by PARIS
   :co-processor-load-mode 0          ;Required by hardware, Do not change!
   :fpex-active-high 1                ;Required by hardware, Do not change!
   :double-pump-mode 0                ;Required by hardware, Do not change!
   :internal-b-c-bypass 0
   :y-late-input-mode 0)              ;Required by hardware, Do not change!
```

Note that certain fields are required for proper operation of PARIS. You must restore these bits to their default values. before using PARIS instructions. Bits marked as being required by the hardware should never be changed.

For example, the following code fragment alters the mode register to cause the FIX function to round towards negative infinity, rather than to the nearest integer:

```
(CMIS-FPU-STATIC
   (imp:wtl3132-static-instruction :wtl3132-function :miscellaneous :c-port :disable))
(CMIS-FPU-DYNAMIC
   (wtl3132-load-mode-reg-dynamic-instruction :fix-rounds-to-neg-inf 0))
```

Note that the WTL3132 requires that no arithmetic operations be in progress when the mode register is written. As a result there may be cases where an IMP-NOP instruction or two are required before writing the mode register. We have also seen cases where a few IMP-NOP instructions after writing the mode register seemed to make a difference. It is probably a good idea to setup the mode register early and in a separate IMP to avoid timing problems.

4.4.3.1. CMIS-FPU-RESET-TRANSPOSERS

This instruction sets all transposer pointers (including the STATUS transposer) to zero. As a side effect, the BYPASS register is also set to zero.

It is a good idea to use this instruction if any PARIS instructions have been called prior to the execution of your IMP. PARIS does not guarantee the transposer pointers to be set to any particular value after execution of its instructions.

There are no arguments to this instruction.

4.4.3.2. CMIS-FPU-WRITE-POINTERS

This instruction transfers the current contents of the BYPASS register on all SPRINT chips to the SPRINT Pointer RUG Register. This register contains the transposer pointers for the data transposers (but not the status transposer). The BYPASS register must be preloaded with the desired value using one of the CMIS-FPU-LOAD-BYPASS-xxx instructions. See the SPRINT Chip Specification for details on the use of the Pointer RUG Register. The format of this register follows:

Bits	*Meaning*
0-4	Transposer A Pointer
5-9	Transposer B Pointer
10-14	Transposer C Pointer
15-19	Transposer D Pointer

Note that this register is cleared by the CMIS-FPU-RESET-TRANSPOSERS instruction (which also resets the Status Transposer pointer).

4.4.3.3. CMIS-FPU-READ-POINTERS

This instruction transfers the value of the current SPRINT Pointer RUG register to the SPRINT Bypass register in each SPRINT section. The Pointer RUG register is not modified by this operation.

4.4.3.4. CMIS-FPU-WRITE-STATUS-POINTER

This instruction transfers the current contents of the BYPASS register on all SPRINT chips to the SPRINT Status Pointer RUG Register. This register contains the transposer pointer for the status transposer. The BYPASS register must be preloaded with the desired value using one of the CMIS-FPU-LOAD-BYPASS-xxx instructions. See the SPRINT Chip Specification for details on the use of the Status Pointer RUG Register. The format of this register follows:

Bits	*Meaning*
0-4	status transposer pointer
5-31	unused

Note that this register is cleared by the CMIS-FPU-RESET-TRANSPOSERS instruction.

4.4.3.5. CMIS-FPU-READ-STATUS-POINTER

This instruction transfers the value of the current SPRINT Status Pointer RUG register to the SPRINT Bypass register in each SPRINT section. The Status Pointer RUG register is not modified by this operation.

4.4.3.6. CMIS-FPU-ALWAYS

This instruction conditions the SPRINT chip for ALWAYS operation, specifically it sets the SPRINT rug bits to ignore the condition register when performing Float Bus transfers.

Note that the execution of PARIS instructions is not guaranteed to leave the SPRINT configuration register in any particular state. As a result, it is a good idea to issue this instruction (or CMIS-FPU-CONDITIONAL below) at the start of any IMP which will operate the SPRINT chip.

There are no arguments to this instruction.

4.4.3.7. CMIS-FPU-CONDITIONAL

This instruction conditions the SPRINT chip for conditional (contextualized) operation. Specifically, the SPRINT rug bits are setup to perform conditionalized Float Bus transfers, conditionalized indirect addressing transfers, and the SPRINT Bypass register is loaded with zeros. To use conditional Float Bus transfers or conditionalized Indirect Addressing, the SPRINT Condition Register must also be loaded using the CMIS-FPU-MWC[-%P*n*] instructions, described below.

4.4.3.8. CMIS-FPU-CONDITIONAL-WRITE

This instruction *partially* conditions the SPRINT chip for conditional (contextualized) operation. Specifically, the SPRINT rug bits are setup to perform conditionalized Float Bus to SPRINT transfers (Float Bus writes), but *not* SPRINT to Float Bus transfers (Float Bus reads) or conditionalized indirect addressing transfers. The SPRINT Bypass register is loaded with zeros. To use conditional Float Bus transfers, the SPRINT Condition Register must also be loaded using the CMIS-FPU-MWC[-%P*n*] instructions, described below.

4.4.3.9. CMIS-FPU-MWC[-%P{1-4}]

This instruction is used during conditional sprint operations which transfer data through the transposers. The effect of the instructions is to load a memory slice into the SPRINT Condition Register. In the CMIS-FPU-MWC instruction, the slice is taken from the memory location pointed to by the context flag base register %CTX-BASE, i.e. is the current value of the PARIS context flag. For the "-%P*n*" instructions, the slice is taken from the memory location pointed to by the specified Pointer register.

4.4.3.10. CMIS-FPU[-M{R,W}T{i}-%P{1-4}[-STRIDE]][[-FRT{j}]-DYN[-STATUS-L{l}]]

These instructions simultaneously do the following:

1. [optional] N slices of memory data are written into, or read-from, transposer *i* using the memory address stored in %P*n* incrementing by one or a stride in %I*n*,
2. N 32-bit words are read from a different transposer *j* onto the float bus (optional),
3. N Dynamic Floating Point Unit (FPU) instructions are executed (optional), and
4. Status is collected from the FPU after a latency of *l* cycles.

As a result of the optional functions, the following basic instruction groups result:

(1)	**CMIS-FPU-M{R,W}T{A-D}-%P{1-4}[-STRIDE]**	Read/Write transposer to/from memory
(2)	**CMIS-FPU-DYN**	Execute N dynamic instructions
(3)	**CMIS-FPU-FRT{j}-DYN[-STATUS-L{l}]**	Read trans onto Float Bus, do N dynamic instrs

and combinations of (1) with either (2) or (3).

The first argument to these instructions is always the loop count. Note that when loading transposers, a loop count of 32 is often necessary. If FPU dynamic instructions are to be executed, the remaining COUNT arguments are the dynamic instructions themselves.

Argument 1:	Count (Minimum *l*-1 if status, otherwise 1)
Arguments 2-(N+1):	Dynamic Instruction
	. . .

When data is transferred between memory and a transposer, the Pointer Register %P*n* is used as the base memory address. In the -STRIDE versions of the instructions, %P*n* is incremented by the a stride taken from the Increment Register %I*n*, and %P*n* is left incremented beyond the last datum (i.e. the instruction effectively increments %P*n* by %I*n* times the count argument. In the non-striding versions, %P*n* is incremented by one each time, and the contents of the pointer register %P*n* is *not modified by*

the instruction.

The **IMPASS** argument constructor **wtl3132-dynamic-instruction** (see above) can be used to construct dynamic instructions for the Weitek 3132.

The constructor function **wtl3132-dynamic-instruction-up** can be used to generate N such instructions, with the register fields (A-ADDR, B-ADDR, and C-ADDR) incremented by a stride.

```
(wtl3132-dynamic-instruction-up
   :count 32
   :a-addr 0  :a-addr-inc 1
   :b-addr 0  :b-addr-inc 1
   :c-addr 0  :c-addr-inc 1
   :mult-b-input :BBUS
   :condition-register :ENABLE
   :io-direction :NOP
   :alu-destination :CBUS
   :alu-b-input :ZERO)
```

4.4.3.11. CMIS-FPU-M{R,W}T{i}-%P{1-4}[-STRIDE]-%C4

These instructions transfer a counted number of values between a specified transposer and memory. The "MWT*i*" instructions write transposer *i* with data taken from memory, and the "MRT" instructions read data from transposer *i* and write it to memory.

The address at which memory is accessed is taken from a Pointer register (%P1 through %P4). If "-STRIDE" is specified in the name, the address is incremented by the contents of the corresponding Increment register, %I*n*, after each memory access, and at the end, %P*n* is left incremented beyond the last datum (i.e. the instruction effectively increments %P*n* by %I*n* times the count. If "-STRIDE" is not given, this address is incremented by 1 after each access, and %P*n* is left unchanged after completion of the instruction.

The count is taken from the %C4 register and must be greater than 0.

These instructions require no arguments.

4.4.3.12. CMIS-FPU[-M{R,W}T{i}-%P{1-4}[-STRIDE]]-FWT{j}-DYN

These instructions simultaneously perform the following operations:

1. N slices of memory data are written into, or read-from, transposer *i* using the memory address stored in %P*n*, which is either incremented by one or by a stride in %I*n*,
2. N Dynamic Floating Point Unit (FPU) instructions are executed,
3. N 32-bit words are read from Float Bus into transposer *j* The Float Bus timing is suitable for executing STORE operations on the FPU, but is *not* suitable for using the "Store Bypass" feature available on the Weitek FPUs.

The transposer read or written from memory cannot be the same as, or the companion of, the transposer written from the Float Bus:

Transposer j	*Transposer i can be:*
A	B or D
B	A or C
C	B or D
D	A or C

It is important to note that when writing a transposer from the FPU, the transposer pointers of that transposer *and its companion* are both advanced. Thus, the instruction CMIS-FPU-FWTA-DYN with a count of N will advance the pointers of *both* transposers A and C by N.

Arguments are as in the previous FPU instructions. Since the dynamic instructions must be done, these arguments must be present.

Argument 1:	Count (minimum 3)
Arguments 2-(N+1):	Dynamic Instruction
	. . .

When data is transferred between memory and a transposer, the Pointer Register %P*n* is used as the base memory address. In the -STRIDE versions of the instructions, %P*n* is incremented by the stride taken from the Increment Register %I*n*, and %P*n* is left incremented beyond the last datum (i.e. the instruction effectively increments %P*n* by %I*n* times the count argument. In the non-striding versions, %P*n* is incremented by one each time, and the pointer register %P*n* is not modified by the instruction.

4.4.4. Fieldwise Instructions for Double Precision

The following instructions support the use of floating point units that support double precision. In particular, the ability to present operands to the float bus in two parts is provided.

4.4.4.1. CMIS-FPU[-M{R,W}T{i}-%P{1-4}[-STRIDE]]-FRT{j}{k}-DYN2[-STATUS-L3]

These instructions simultaneously do the following (N is the value of the *count* argument):

1. [optional] N slices of memory data are written into, or read-from, transposer *i* using the memory address stored in %P*n* incrementing by one or a stride in %I*n*,
2. N 32-bit words are read from transposer *j* and N 32-bit words are read from transposer *k*, and are written onto the float bus in an alternating fashion (transposer *j*, then *k*, then *j*, and so forth).
3. 2N Dynamic Floating Point Unit (FPU) instructions are executed, and
4. [optional] Status is collected from the FPU after a latency of *l* cycles from every other dynamic instruction, i.e. As a result, N status values are loaded into the status transposer.

The first argument to these instructions is always the loop count (N not 2N). Note that when loading transposers from memory, a loop count of 32 is often necessary. If FPU dynamic instructions are to be executed, the remaining COUNT * 2 arguments are the dynamic instructions themselves.

Argument 1:	Count (minimum 1, 2 if status)
Arguments 2 through (2N+1):	Dynamic Instruction
	. . .

When data is transferred between memory and a transposer, the Pointer Register %P*n* is used as the base memory address. In the -STRIDE versions of the instructions, %P*n* is incremented by the stride taken from the Increment Register %I*n*, and %P*n* is left incremented beyond the last datum (i.e. the instruction effectively increments %P*n* by %I*n* times the count argument. In the non-striding versions, %P*n* is incremented by one each time, and the contents of the pointer register %P*n* is *not modified by the instruction.*

The constructor function **wtl3164-dynamic-inst-up-2** can be used to generate 2N "interwoven" instructions for the Weitek 3164 FPU. This constructor is the same as **wtl3164-dynamic-inst-up** except that there are two sets of instruction field arguments, one for the even numbered instructions and one for the odd numbered instructions.

```
(wtl3164-dynamic-inst-up-2
   :count 32

   :aaddr1  0          ;addresses/strides for 1st, 3rd, 5th, etc, instrs
   :baddr1  0
   :cdaddr1 0
   :efaddr1 0
   :xcnt1 :nop
   :aaddr-inc1 1
   :baddr-inc1 1
   :cdaddr-inc1 1
   :efaddr-inc1 1

   :aaddr2  0          ;addresses/strides for 2nd, 4th, 6th, etc, instrs
   :baddr2  0
   :cdaddr2 0
   :efaddr2 0
   :xcnt2 :nop
   :aaddr-inc2 1
   :baddr-inc2 1
   :cdaddr-inc2 1
   :efaddr-inc2 1)
```

The usual strategy is to use each pair of dynamic instructions to load one value (usually into the X or Y register) and then start an operation that uses the X or Y register as one operand. Note that the first dynamic instruction is intended only to load one half of the X or Y register, yet will have the side effect of starting an arithmetic operation.

For example, the following instruction transfers 8 double-precision (64-bit) data values whose low halves (LSH's) are stored in transposer B and whose high halves (MSH's) are stored in transposer C onto the Float Bus and adds them to registers 8 through 15 (WTL3164 FPU is assumed):

```
(CMIS-FPU-STATIC
   (wtl3164-static-inst
      ;;Set up for Rcd <- Ra + Y
      :func :FADD-SUB :aain :AADD :abin :Y :main 0 :mbin 0))
(CMIS-FPU-FRTBC-DYN2
   (wtl3164-dynamic-inst-up-2
      :count 8
                                          ;; EVEN instructions load low half of Y, do dummy operation to
      ;; same register as next ODD instruction (will be overwritten)
      :cdaddr2 8  :cdaddr-inc2 1          ;dummy result register
      :xcnt1 :LOAD-LSH-Y
                                          ;; ODD instructions load high half of Y, then do Rcd <- Ra op Y
      :aaddr2 8  :aaddr-inc2 1
      :cdaddr2 8  :cdaddr-inc2 1          ;real result register
      :xcnt2 :LOAD-MSH-Y))
```

4.4.5. Fieldwise Instructions for Weitek WTL3164 Divide and Square Root

The following instructions are designed specifically for operating on fieldwise data using the WTL3164 floating point chip. These instructions *will not work* in systems having WTL3132 floating point chips, as they assume hardware changes associated with boards populated with WTL3164's.

4.4.5.1. CMIS-FPU[-M{R,W}T{i}-%P{1-4}]-FRT{j}-DYN-DELAY-%C3-STATUS-L3

This set of instructions allows the piped execution of multiple single-precision divide or square root instructions on the Weitek WTL3164 chip. The instruction repeats COUNT dynamic instructions taking Float Bus data from a given transposer. The arguments to these instructions follow:

Argument 1:	Count (N, minimum 2 if status, otherwise 1)
Arguments 2 through (2N+1):	Dynamic Instruction
	. . .

The resulting pipeline is as follows. D is the delay, taken from %C3:

Number of Cycles	*Operation*
1	Dynamic Instruction 1
D	Delay
1	Dynamic Instruction 2
D	Delay, optionally collect status from Dyn Instr 1
1	Dynamic Instruction 3
D	Delay, optionally collect status from Dyn Instr 2
etc.	

The delay inserted after each NOP is designed to account for the time which the given instruction (assumed to be a divide or square root operation) is in the DSR unit of the chip. The number of delay cycles must be loaded into %C3 prior to the execution of the instruction. The following table gives the proper delay times for the single precision DSR operations:

Operation	*Delay*
Single Precision Divide	8
Single Precision Square Root	14

4.4.5.2. CMIS-FPU[-M{R,W}T{A}-%P{1-4}-FRT{B}{C}-DYN2-DELAY-%C3-STATUS-L3

This set of instructions allows the piped execution of multiple double-precision divide or square root instructions on the Weitek WTL3164 chip. The arguments to these instructions are as follows:

Argument 1:	Count (N, minimum 2 if status, otherwise 1)
Argument 2:	Static Instruction A
Argument 3:	Static Instruction B
Arguments 4 through (2N+3):	Dynamic Instruction
	. . .

The instruction repeats COUNT pairs of dynamic instructions with data written onto the Float Bus from two transposers (currently only transposer B and C) in an alternating fashion. The first dynamic instruction is preceded by a constant static instruction (Static Instruction A), and the second by Static Instruction B. The following pipeline results. D is the delay, taken from %C3:

Number of Cycles	*FBus Data*	*Operation*
1		Static Instruction A
1	Transposer B	Dynamic Instruction 1a
1		Static Instruction B
1	Transposer C	Dynamic Instruction 1b
D	Delay	
1		Static Instruction A
1	Transposer B	Dynamic Instruction 2a
1		Static Instruction B
1	Transposer C	Dynamic Instruction 2b
D	Delay, Collect Status from Dynamic Instruction 1b	

etc.

It is important to note that the WTL3164 requires that the first dynamic instruction of each pair *not* start an arithmetic operation. This has been verified experimentally. As a result, the first (A) dynamic instruction and its corresponding static instruction must specify a NOP. To do this, Static Instruction A must specify FUNCTION=MONADIC and all input operand fields (aain, abin, main, mbin) as zero. The dynamic instruction must specify BADDR=0. If the **wtl3164-dynamic-inst-up-2 argument constructor is used, be sure to set BADDR-INC1 to 0 to prevent the BADDR from being incremented.**

The delay inserted after each NOP is designed to account for the time which the given instruction (assumed to be a divide or square root operation) is in the DSR unit of the chip. The number of delay cycles must be loaded into %C3 prior to the execution of the instruction. The following table gives the proper delay times for the single precision DSR operations:

Operation	*Delay*
Double Precision Divide	14
Double Precision Square Root	28

4.4.6. Slicewise (Bypass) Instructions

The following instructions generally support the use of the SPRINT and FPU chips (all varieties) when slicewise memory data is used.

4.4.6.1. CMIS-FPU-LOAD-BYPASS-CONSTANT

This instructions copies a 32-bit constant into the SPRINT bypass register.

This instruction takes a single argument, the value to be loaded.

Argument 1: | 32-bit Value |

4.4.6.2. CMIS-FPU-CLEAR-BYPASS

This instructions write a zero (32-bit) into the SPRINT bypass register.

This instruction takes no arguments.

4.4.6.3. CMIS-FPU-LOAD-BYPASS-DP1H

This instructions the hex value 3FF00000 (the upper half of a double-precision floating point value 1.0) into the SPRINT bypass register.

This instruction takes no arguments.

4.4.6.4. CMIS-FPU-LOAD-BYPASS-SP1

This instructions the hex value 3F800000 (the single-precision floating point value 1.0) into the SPRINT bypass register.

This instruction takes no arguments.

4.4.6.5. CMIS-FPU-LOAD-BYPASS-%A{1-12}

This instruction loads the bypass register with the 32-bit value in %A*n*.

There are no arguments to this instruction.

4.4.6.6. CMIS-FPU-FWB-DYNAMIC

This instruction performs a single FPU instruction (presumably a STORE operation), and 2 cycles later transfers the data on the Float Bus to the bypass register. The data may then be accessed using instructions that read the bypass register, such as CMIS-FPU-MRB-%P*n*.

This instruction takes a single argument, the FPU dynamic instruction to be executed.

Argument 1:	Dynamic Instruction

4.4.6.7. CMIS-FPU-MWB-%P{1-4}[-STRIDE]

This instructions reads the 32-bit slice in memory at the address contained in %P*n* and copies it to the SPRINT bypass register.

There are no arguments to these instructions.

The -STRIDE version of the instruction group causes the designated pointer register to be incremented by its corresponding increment register (for example, %P3 is incremented by %I3 and so forth). Incrementing is done *after* the transfer.

4.4.6.8. CMIS-FPU[-MWB-%P{1-4}-STRIDE]-FRB-DYN[-%C4]

These instructions move slicewise data from addresses in memory at regular intervals to the float bus (a one-cycle pipeline) and execute dynamic FPU instructions in phase with their arrival on the float bus.

Memory addresses in %P*n* are incremented by the contents of %I*n* each time, allowing striding of the stored data.

Loading of the bypass register from memory is optional. In the versions of the instructions where bypass writing is not done, e.g. in the instruction CMIS-FPU-FRB-DYN, the current bypass register contents are written to the float bus each time. The CMIS-FPU-LOAD-BYPASS-%A*n* or CMIS-FPU-MWB-%P*n* instructions can be used to write values into the bypass register beforehand.

The -%C4 versions of the instruction take their loop count (the number of instructions done) from the register %C4. Note that in these instructions, the *count* argument is still present and indicates the number of dynamic instruction arguments which follow it. *Count* must be large enough to accommodate the largest possible value in %C4 so that sufficient dynamic instructions are available.

The minimum value for the count (argument or %C4) is the status latency (or 0 if not collecting status), plus 2. If memory is not written to the bypass register (i.e. it is used as a constant value), the minimum count is the status latency plus one.

Arguments are as in the previous FPU instructions.

Argument 1:	Count (minimum is L+2, L+1 if no memory transfer - see text)
Arguments 2-(N+1):	Dynamic Instruction
	. . .

4.4.6.9. CMIS-FPU-MWB-STRIDE{2,3,4}-FRB-DYN[-%C4]

These instructions move slicewise data from memory locations at 2, 3 or 4 areas of memory to the float bus (a one-cycle pipeline) and execute dynamic FPU instructions in phase with their arrival on the float bus. The memory areas are pointed to by the registers %P1 through %P*n* (*n* = 2, 3, or 4) and data is taken from the areas in strides taken from the corresponding registers %I1 through %I*n*.

These instructions differ from the previously described -DYN- counterparts in that they execute 2, 3 or 4 dynamic instructions in each pass of the loop, one for each of the pointers %P1 through %P*n*. Thus in the -DYN3- versions of the instruction, a count of 10 yields 30 dynamic instructions.

The resulting algorithm is (based on n = 2, 3, or 4):

```
Do count times:
  begin
    float bus = memory[%P1] and do dynamic instruction;
    %P1 = %P1 + %I1;
     ...
    float bus = memory[%Pn] and do dynamic instruction;
    %Pn = %Pn + %In
  end;
```

The -%C4 versions of the instructions take their loop count (the number of outer loops done) from the register %C4. Note that in these instructions, the *count* argument is still present and indicates the number of dynamic instruction arguments which follow it. *Count* must be large enough to accommodate the largest possible value in %C4 multiplied by *n* (2, 3, or 4) so that sufficient dynamic instructions are available.

Note: The *count* value, taken from the argument or from %C4) may not be less than 3.

Arguments are as in the previous FPU instructions. Since the dynamic instructions must be done, these arguments must be present.

Argument 1:	Count (minimum 3)
Arguments 2-(N+1):	Dynamic Instruction
	. . .

4.4.6.10. CMIS-FPU-MRB-%P{1-4}[-STRIDE]

These instructions copies the contents of the SPRINT bypass register and writes them to the memory slice whose address is in %P*n*.

There are no arguments to these instructions.

The -STRIDE version of the instruction group causes the designated pointer register to be incremented by its corresponding increment register (for example, %P3 is incremented by %I3 and so forth). Incrementing is done *after* the transfer.

4.4.6.11. CMIS-FPU-MRB-%P{1-4}-STRIDE-FWB-DYN[-%C4][-GENERAL]

These instructions execute N dynamic instructions, and move float bus data slice wise into memory at regular intervals starting at the address contained in %P*n*. Float Bus timing is set up to be suitable for STORE instructions to the FPU, but *not* for the "Store Bypass" feature of the Weitek WTL3132 and WTL3164 FPU's.

Memory addresses in %P*n* are incremented by the contents of %I*n* each time, allowing striding of the stored data.

Argument 1:	Count (unless -GENERAL, minimum is 4)
Arguments 2-(N+1):	Dynamic Instruction
	. . .

The -%C4 versions of the instruction take their loop count (the number of instructions done) from the register %C4. Note that in these instructions, the *count* argument is still present and indicates the number of dynamic instruction arguments which follow it. *Count* must be large enough to accommodate the largest possible value in %C4 so that sufficient dynamic instructions are available.

If the -GENERAL is omitted in the name, the count, whether taken from the argument or from %C4, is limited to a minimum value of four. Violation of this restriction will result in CM microcode errors. If -GENERAL is given in the name, the count can be any value (including zero), but the instructions will take up to 5 cycles longer.

4.4.6.12. CMIS-FPU-MWB-%P{1-4}-STRIDE-FRT{A-D}-DYN[-%C4]

These instructions move slicewise data from addresses in memory at regular intervals to the Bypass register, and simultaneously read data from a specified transposer onto the Float Bus while performing FPU dynamic instructions.

The usefulness of this instruction only becomes apparent when you consider how conditional float bus reads work on the SPRINT chip. Conditional float bus reads are enabled for the SPRINT chip via the CMIS-FPU-CONDITIONAL instruction.

Recall that when conditionally reading a transposer onto the Float Bus, the data is read instead from the BYPASS register for *deselected* processors. Processor selection is controlled by bit *k* of the transposer pointer of the transposer being read. The Condition Register is written via the CMIS-FPU-MWC or CMIS-FPU-MWC-%P*n* instructions.

As a result, if the Condition Register is all zeroes, these instructions becomes slicewise pipelines of data through the Bypass Register, identical to the instructions CMIS-FPU-MWB-%P*n*[-STRIDE]-FRB-DYN[-%C4]. In those transfers where the transposer pointer equals *k* and bit *k* of the Condition Register contains a one, the data is taken instead from the specified transposer.

By loading the Condition Register with the inverse of the 32 processor context bits, and filling the given transposer with an appropriate value for *deselected* processors, this instruction effectively implements a "conditional slicewise" FPU pipeline.

Memory addressing is the same as the other slicewise FPU pipelines, i.e. Memory addresses in %P*n* are incremented by the contents of %I*n* each time, allowing striding of the stored data. After completion, %P*n* is left pointing to the last transferred datum plus the stride, thus is effectively advanced by the stride times the count.

The -%C4 versions of the instruction take their loop count (the number of instructions done) from the register %C4. Note that in these instructions, the *count* argument is still present and indicates the number of dynamic instruction arguments which follow it. *Count* must be large enough to accommodate the largest possible value in %C4 so that sufficient dynamic instructions are available.

The minimum value for the count (argument or %C4) is 2.

Argument 1:	Count (minimum is 2)
Arguments 2-(N+1):	Dynamic Instruction
	. . .

4.4.7. Slicewise Instructions for Double Precision

The following instructions are designed specifically for operating on slicewise data using the WTL3164 floating point chip. These instructions *will not work* in systems having WTL3132 floating point chips, as they assume hardware changes associated with boards populated with WTL3164's.

4.4.7.1. CMIS-FPU-MWB-%P{1-4}-STRIDE-FRB-DYN2[-%C4]

These instructions move a counted number of pairs of 32-bit slices through the bypass register onto the FPU float bus, executing dynamic instructions upon the arrival of each datum.

It is presumed that each pair of data consists of the two halves of a 64-bit value, stored consecutively in memory. It is traditional that double precision floats be stored low half followed by high half. Therefore, striding of %P*n* is done alternately by one and by %I*n*. As a result, %I*n* must be preloaded with the stride between each pair of numbers in memory, minus one.

The count (the number of pairs) is taken from an inline argument. This count cannot be less than 2. The remaining arguments are COUNT*2 dynamic instructions constructed for the appropriate FPU chip.

In the "-%C4" versions, the count (number of pairs) is taken from %C4. In this case, the *count* argument is the number of dynamic instruction arguments given (not the number of pairs!), of which the first 2 * %C4 will be executed. These instructions are slower than the ones with the inline counter.

Argument 1:	Count (minimum 2)
Arguments 2-(N+1):	Dynamic Instruction 1a
Arguments 2-(N+1):	Dynamic Instruction 1b
Arguments 2-(N+1):	Dynamic Instruction 2a
	. . .

4.4.7.2. CMIS-FPU-MRB-%P{1-4}-STRIDE-FWB-DYN2[-%C{1-4}[-GENERAL],-1,-2]

These instructions execute a counted number of dynamic instructions (presumably STORE instructions) and move the data output by the FPU on the float bus to memory as 32-bit slices. A latency of 2 cycles is assumed between each dynamic instruction and the arrival of data on the float bus (correct for the WTL3164 chip running in delayed-data-store mode).

It is presumed that each pair of data consists of the two halves of a 64-bit value, stored consecutively in memory. It is traditional that double precision floats be stored low half followed by high half. Therefore, striding of %P*n* is done alternately by one and by %I*n*. As a result, %I*n* must be preloaded with the stride between each pair of numbers in memory, minus one.

The count (the number of pairs) is taken from an inline argument. This count cannot be less than 3. The remaining arguments are COUNT*2 dynamic instructions constructed for the appropriate FPU chip.

In the "-%C4" versions, the loop count (number of pairs) is taken from %C4. In this case, the *count* argument is the number of dynamic instruction arguments given (not the number of pairs!), of which the first 2 * %C4 will be executed. These instructions are slower than the ones with the inline counter. A minimum count of 3 is allowed, unless the "-%C4-GENERAL" instructions are used, in which case any count is allowed at the cost of up to 3 additional cycles.

The version of these instructions with "-1" or "-2" at the end use a fixed count of 1 or 2. These are required since the inline-counter versions have a minimum count of 3. These instructions take no *count*

argument, but require only two ("-1") or four ("-2") dynamic instruction arguments.

Argument 1:	Count (minimum 3) Omit in "-1" and "-2" cases
Arguments 2-(N+1):	Dynamic Instruction 1a (store)
Arguments 2-(N+1):	Dynamic Instruction 1b (store)
Arguments 2-(N+1):	Dynamic Instruction 2a (store)
	. . .

4.4.8. Instructions for Reading the Status Transposer

When -STATUS-L*n* variants of the Floating Point Unit pipelines are used, status information is collected from the FPU and stored in the SPRINT Status Transposer. There are 5 status bits, an exception pin (FPEX) and 4 status bits, S0, through S3.

The meaning of the 4 status pins varies with the FPU type. For the Weitek WTL3164 double precision FPU, the status pins are encoded as shown in the WTL3164 manual in Figure 55 on page 63. On the WTL3132, only status pins S0 and S1 are used. S0 is the FPCN pin output which gives comparison results (when the ENCN instruction field requests comparison output), and S1 is the ZERO pin which indicates a zero result.

As a result, only a few memory slices read from the status transposer have meaning. Furthermore, if less than 32 operations are performed yielding status, only that many low order bits of the slices will contain valid status.

Instructions are provided for reading information out of the Status Transposer and saving the resulting information into the memory slices pointed to by the flag base registers.

Note: The WTL3132 signals FPEX on an underflow from Adds and Subtracts (but not Multiplies), so you should be careful when using the FPEX signal to indicate overflows. In the event of a signalled error, you can look at the exponent of the result. If the exponent is zero, it was an underflow which caused FPEX, otherwise it was an overflow.

4.4.8.1. CMIS-FPU-MRTS-{CONDITIONAL,ALWAYS}{[-STICKY]-%OFL-BASE,%TST-BASE,%P1}[-ADV]

These instructions read a single slice from the Status Transposer ("TS") and write the slice to the memory location pointed to by the specified flag base register (or %P1). The status transposer's pointer is unaffected unless the -ADV variant is used, in which case the pointer is advanced. In no case is the flag base register advanced. If the -STICKY variant is used, the slice is bitwise OR'ed into the same memory location instead, thus implementing a "sticky" flag, i.e. one which remains set once set.

4.5. Indirect Addressing Instructions

The SPRINT chip has the ability to read or write its devices to/from memory using the contents of a transposer as the memory addresses. This mechanism is referred to as *SPRINT Indirect Addressing* In a sequence of N transfers, N *offsets* are taken from a transposer, compared to a Bounds Register (an error occurs if the offset exceeds the bounds), and then added to a Base Register. The resulting address is used to store the data, located in another transposer or the Bypass Register, to or from memory.

The following instructions support the use of this mechanism from CMIS:

4.5.1. CMIS-FPU-CLEAR-T{A-D}[[-REST]-%C4]

This instruction writes *count* zeros to the specified transposer. Without the "-%C4", the instructions take *count* in a single argument. In the "-%C4" versions, count is taken from the %C4 register. In the "-REST-%C4", the count is derived from the difference between 32 and the contents of %C4, i.e.

count = 32 - %C4

A count of 32 clears the entire transposer. A side effect of these instructions is that the transposer pointer for the specified transposer is advanced by the count.

Argument 1: | Count (minimum 1) |

An example usage is when loading offsets into the transposer for use in indirect addressing and the offsets are taken from processor-wise fields of length *J* bits. If *J* is less than 32, following the loading of the index fields, 32 - *J* zeros must be loaded into the transposer. Assuming that *J* is in %C4, the CMIS-FPU-CLEAR-Ti-REST-%C4 instruction can be used to do this.

4.5.2. CMIS-IA-LOAD-BASE

This instruction writes the contents of the bypass register into the SPRINT Indirect Addressing Base Address Register. There are no arguments.

4.5.3. CMIS-IA-LOAD-SCRATCH

This instruction writes the contents of the BYPASS register into the SPRINT Indirect Addressing Scratch Address Register. This address receives data stored to memory via the indirect addressing mechanism when the condition bit is off (deselected). There are no arguments.

4.5.4. CMIS-IA-LOAD-BOUND

This instruction writes the contents of the BYPASS register into the SPRINT Indirect Addressing Bounds Register. This register is compared against the indirect address offset during each indirect load or store. There are no arguments.

4.5.5. CMIS-IA-MRT{j}-INDIRECT-T{i}[-%C4]

This instruction stores transposer *j* to memory using the contents of transposer *i* as the indirect address offsets. COUNT slices are transferred, where COUNT is taken from either %C4 ("-%C4" versions of the instruction) or from the first argument:

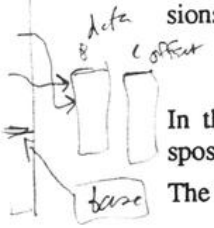
Argument 1: | Count (minimum 2) |

In the current implementation, Transposer *i* (the address transposer) must be Transposer C, and transposer *j* (the data transposer) may be either transposer A or transposer B.

The transposer pointer for transposers *i* and *j* are each advanced by the COUNT.

4.5.6. CMIS-IA-MWT{j}-INDIRECT-T{i}[-%C4]

This instruction loads transposer *j* from memory using the contents of transposer *i* as the indirect address offsets. COUNT slices are transferred, where COUNT is taken from either %C4 ("-%C4" versions of the instruction) or from the first argument:the first argument:



Argument 1:

Count (minimum 2)

In the current implementation, Transposer *i* (the address transposer) must be Transposer C, and transposer *j* (the data transposer) may be either transposer A or transposer B.

The transposer pointer for transposers *i* and *j* are each advanced by the COUNT.

4.5.7. CMIS-IA-MRT{j}-INDIRECT-%Pn[-%C4]

This instruction stores transposer j to memory using the slices stored starting at %Pn as the indirect address offsets. COUNT slices are transferred, where COUNT is taken from either %C4 or from the first argument.

Any of transposers A, B, or C can be used as the data transposer. The transposer pointer for transposer j is advanced by COUNT.

Argument 1:

Count (minimum is 2) - omit in -%C4 versions
. . .

It is important that this instruction *not* be used with conditionalization enabled in the SPRINT chip, i.e. you should always execute the CMIS-FPU-ALWAYS instruction before using this instruction.

4.5.8. CMIS-IA-MWT{j}-INDIRECT-%Pn[-%C4]

This instruction loads transposer j from memory using the slices stored starting at %Pn as the indirect address offsets. COUNT slices are transferred, where COUNT is taken from either %C4 or from the first argument.

Any of transposers A, B, or C can be used as the data transposer. The transposer pointer for transposer j is advanced by COUNT.

Argument 1:

Count (minimum is 2) - omit in -%C4 versions
. . .

It is important that this instruction *not* be used with conditionalization enabled in the SPRINT chip, i.e. you should always execute the CMIS-FPU-ALWAYS instruction before using this instruction.

4.5.9. CMIS-IA-MRB-INDIRECT-T{i}[-%C4]

This instruction stores the SPRINT Bypass Register to memory using the contents of transposer *i* as the indirect address offsets. COUNT slices (each being the same value) are transferred, where COUNT is taken from either %C4 ("-%C4" versions of the instruction) or from the first argument:

Argument 1:

Count (minimum 2)

In the current implementation, Transposer *i* (the address transposer) may only be transposer C.

The transposer pointer for transposer *i* is advanced by the COUNT.

It is important that this instruction *not* be used with conditionalization enabled in the SPRINT chip, i.e. you should always execute the CMIS-FPU-ALWAYS instruction before using this or any slicewise

indirect addressing instruction.

4.5.10. CMIS-IA-MWB-INDIRECT-T{i}

This instruction loads the SPRINT Bypass Register from memory using the contents of transposer *i* as the indirect address offsets. One slice is transferred. This instruction has no arguments.

In the current implementation, Transposer *i* (the address transposer) may be transposer A, B, or C.

The transposer pointer for transposer *i* is advanced by one.

It is important that this instruction *not* be used with conditionalization enabled in the SPRINT chip, i.e. you should always execute the CMIS-FPU-ALWAYS instruction before using this or any slicewise indirect addressing instruction.

4.5.11. CMIS-IA-GATHER-SLICES-TO-%P{n}-STRIDE-INDIRECT-T{i}[-%C4]

This instruction moves slices. The source slices are indirectly addressed using the contents of transposer i as the offsets and are written to %Pn strided by %In. COUNT slices are transferred, where COUNT is taken from either %C4 or from the first argument.

In the current implementation only transposer C can be used as the address transposer. Its transposer pointer is advanced by COUNT.

Argument 1:

Count (minimum is 2) - omit in -%C4 versions
. . .

It is important that this instruction *not* be used with conditionalization enabled in the SPRINT chip, i.e. you should always execute the CMIS-FPU-ALWAYS instruction before using this or any slicewise indirect addressing instruction.

4.5.12. CMIS-IA-SCATTER-SLICES-FROM-%P{n}-STRIDE-INDIRECT-T{i}[-%C4]

This instruction moves slices. The source slices are taken from %Pn strided by %In and are written indirectly, using the contents of transposer i as the offsets. COUNT slices are transferred, where COUNT is taken from either %C4 or from the first argument.

In the current implementation only transposer C can be used as the address transposer. Its transposer pointer is advanced by COUNT.

Argument 1:

Count (minimum is 2) - omit in -%C4 versions
. . .

It is important that this instruction *not* be used with conditionalization enabled in the SPRINT chip, i.e. you should always execute the CMIS-FPU-ALWAYS instruction before using this or any slicewise indirect addressing instruction.

4.6. Cube Wire Communication Instructions

The following instructions provide some support for communicating data on the cube wires of the CM2. The current functionality provided controls only the the upper 12 dimensions of the hypercube, i.e. those which interconnect the individual CM2 chips. The lower 4 dimensions (within which communication is done on the CM2 chip using the flipper) are not supported. Rather, this functionality is intended to be used in a slicewise model where each section (2 CM2 chips, 1 SPRINT, 1 FPU) is a node in the communication architecture. Since there are two CM2 chips per section, of the twelve, the lowest dimension cube wire connects the two CM2 chips within each section.

In particular, support is given for moving data directly between SPRINT transposers and the cube wires, offering a fast slicewise communication mechanism (slices are transposed, sent, received, and transposed back to memory), as well as a method for communicating simultaneously across multiple cube wires, thus maximizing the communication bandwidth of the machine.

A method by which the transposer can be randomly accessed is also provided so that data can be selectively received and/or shuffled. With that and by using the remainder of the CMIS instruction set, powerful algorithms involving complex on-section and hypercube data layouts are possible.

4.6.1. CMIS-TP-M{R,W}T{A,B}-%P*n*-TABLE-%A{8+*n*}[-STRIDE]

This instruction allows the reading or writing of a transposer randomly using a slicewise table of transposer pointers. A counted number of values are read or written as follows (note: *n* is the Pointer Register number, *x* is A or B):

```
for i = 0 to (count-1)
   begin
      Transposer x pointer <— memory[%An]
      mem[%Pn] <—> Transposer x
      %Pn = %Pn + ( %In or 1 )
   end
```

If -STRIDE is specified, the memory pointer %P*n* is incremented by the stride in %I*n*, and %P*n* is left at the end containing the address of the last slice moved plus the stride. If -STRIDE is not given, the pointer is advanced by one after each transfer and is restored to its initial value at the end (i.e. is unaffected by the instruction).

If -%C4 is given, the count is taken from the %C4 register and the instruction requires no arguments. Otherwise, the count is given as an argument. In either case, the minimum count is two.

Argument 1: | Count (minimum is 2) - omit in -%C4 versions |

4.6.2. CMIS-CUBE-SWAP-TA-TO-T{B,C}

This instruction transfers 24 values loaded from slicewise memory into transposer A across two sets of 12 cube wires and receives the same into transposer B or C.

Assuming that sprint chips are numbered so that sprint chip i is connected to beta chips 2i and 2i+1, and that there are W wires attached to each beta chip, the operation this routine performs is:

```
for all sprint chips i
   for 0 < j < W
      Slots j and j+16 of Transposer A in sprint chip i are written to
        slots j and j+16 of Transposer B in sprint chip (i XOR 2^(j-1))
   Slots 0 and 16 of Transposer A are written to slots 16 and 0 of
    Transposer B in each sprint chip.
```

This instruction takes 26 ms.

4.6.3. CMIS-CUBE-SWAP-TB-TO-TA

This instruction is equivalent to the CMIS-CUBE-SWAP-TA-TO-TB but works in the opposite direction, i.e. transmits data from Transposer B and receives into Transposer A.

4.6.4. CMIS-CUBE-SWAP-%P1-TO-%P2[-%C4]

These instructions perform a cube swap from memory to memory on fieldwise values. This implements the algorithm:

```
for all CM chips c in parallel
  for all processors p in {0 ... NWIRES-1} in parallel
    the value in CM chip c, processor p is sent to CM chip c XOR 2^p,  processor p
```

The source data is the field whose address is in %P1, and the destination is the field whose address is in %P2. Note that for each CM chip, only the first NWIRES processors swap data. NWIRES is the number of cube wires connecting dimensions 4 and above. For example, for a full connection machine, NWIRES is 12. All other processors through processor 15 in each chip receive undefined values.

The field length is given by a single *count* argument, or in the case of the -%C4 variants, in the %C4 register.

Argument 1:	Count (omit in -%C4 versions)

If %P1 and %P2 are on the same DRAM page this takes 0.75 usec per bit in the inner loop, otherwise 1.05 usec per bit. For 32 bits this is 25 usec with no page faults, 34 usec with page faults.

This instruction has the side effect of advancing the transposer B pointer count times (effectively count MOD 32 times).

5. CMIS Examples

The following are a few examples culled from the current PARIS code and from various test code. The intent here is to show sequences of CMIS instructions and some examples of algorithm construction. For syntax and tools usage examples, consult the LISP or C/UNIX IMP Tools Reference Manual of your choice.

5.1. WTL3164 Version of "FADD"

Both the LISP and UNIX Tools manuals give an example of an IMP which does a vectored floating add between two 32-bit physical fields. The IMP from those examples is reproduced here as a CMIS example, and also is shown below in both the WTL3132 and WTL3164 versions

```
# IMP to do a physical floating add on the WTL3164.

#include /usr/include/cm/impmacros.h

$imp "FADD"
      $define length 32

      IMP_RECEIVE_%P1                    # source 1 physical address
      IMP_RECEIVE_%P2                    # source 2 physical address
      IMP_RECEIVE_%P3                    # destination physical address
      CMIS_FPU_ALWAYS                    # setup for ALWAYS operation
      CMIS_FPU_RESET_TRANSPOSERS         # reset transposer_pointers
      # Load SOURCE1 into Transposer A
      CMIS_FPU_MWTA_%P1
        32                               # load transposer A from source1 %P1
      # Move SOURCE1 to the WTL3164 registers, Load SOURCE2 into Transposer B
      CMIS_FPU_STATIC
        wtl3164_static_inst func=MONADIC, AAIN=0, ABIN=0, MAIN=0, MBIN=0
      CMIS_FPU_MWTB_%P2_FRTA_DYN
        32
        wtl3164_dynamic_inst_up count=32, baddr_inc=0, efaddr=0,        xcnt=LOAD_MSH_REG
      # For i=0 to 31: Ri = Ri + Y  (Y loaded from SOURCE2 in Transposer B)
      CMIS_FPU_STATIC
        wtl3164_static_inst func=FADD_SUB, aain=AADD, abin=Y, main=0, mbin=0
      CMIS_FPU_FRTB_DYN
        32
        wtl3164_dynamic_inst_up count=32, aaddr=0, cdaddr=0,        xcnt=LOAD_MSH_Y
      # Move Results to Transposer C
      CMIS_FPU_STATIC
        wtl3164_static_inst func=MONADIC, AAIN=0, ABIN=0, MAIN=0, MBIN=0
      CMIS_FPU_FWTC_DYN
        32
        wtl3164_dynamic_inst_up count=32, baddr_inc=0,        xcnt=STORE_MSH_REG
      # Store Results to DESTINATION
      CMIS_FPU_MRTC_%P3
        32                               # store transposer to dest %P3
      IMP_EXIT

$end
```

5.2. PARIS WTL3164 CM:F-ADD-2-1L IMP

The following IMP implements the PARIS **cm:f-add-2-1l** instruction for the WTL3164 FPU. It assumes %I1 and %I2 are pre-loaded with the VP increments (strides) for the destination and source respectively. It receives two values from the IFIFO - the physical destination and source field addresses, respectively.

Also shown is an example of the use of **defimp**.

```
(DEFIMP WTL3164-DOUBLE-F-ADD-2-1L-IMP (ARG1 ARG2)
 "initialize things"
 (IMP-SAVE-FLAG-BASES-OFL-CTX)
 (CMIS-FPU-RESET-TRANSPOSERS)
 (CMIS-FPU-CONDITIONAL-WRITE)
 (CMIS-FPU-MWC)
 (IMP-MOVE-%V-TO-%C1)
 (IMP-LOAD-%A12 32)
 (IMP-LOAD-%A11 -32)
 (IMP-RECEIVE-%P1 "destination  (C)")
 (IMP-RECEIVE-%P2 "source       (B)")
 "preamble"
 (CMIS-FPU-MWTA-%P1 32)
 (IMP-ADD-%A12-TO-%P1)
 (CMIS-FPU-STATIC (WTL3164-STATIC-INST :FUNC :MONADIC))
 (CMIS-FPU-MWTB-%P1-FRTA-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :XCNT :LOAD-LSH-REG-VIA-F :BADDR :NOP-RECR :BADDR-INC 0))
 (CMIS-FPU-MWTC-%P2-FRTB-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :XCNT :LOAD-MSH-REG-VIA-F :BADDR :NOP-RECR :BADDR-INC 0))
 "vp loop"
 (LABEL VP-LOOP)
 (IMP-ADD-%A12-TO-%P2)
 (CMIS-FPU-MWTB-%P2 32)
 (IMP-ADD-%A11-TO-%P1)
 "the second static-dynamic pair - this is where the floating point ADD is performed"
 (CMIS-FPU-STATIC
  (WTL3164-STATIC-INST :FUNC :DFADD-SUB :AAIN :AADD :ABIN :Y :MAIN 0 :MBIN 0))
 (CMIS-FPU-MWTA-%P1-FRTBC-DYN2-STATUS-L3
  32
  (WTL3164-DYNAMIC-INST-UP-2 :COUNT 32 :XCNT1 :LOAD-MSH-Y :XCNT2 :LOAD-LSH-Y))
 "this is the only guy that needs to be looped around (to prevent p3 from pointing into CM scratch)"
 (IMP-DEC-%C1-JUMP-Z (OFFSET LAST-PASS))
 (IMP-ADD-%I1-TO-%P1)
 "the third static-dynamic pair: TB<-(next)src2, TC<-(Fbus or TA)"
 "this most be done twice - once for lsh and again for msh.  Transposers are only conditional on writes."
 (CMIS-FPU-STATIC (WTL3164-STATIC-INST :FUNC :MONADIC))
 (CMIS-FPU-MWTB-%P1-FWTC-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                           :XCNT :STORE-LSH-REG-VIA-E-PORT))
 (IMP-SUBTRACT-%I1-FROM-%P1)
 (CMIS-FPU-MRTC-%P1-FRTB-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                           :XCNT :LOAD-LSH-REG-VIA-F))
 (IMP-ADD-%A12-TO-%P1)
 (CMIS-FPU-MWTC-%P1 32)
 (IMP-ADD-%I1-TO-%P1)
 (CMIS-FPU-MWTB-%P1-FWTA-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                           :XCNT :STORE-MSH-REG-VIA-E-PORT))
```

```
(IMP-SUBTRACT-%I1-FROM-%P1)
(CMIS-FPU-MRTA-%P1-FRTB-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                  :XCNT :LOAD-MSH-REG-VIA-F))
"now, clean up pointers for next vp bank"
(IMP-ADD-%I1-TO-%P1)
(IMP-ADD-%I2-TO-%P2)
(IMP-ADD-%A11-TO-%P2)
"this guy sets things up for next bank"
(CMIS-FPU-MWTC-%P2 32)
"this guy will read the status transposer to memory.  Sticky means that the flags will be or'ed."
"[%ofl-base<-TS, nil]"
(CMIS-FPU-MRTS-CONDITIONAL-STICKY-%OFL-BASE)
(IMP-ADVANCE-FLAGS-OFL-CTX)
(CMIS-FPU-MWC)
(IMP-JUMP (OFFSET VP-LOOP))
(LABEL LAST-PASS)
"we are in the last vp-bank - do everything except set up for next vp bank"
(CMIS-FPU-STATIC (WTL3164-STATIC-INST :FUNC :MONADIC))
(CMIS-FPU-FWTC-DYN
  32 (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                    :XCNT :STORE-LSH-REG-VIA-E-PORT))
(CMIS-FPU-MRTC-%P1 32)
(IMP-ADD-%A12-TO-%P1)
(CMIS-FPU-MWTC-%P1 32)
(CMIS-FPU-FWTA-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                  :XCNT :STORE-MSH-REG-VIA-E-PORT))
(CMIS-FPU-MRTA-%P1 32)
"OR the FPEX bit from the status into the Overflow Flag"
(CMIS-FPU-MRTS-CONDITIONAL-STICKY-%OFL-BASE)
"restore the flag base pointers"
(IMP-RESTORE-FLAG-BASES-OFL-CTX))
```

5.3. PARIS WTL3164 CM:F-DIVIDE-2-1L IMP

The following IMP implements the PARIS **cm:f-divide-2-1l** instruction. It assumes %I1 and %I2 are pre-loaded with the VP increments (strides) for the destination and source respectively. It receives two values from the IFIFO - the physical destination and source field addresses, respectively.

Of particular note is the operation of the WTL3164 Divide/SQRT (DSR) unit, and the use of the CMIS-FPU-MWTA-%P1-FRTC-DYN-DELAY-%C3-STATUS-L3 instruction to insert the necessary pipeline delays during the DSR recirculation time. The delay used, taken from %C3, is set to 8 at the top of the IMP.

```
;;; Perform fieldwise divide with VP looping. This is essentially CM:F-DIVIDE-2-1L
;;; for single precision (23 8), without the status collection.

(DEFIMP WTL3164-SINGLE-F-DIVIDE-2-1L-IMP (ARG1 ARG2)
 "initialize things"
 (CMIS-FPU-RESET-TRANSPOSERS)
 (CMIS-FPU-CONDITIONAL)
 (CMIS-FPU-MWC)
 (IMP-MOVE-%V-TO-%C1 "put vp ratio into counter 1")
 (IMP-RECEIVE-%P1 "destination (C)")
 (IMP-RECEIVE-%P2 "source      (B)")
 "Load the proper delay for single precision divide"
 (IMP-LOAD-%C3 8)
 "preamble"
 "the first static-dynamic pair: TC<-src1, Fbus<-TB; op: fpu-reg <- Fbus"
 (CMIS-FPU-STATIC (WTL3164-STATIC-INST :FUNC :MONADIC))
 (CMIS-FPU-MWTB-%P1 32)
 (CMIS-FPU-MWTC-%P2-FRTB-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                :XCNT :LOAD-MSH-REG-VIA-F))
 "VP loop"
 (LABEL VP-LOOP)
 "the second static-dynamic pair   ;;;[TA<-dest, Fbus<-TC; op: fpu-reg <- fpu-reg * Fbus]"
 (CMIS-FPU-STATIC (WTL3164-STATIC-INST :FUNC :FDIV :AAIN 0 :ABIN 0 :MAIN :AADD :MBIN :Y))
 (CMIS-FPU-MWTA-%P1-FRTC-DYN-DELAY-%C3-STATUS-L3
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :XCNT :LOAD-MSH-Y))
 (IMP-DEC-%C1-JUMP-Z (OFFSET LAST-PASS))
 "the third static-dynamic pair: TB<-(next)src2, TC<-(Fbus or TA); op: Fbus <- fpu-reg"
 (CMIS-FPU-STATIC (WTL3164-STATIC-INST :FUNC :MONADIC))
 (IMP-ADD-%I1-TO-%P1)
 (CMIS-FPU-MWTB-%P1-FWTC-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                :XCNT :STORE-MSH-REG-VIA-E-PORT))
 (IMP-SUBTRACT-%I1-FROM-%P1)
 "Move results to Transposer C"
 (CMIS-FPU-MRTC-%P1 32)
 (CMIS-FPU-MRTS-CONDITIONAL-STICKY-DIVIDE)
 (CMIS-FPU-MWC)
 (IMP-ADD-%I1-TO-%P1)
 (IMP-ADD-%I2-TO-%P2)
 (CMIS-FPU-MWTC-%P2-FRTB-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                :XCNT :LOAD-MSH-REG-VIA-F))
 (IMP-JUMP (OFFSET VP-LOOP))
 (LABEL LAST-PASS)
 (CMIS-FPU-STATIC (WTL3164-STATIC-INST :FUNC :MONADIC))
 (CMIS-FPU-FWTC-DYN
  32
  (WTL3164-DYNAMIC-INST-UP :COUNT 32 :BADDR :NOP-RECR :BADDR-INC 0
                :XCNT :STORE-MSH-REG-VIA-E-PORT))
```

```
"Move the results to transposer C"
(CMIS-FPU-MRTC-%P1 32)
(IMP-EXIT))
```

5.4. Modulo PI IMP

The following IMP implements the internal Modulo PI function used to precondition arguments for the SIN routine, used by **cm:f-sin-2-1l**. The calculation performed is:

```
result = x - (2 * pi * floor((x/(2*pi)) + .5))
```

This code does VP looping and is equivalent to:

```
(defun paris-single-mod-pi (dest source)
  (with-paris-stack-frame ((div 32) (my-pi 32) (two-pi 32) (scratch 32))
    (move-integer-constant my-pi cml-pi-single 23 8)
    (move-integer-constant two-pi 1/2pi-single 23 8)
    (cm:f-add-3-1l dest source my-pi 23 8)
    (cm:f-multiply-3-1l div dest two-pi 23 8)
    (cm:floor scratch div 32 23 8)
    (cm:f-s-float-2-2l div scratch 32 23 8)
    (move-integer-constant two-pi two-pi-single 23 8)
    (cm:f-multiply-2-1l div two-pi 23 8)
    (cm:f-subtract-2-1l dest div 23 8)
    (cm:f-subtract-2-1l dest my-pi 23 8))
  (values))
```

The IMP requires a 32-bit field in memory for use as a temp, which is allocated on the front-end before the IMP is called. This temp field is per-physical processor only, i.e. need not be allocated one per VP.

The VP loop does identical calculation for each of N "bands", where N is the VP ratio. Within each band, calculations are done 8 at a time, taking 8 source values at a time from transposers B and C. The strategy for storing the results is to write LSH's into transposer A, which is writing out fieldwise to memory after each band. Due to the lack of an available transposer for the MSH's, they are written through the Bypass register slicewise, and then transposed in place.

5.4.1. WTL3132 Single Precision Version

This version of the Modulo Pi IMP is written for doing the calculation in single precision on the WTL3132 FPU.

```
(defimp w32-single-mod-pi (dest source temp)
  (declare (library math-library))
  (imp-receive-%p1)        ;; destination
  (imp-receive-%p2)        ;; source
  (imp-receive-%p3)        ;; temp physical space, length=32
  (imp-move-%i3-to-%a12)
  (imp-load-%i3 0)
  (cmis-fpu-reset-transposers)
  (cmis-fpu-always)

  ;; Begin VP Loop
  (imp-move-%v-to-%c2)
  (imp-save-flag-bases-ctx)
  (label modpi-vp-loop)
  (cmis-load-flags-always-ctx)

  ;; Put 32 X values into Transposer A
  (cmis-fpu-mwta-%p2 32)

  ;; Begin Inner Band Loop.  Loop 4 times, each time calculate P(X) for 8 values of X.
  (imp-load-%c1 4)
  (label modpi-band-loop)

  ;; move next values of X into Weitek rugs
  (cmis-fpu-static
    (imp:wtl3132-static-instruction :c-port :disable))
  (cmis-fpu-frta-dyn
    8
    (imp:wtl3132-dynamic-instruction-up
      :count 8
      :c-addr mod-src :c-addr-inc 1
      :io-direction :load))

  ;; load .5 into temp1
  (cmis-fpu-load-bypass-constant
    (first half-single))
  (cmis-fpu-static
    (imp:wtl3132-static-instruction :c-port :disable :wtl3132-function :float-add))
  (cmis-fpu-frb-dyn
    1
    (imp:wtl3132-dynamic-instruction
      :a-addr 0 :c-addr 0
      :io-direction :load :alu-destination :temp1-and-cbus
      :alu-b-input :zero :mult-b-input :cbus))

  ;; regs[dest..dest+count-1]
  ;;    <- regs[src..src+count-1]*1/2pi + .5
  ;; or:
  ;; regs[dest..dest+count-1]
  ;;    <- regs[src..src+count-1]*1/pi + .5
  (cmis-fpu-load-bypass-constant
    (first 1/2pi-single))

  (cmis-fpu-static
```

```
  (imp:wtl3132-static-instruction :c-port :enable
                        :wtl3132-function :float-mult-and-add))
 (cmis-fpu-frb-dyn
  8
  (imp:wtl3132-dynamic-instruction-up
   :count 8
   :a-addr mod-dest
   :c-addr mod-dest
   :b-addr mod-src
   :mult-b-input :bbus
   :alu-b-input :temp1
   :alu-destination :cbus
   :io-direction :load))

 ;; regs[dest..dest+count-1]
 ;;    <- floor(regs[dest..dest+count-1])
 (cmis-fpu-static
  (imp:wtl3132-static-instruction :c-port :enable
                        :wtl3132-function :miscellaneous))
 (cmis-fpu-dyn
  (+ (* 2 8 ) 1)
  (imp:wtl3132-dynamic-instruction-up
   :count 8
   :a-addr mod-dest
   :c-addr mod-dest
   :b-addr :float-to-fix
   :b-addr-inc 0
   :alu-b-input :zero :alu-destination :cbus)

  ;; Weitek chip has a bug which clobbers the destination register of
  ;; the last float-to-fix instruction therefore we clobber an extra one...
  (imp:wtl3132-dynamic-instruction
   :a-addr 0
   :c-addr 0
   :b-addr :float-to-fix
   :alu-b-input :zero :alu-destination :cbus)

  ;; now convert back to float
  (imp:wtl3132-dynamic-instruction-up
   :count 8
   :a-addr mod-dest
   :c-addr mod-dest
   :b-addr :fix-to-float
   :b-addr-inc 0
   :alu-b-input :zero
   :alu-destination :cbus))

 ;; regs[dest..dest+count-1]
 ;;    <- regs[dest..dest+count-1] * 2pi)
 ;; or:
 ;; regs[dest..dest+count-1]
 ;;    <- regs[dest..dest+count-1] * pi)
 (cmis-fpu-load-bypass-constant (first two-pi-single))
```

```
(cmis-fpu-static
  (imp:wtl3132-static-instruction :c-port :enable
                    :wtl3132-function :float-mult-and-add))
(cmis-fpu-frb-dyn
  8
  (imp:wtl3132-dynamic-instruction-up
    :count 8
    :a-addr mod-dest
    :c-addr mod-dest
    :b-addr mod-dest
    :mult-b-input :bbus
    :alu-b-input :zero
    :alu-destination :cbus
    :io-direction :load))

;; regs[dest..dest+count-1]
;; <- regs[src..src+count-1] - regs[dest..dest+count-1]
(cmis-fpu-static
  (imp:wtl3132-static-instruction :c-port :enable
                    :wtl3132-function :float-subtract))
(cmis-fpu-dyn
  8
  (imp:wtl3132-dynamic-instruction-up
    :count 8
    :a-addr mod-src
    :c-addr mod-dest
    :b-addr mod-dest
    :alu-b-input :bbus
    :alu-destination :cbus
    :io-direction :nop))

;; regs to transposer b
(cmis-fpu-static
  (imp:wtl3132-static-instruction :c-port :disable))
(cmis-fpu-fwtb-dyn
  8
  (imp:wtl3132-dynamic-instruction-up
    :count 8
    :a-addr mod-dest
    :c-addr mod-dest
    :io-direction :store))

;; end of band loop
(imp-dec-%c1)
(imp-jump-nz-%c1 (offset modpi-band-loop))

;; write LSH of results (now in transposer b) out to memory
(cmis-fpu-mrtb-%p3 32)

;; copy temp to dest conditionally
(imp-move-%p2-to-%a2)
(imp-move-%p3-to-%p2)
(cmis-lls-up
  32
```

```
  (imp:lls-args :sum-func ':b))

(imp-move-%a2-to-%p2)
(imp-add-const-to-%p1 -32)

;; Advance the flags and field pointers for the next VP
(imp-advance-flags-ctx)
(imp-advance-pointers)

(imp-dec-%c2)
(imp-jump-nz-%c2 (offset modpi-vp-loop))

;; All done
(imp-restore-flag-bases-ctx)
(imp-move-%a12-to-%i3)
(imp-exit))
```

5.4.2. WTL3164 Double Precision Version

This version of the Modulo Pi IMP is written for doing the calculation in double precision on the WTL3164 FPU.

```
(defimp DOUBLE-MOD-PI (dest source temp32)
  (imp-receive-%p1)     ;;dest (result) physical address
  (imp-receive-%p2)     ;; source (x) physical address
  (imp-receive-%p3)     ;; temp physical space len=32

  (cmis-fpu-reset-transposers)
  (cmis-fpu-always)

  (imp-move-%i3-to-%a9)
  (imp-move-%i4-to-%a10)
  (imp-load-%i3 1)
  (imp-load-%i4 1)

  ;; Begin VP Loop
  (imp-move-%v-to-%c2)
  (imp-save-flag-bases-ctx)

  (label vp-loop)
  (cmis-load-flags-always-ctx)

  ;; Put lsh of X values into Transposer B, and msh into Transposer C
  (cmis-fpu-mwtb-%p2 32)
  (imp-add-const-to-%p2 32)
  (cmis-fpu-mwtc-%p2 32)
  (imp-add-const-to-%p2 -32)

  ;; set up %P4 to point to where MSH of dest will be.  Set up %I4 to be
  ;; 1 because the the MSH of the result is saved in memory
  ;; transposed (directly from the SPRINT bypass reg), and therefore
  ;; need to increment by exactly one after each bypass-to-memory transfer.

  (imp-move-%p3-to-%p4)
  (imp-add-const-to-%p4 32)

  ;; Begin Inner Band Loop.  Loop 4 times, each time calculate P(X) for 8 values of X.
  (imp-load-%c1 4)
  (label modpi-band-loop)

  ;;regs[SOURCE] <- source data
  (cmis-fpu-static
    (imp:wtl3164-static-inst :func :monadic :aain 0 :abin 0 :main 0 :mbin 0))
  (cmis-fpu-frtbc-dyn2
    8
    (imp:wtl3164-dynamic-inst-up-2
      :count 8
      :baddr1 0 :baddr-inc1 0
      :efaddr1 mod-src :efaddr-inc1 1
      :xcnt1 :load-lsh-reg-via-f
      :baddr2 0 :baddr-inc2 0
      :efaddr2 mod-src :efaddr-inc2 1
      :xcnt2 :load-msh-reg-via-f))

  ;; regs[dest] <- regs[src] * 1/modulo
  (cmis-fpu-load-bypass-constant
```

```
  1841940611)                    ;lower 32 bits of double precision 1/(2 * pi)
 (cmis-fpu-frb-dynamic
  (imp:wtl3164-dynamic-inst :baddr 0 :xcnt :load-lsh-y))
 (cmis-fpu-load-bypass-constant
  1069834032)                    ;upper 32 bits of double precision 1/(2 * pi)
 (cmis-fpu-static
  (imp:wtl3164-static-inst :func :dfmul :main :aadd :mbin :y)) ;A * Y
 (cmis-fpu-frb-dyn
  8
  (imp:wtl3164-dynamic-inst-up :count 8
                               :aaddr mod-src
                               :cdaddr mod-dest
                               :xcnt :load-msh-y))

 ;; regs[dest] <- floor(regs[dest])
 (cmis-fpu-static
  (imp:wtl3164-static-inst :func :monadic :mbin 0 :abin 0 :aain :aadd :main 0)) ;dfixr(A)
 (cmis-fpu-dyn
  8
  (imp:wtl3164-dynamic-inst-up :count 8
                               :baddr 8 :baddr-inc 0
                               :aaddr mod-dest
                               :cdaddr mod-dest))

 ;; regs[dest] <- float(regs[dest])
 (cmis-fpu-static
  (imp:wtl3164-static-inst :func :monadic :mbin 0 :abin 1 :aain :aadd :main 0)) ;dfloat(A)
 (cmis-fpu-dyn
  8
  (imp:wtl3164-dynamic-inst-up :count 8
                               :baddr 9 :baddr-inc 0
                               :aaddr mod-dest
                               :cdaddr mod-dest))

 ;; regs[dest] <- regs[dest] * 2pi
 ;; or:
 ;; regs[dest] <- regs[dest] * pi)
 (cmis-fpu-load-bypass-constant
  1413754136)                    ;lower 32 bits of double precision 2*pi
 (cmis-fpu-frb-dynamic
  (imp:wtl3164-dynamic-inst :baddr 0 :xcnt :load-lsh-y))
 (cmis-fpu-load-bypass-constant
  1075388923)                    ;upper 32 bits of double precision 2*pi
 (cmis-fpu-static
  (imp:wtl3164-static-inst :func :dfmul :main :aadd :mbin :y)) ;A * Y
 (cmis-fpu-frb-dyn
  8
  (imp:wtl3164-dynamic-inst-up :count 8
                               :aaddr mod-dest
                               :cdaddr mod-dest
                               :xcnt :load-msh-y))

 ;; regs[dest] <- regs[src] - regs[dest]
 (cmis-fpu-static
```

```
  (imp:wtl3164-static-inst :func :FADD-sub :aain :aadd :abin :badd :main 1 :mbin 0))    ;A - B
(cmis-fpu-frb-dyn
  8
  (imp:wtl3164-dynamic-inst-up :count 8
                               :aaddr mod-src
                               :baddr mod-dest
                               :cdaddr mod-dest))

(cmis-fpu-static
  (imp:wtl3164-static-inst :func :monadic :aain 0 :abin 0 :main 0 :mbin 0))

;;store the LSH of the results into Transposer A

(cmis-fpu-fwta-dyn
  8
  (imp:wtl3164-dynamic-inst-up :count 8
                               :baddr 0 :baddr-inc 0
                               :xcnt :store-lsh-reg-via-e-port
                               :efaddr mod-dest))

;;store the MSH of the results into memory transposed

(cmis-fpu-mrb-%p4-stride-fwb-dyn
  8
  (imp:wtl3164-dynamic-inst-up :count 8
                               :baddr 0 :baddr-inc 0
                               :xcnt :store-msh-reg-via-e-port
                               :efaddr mod-dest))

;; end of band loop
(imp-dec-%c1)
(imp-jump-nz-%c1 (offset modpi-band-loop))

;; Copy LSHs of results from Tranposer A to destination
(cmis-fpu-mrta-%p3 32)

;; do a transpose on the MSHs of the result (stored in memory 90 degrees transposed)
(imp-add-const-to-%P4 -32)
(cmis-fpu-mwta-%P4 32)
(cmis-fpu-mrta-%P4 32)

;; copy temp to dest conditionally
(imp-move-%p2-to-%a2)
(imp-move-%p3-to-%p2)
(cmis-lls-up
  64
  (imp:lls-args :sum-func ':b))
(imp-move-%a2-to-%p2)
(imp-add-const-to-%p1 -64)

;; Advance the flags and field pointers for the next VP
(imp-advance-flags-ctx)
(imp-add-%I1-to-%P1)
(imp-add-%I2-to-%P2)
```

```
(imp-dec-%c2-jump-nz (offset vp-loop))

;; All done
(imp-restore-flag-bases-ctx)
(imp-move-%A9-to-%I3)
(imp-move-%A10-to-%I4)
(imp-exit))
```

5.5. WTL3132 Division Example

The following IMP fragment performs a divide on the WTL3132 FPU. In particular, it divides the contents of R1 by R2 and places the result in R3. R4 is used as a temporary. The code is written for IMPASS for use from C.

The algorithm computes R3 = R1 / R2 (using R4 as a temporary) as follws:

1. R4 = lookup(R2)	Get the inverse seed from the table
2. R3 = 2 - (R4 * R2)	Iteration 1...
3. R4 = R3 * R4	
4. R3 = 2 - (R4 * R2)	Iteration 2...
5. R4 = R3 * R4	R4 = inverse(R2)
6. R3 = R4 * R1	result = numerator * inverse(denominator)

The IMP code follows:

```
# R4 = lookup(R2)
CMIS_FPU_STATIC
  wtl3132_static_instruction c_port=ENABLE, wtl3132_function=MISCELLANEOUS
CMIS_FPU_DYN
  1
  wtl3132_dynamic_instruction a_addr=2, c_addr=4, b_addr=FLUT
CMIS_FPU_STATIC
  wtl3132_static_instruction        c_port=ENABLE, wtl3132_function=FLOAT_MULT_NEGATE_AND_ADD
# R3 = 2-(R4*R2)
CMIS_FPU_DYN
  1
  wtl3132_dynamic_instruction       a_addr=4, b_addr=2, c_addr=3,       mult_b_input=BBUS, alu_b_input=TWO, a
# R4 = R3*R4
CMIS_FPU_STATIC
  wtl3132_static_instruction        c_port=ENABLE, wtl3132_function=FLOAT_MULT_AND_ADD
CMIS_FPU_DYN
  1
  wtl3132_dynamic_instruction       a_addr=3, b_addr=4, c_addr=4,       mult_b_input=BBUS, alu_b_input=ZERO,
# R3 = 2-(R4*R2)
CMIS_FPU_STATIC
  wtl3132_static_instruction        c_port=ENABLE, wtl3132_function=FLOAT_MULT_NEGATE_AND_ADD
CMIS_FPU_DYN
  1
  wtl3132_dynamic_instruction       a_addr=4, b_addr=2, c_addr=3,       mult_b_input=BBUS, alu_b_input=TWO, a
# R4 = R3 * R4
# R3 = R4 * R1
CMIS_FPU_STATIC
  wtl3132_static_instruction        c_port=ENABLE, wtl3132_function=FLOAT_MULT_AND_ADD
CMIS_FPU_DYN
  5
  wtl3132_dynamic_instruction       a_addr=3, b_addr=4, c_addr=4,       mult_b_input=BBUS, alu_b_input=ZERO,
  wtl3132_dynamic_instruction c_addr=31   # nop
  wtl3132_dynamic_instruction c_addr=31   # nop
  wtl3132_dynamic_instruction c_addr=31   # nop
  wtl3132_dynamic_instruction       a_addr=4, b_addr=1, c_addr=3,       mult_b_input=BBUS, alu_b_input=ZERO,
```