Chapter 17

Local Architecture

17.1 Control Processor Architecture

A control processor (CP) is essentially like a standard high-performance workstation computer. It consists of a standard RISC microprocessor, associated memory and memory interface, and perhaps I/O devices such as local disks and Ethernet connections. It also includes a CM-5 Network Interface, providing access to the Control Network and Data Network.

A control processor acting as a partition manager (PM) controls each partition and communicates with the rest of the CM-5 system through the Control Network and Data Network. For example, a PM initiates I/O by sending a request through the Data Network to a second CP, an I/O control processor. A PM initiates task-switching by using the Control Network to send a broadcast interrupt to all processing nodes; privileged operating-system support code in each node then carries out the bulk of the work. To access the Control Network and Data Network, each CP uses its Network Interface, a memory-mapped device in the memory address space of its microprocessor.

The microprocessor supports the customary distinction between user and supervisor code. User code can run in the control processor at the same time that user code for the same job is running in the processing nodes. Protection of the supervisor, and of one user from another, is supported by the same mechanisms used in workstations and single-processor timeshared computers, namely memory address mapping and protection and the suppression of privileged operations in user mode. In particular, the operating system prevents a user process from performing privileged Network Interface operations; the privileged control registers simply are not mapped into the user address space.

Copyright © 1993 Thinking Machines Corporation

Current implementations of the CM-5 control processor use either SPARC or SuperSparc microprocessors. It is expected that, over time, the implementation of the CP will track the RISC microprocessor technology curve to provide the best possible functionality and performance at any given point in time; therefore it is recommended that low-level programming be carried out in C as much as possible, rather than in assembly language.

17.2 Processing Node Architecture

The CM-5 processing node is designed to deliver very good cost-performance when used in large numbers for data parallel applications. Like the control processor, the node makes use of industry-standard RISC microprocessor technology. This microprocessor may optionally be augmented with a special high-performance hardware arithmetic accelerator that uses wide data paths, deep pipelines, and large register files to improve peak computational performance.

The node design is centered around a standard 64-bit bus. To this node bus are attached a RISC microprocessor, a CM-5 Network Interface, and memory. Note that all logical connections to the rest of the system pass through the Network Interface.

The node memory consists of standard DRAM chips and an 8-Kbyte boot ROM; the microprocessor also has a 64-Kbyte cache that holds both instructions and data. All DRAM memory is protected by ECC checking, which corrects single-bit failure and detects two-bit errors and DRAM chip failures. The boot ROM contains code to be executed following a system reset, including local processor and memory verification and the communications code needed to download further diagnostics or operating system code.

The memory configuration depends on whether the optional high-performance arithmetic hardware is included. Without the arithmetic hardware, the memory is connected by a 72-bit path (64 data bits plus 8 ECC bits) to a memory controller that in turn is attached to the node bus. (See Figure 29.) In this configuration the memory size can be 8, 16, or 32 Mbytes. (This assumes 4-Mbit DRAM technology. Future improvements in DRAM technology will permit increases in memory size. The CM-5 architecture and chip implementations anticipate these future improvements.)

November 1993
Copyright © 1993 Thinking Machines Corporation

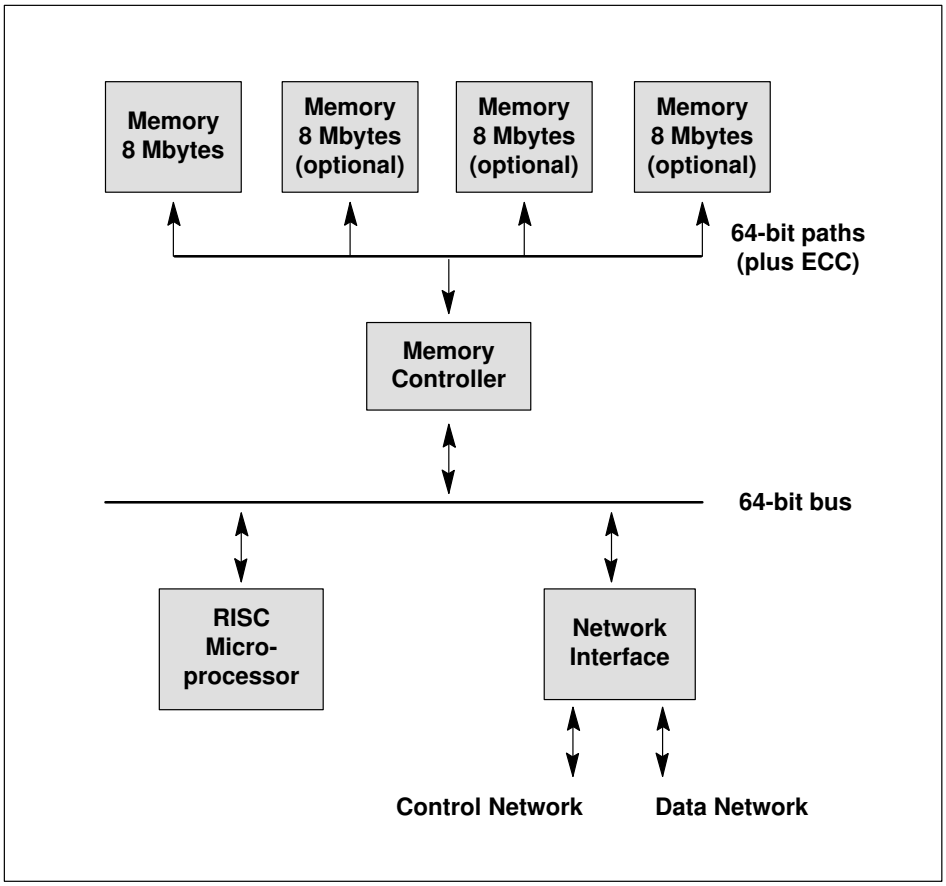

Figure 29. Processing node.

The basic CM-5 processing node consists of a RISC microprocessor, memory subsystem, and a CM-5 Network Interface all connected to a standard 64-bit bus. The RISC microprocessor is responsible for instruction fetch, instruction execution, processing data, and controlling the Network Interface. The memory subsystem consists of a memory controller and either 8 Mbytes, 16 Mbytes, or 32 Mbytes of DRAM memory. The path from each memory back to the memory controller is 72 bits wide, consisting of 64 data bits and 8 bits of ECC code. The ECC circuits in the memory controller can correct single-bit errors and detect double-bit errors as well as failure of any single DRAM chip. The Network Interface connects the node to the rest of the system through the Control Network and Data Network.

Copyright © 1993 Thinking Machines Corporation

If the high-performance arithmetic hardware is included, then the node memory is divided into four independent banks, each with a 72-bit (64 data bits plus 8 ECC bits) access path. (See Figure 30.)

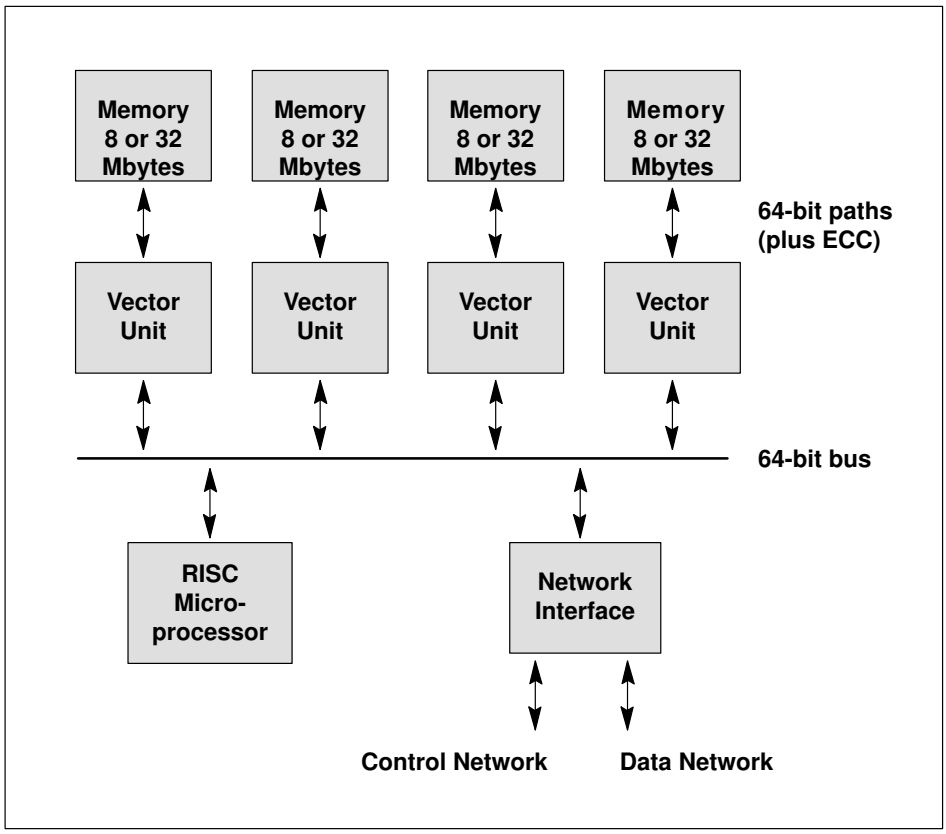

Figure 30. Processing node with vector units.

A CM-5 processing node may optionally contain an arithmetic accelerator. In this configuration the node has 32 or 128 Mbytes of memory, four banks of 8 or 32 Mbytes each. The memory controller is replaced by four vector units. Each vector unit has a dedicated 72-bit path to its associated memory bank, providing peak memory bandwidth of 160 Mbytes/sec per vector unit, and performs all the functions of a memory controller, including generation and checking of ECC bits. Each vector unit has 40 Mflops peak 64-bit floating-point performance and 40 Mops peak 64-bit integer performance. The vector units execute vector instructions issued to them by the RISC microprocessor. Each vector instruction may be issued to a specific vector unit (or pair of units), or broadcast to all four vector units at once. The microprocessor takes care of such "housekeeping" computations as address calculation and loop control, overlapping them with vector instruction execution. Together, the vector units provide 640 Mbytes/sec memory bandwidth and 160 Mflops peak 64-bit floating-point performance. A single CM-5 node with vector units is a supercomputer in itself.

Copyright © 1993 Thinking Machines Corporation

The special arithmetic hardware consists of four vector units (VUs), one for each memory bank, connected separately to the node bus. In this configuration the memory size is either 8 or 32 Mbytes per VU for a total of 32 or 128 Mbytes per node. (This figure assumes either 4-Mbit or 16-Mbit DRAM technology and will increase as industry-standard memories are improved.) Each VU also implements all memory controller functions, including ECC checking, so that the entire memory appears to be in the address space of the microprocessor exactly as if the arithmetic hardware were not present.

The memory controller or vector unit also provides a word-based interface to the system Diagnostics Network (see Section 19.8). This provides an extra communications path to the node; it is designed to be slow but reliable and is used primarily for hardware fault diagnosis.

As with the control processors, the implementation of the CM-5 processing node is expected to track the RISC microprocessor technology curve to provide the best possible functionality and performance at any given point in time; therefore it is recommended that low-level programming be carried out in C as much as possible, rather than in assembly language. Current implementations of the CM-5 processing node use SPARC or SuperSparc microprocessors.

17.3 Vector Unit Architecture

Each vector unit (VU) is a memory controller and computational engine controlled by a memory-mapped control-register interface. (See Figure 31.) When a read or write operation on the node bus addresses a VU, the memory address is further decoded. High-order bits indicate the operation type:

- For an ordinary *memory transaction*, the low-order address bits indicate a location in the memory bank associated with the VU, which acts as a memory controller and performs the requested memory read or write operation.
- For a *control register access*, the low-order address bits indicate a control register to be read or written.
- For a *data register access*, the low-order address bits indicate a data register to be read or written.

Copyright © 1993 Thinking Machines Corporation

- For a *vector-unit instruction*, the node memory bus operation must be **`write`** (an attempt to **`read`** from this part of the address space results in a bus error). The data on the memory bus is not written to memory but is interpreted as an instruction to be executed by the vector execution portion of the VU. The low-order address bits indicate a location in the memory bank associated with the VU; the instruction uses this address if it includes operations on memory. A vector-unit instruction may be addressed to any single VU (in which case the other three VUs ignore it), to a pair of VUs, or to all four VUs simultaneously.

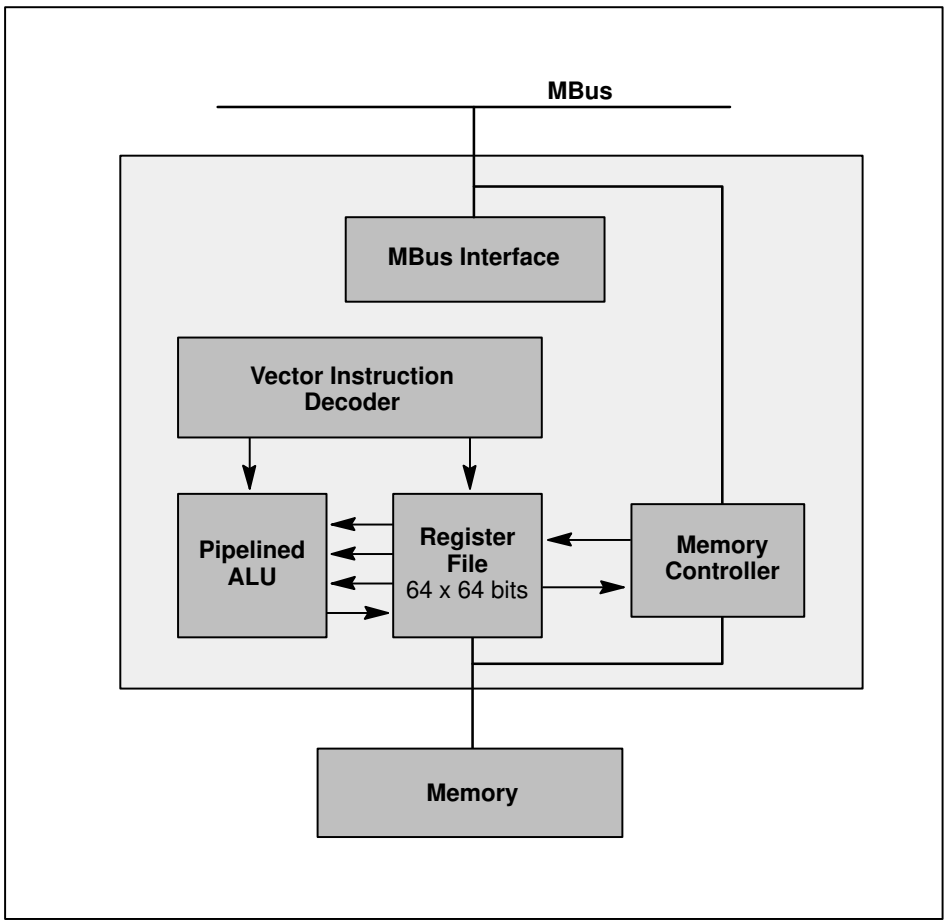

Figure 31. Vector unit functional architecture.

The first two types of operation are identical to those performed by the memory controller when vector units are absent. The third type permits the microprocessor to read or write the register file of any vector unit. The fourth type of operation initiates high-performance arithmetic computation. This computa-

November 1993
Copyright © 1993 Thinking Machines Corporation

tion has both vector and parallel characteristics: each VU can perform vector operations, and a single instruction may be issued simultaneously to all four. If the vector length is 16, then issuing a single instruction can result in as many as 64 individual arithmetic instructions (16 for each of the four VUs), or even 128 operations if the instruction specifies a compound operation such as **multiply-add**.

Vector units cannot fetch their own instructions; they merely react to instructions issued to them by the microprocessor. The instruction format, instruction set, and maximum vector length have been chosen so that the microprocessor can keep the vector units busy while having time of its own to fetch instructions (both its own and those for the vector units), calculate addresses, execute loop and branch instructions, and carry out other algorithmic bookkeeping.

Each vector unit has 64 64-bit registers, which can also be addressed as 128 32-bit registers. Other control registers worth noting are the 16-bit vector mask (VM) and the 4-bit vector length (VL) registers. The vector mask register controls certain conditional operations and optionally receives single-bit status results for each vector element processed. The vector length register specifies the number of elements to be processed by each vector instruction.

The vector unit actually processes both vector and scalar instructions; a scalar-mode instruction is handled as if it were a vector-mode instruction of length 1. Thus scalar-mode instructions always operate on single registers; vector-mode instructions operate on sequences of registers. Each register operand is specified by a 7-bit starting register number and a 7-bit stride. The first element for that vector operand is taken from the starting register; thereafter the register number is incremented by the stride to produce a new register number indicating the next element to be processed. Using a large stride has the same effect as using a negative stride, so it is possible to process a vector in reverse order. Most instruction formats use a default stride of 1 for 32-bit operands or 2 for 64-bit operands, so as to process successive registers, but one instruction format allows arbitrary strides to be specified for all operands, and another allows one vector operand to take its elements from an arbitrary pattern of registers by means of a mechanism for indirect addressing of the register file.

Each vector unit includes an adder, a multiplier, memory load/store, indirect register addressing, indirect memory addressing, and population count. Every vector-unit instruction can specify at least one arithmetic operation and an independent memory operation. Every instruction also has four register-address fields: three for the arithmetic operation and one for the memory operation. All binary arithmetic operations are fully three-address; an addition, for example, can read two source registers and write into a third destination register. The

Copyright © 1993 Thinking Machines Corporation

memory operation can address a completely independent register. If, however, a load operation addresses a register that is also a source for the arithmetic operation, then load-chaining occurs, so that the loaded memory data is used as an arithmetic operand in the same instruction. Indirect memory addressing supports scatter/gather operations and vectorized pointer indirection.

Two mechanisms provide for conditional processing of vector elements within each processing node. Each vector unit contains a vector mask register; vector elements are not processed in positions where the corresponding vector mask bit is zero. Alternatively, a vector-mask enumeration mechanism may be used in conjunction with the scatter/gather facility to pack vector elements that require similar processing; after bulk application of unconditional vector operations, the results are then unpacked and scattered to their originally intended destinations.

Vector-unit instructions come in five formats. (See Figure 32.) The 32-bit short format allows many common scalar and vector operations to be expressed succinctly. The four 64-bit long formats extend the basic 32-bit format to allow additional information to be specified: a 32-bit immediate operand, a signed memory stride, a set of register strides, or additional control fields (some of which can update certain control registers with no additional overhead).

The short format includes an arithmetic opcode (8 bits), a load/store opcode (3 bits), a vector/scalar mode specifier (2 bits), and four register fields called rLS, rD, rS1, and rS2 that designate the starting registers for the load/store operation and for the arithmetic destination, first source, and second source, respectively. The vector/scalar specifier indicates whether the instruction is to be executed once (scalar mode) or many times (vector mode). It also governs the expansion of the 4-bit register specifiers into full 7-bit register addresses. The short format is designed to support a conventional division of the uniform register file into vector registers of length 16, 8, or (for 64-bit operands only) 4, with scalar quantities kept in the first 16 registers. For a scalar-mode instruction, the 4-bit register field provides the low-order bits of the register number (which is then multiplied by 2 for 64-bit operands); for a vector-mode instruction, it provides the high-order bits of the register number. The rS1 field is 7 bits wide; in some cases these specify a full 7-bit register number for arithmetic source 1 and in other cases 4 bits specify a vector register and the other 3 bits convey stride information.

November 1993
Copyright © 1993 Thinking Machines Corporation

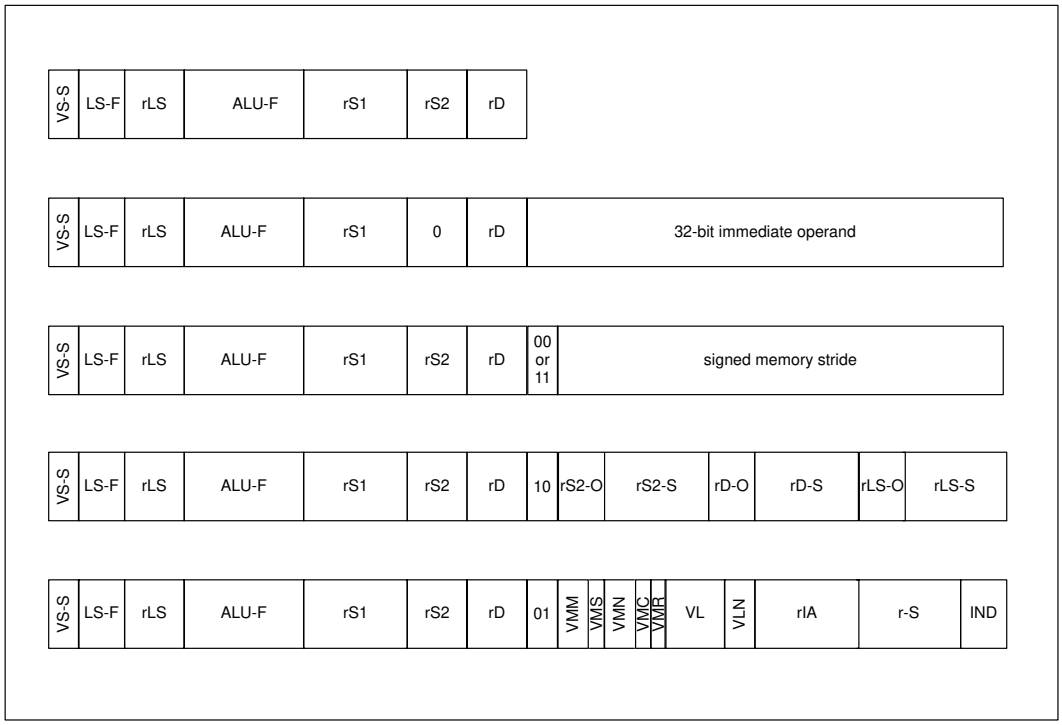

Figure 32. Vector unit instruction formats.

Each instruction issued by the RISC microprocessor to the vector units is 32 bits or 64 bits wide. The 32-bit format is designed to cover the operations and register access patterns most likely to arise in high-performance compiled code. The 32 high-order bits of the 64-bit format are identical to the 32-bit format. The 32 low-order bits provide an immediate operand, a signed memory stride, or specifications for more complex or less frequent operations.

A short scalar-mode instruction can therefore access the first 16 32-bit or 64-bit elements of the register file, simultaneously performing an arithmetic operation and loading or storing a register. (The memory address that accompanies the issued instruction indicates the memory location to be accessed.) One of the arithmetic operands (S1) may be in any of the 128 registers in the register file.

A short vector-mode instruction can conveniently treat the register file as a set of vector registers:

16 4- x 64-bit vector registers
8 8- x 64-bit vector registers
4 16- x 64-bit vector registers
16 8- x 32-bit vector registers
8 16- x 32-bit vector registers

Copyright © 1993 Thinking Machines Corporation

Many options are available for vector-mode instructions. These include a choice between a default memory stride and the last explicitly specified memory stride, as well as a choice of register stride for the S1 operand (last specified, 1, or 0 — stride 0 treats the S1 operand as a scalar to be combined with every element of a vector).

The long instruction formats are all compatible extensions of the short format: the most significant 32 bits of a 64-bit instruction are decoded as a 32-bit instruction, and the least significant 32 bits specify additional operations or operands. If the rS2 field of a long instruction is zero, then the low-order 32 bits of the instruction constitute an immediate scalar value to be used as the S2 operand. If the arithmetic operation requires a 64-bit operand, then the immediate value is zero-extended left if an unsigned integer is required, sign-extended left for a signed integer, or zero-extended right for a floating-point number.

If the rS2 field of a long instruction is not zero, then the two high-order bits of the low 32 are decoded. If the two bits match, then the low-order 32 bits are an explicit signed memory stride. (Note that it is possible to specify such a stride even in a scalar-mode long instruction, in order to latch the stride in preparation for a following vector-mode instruction that might need to use another of the long formats.) Code 10 indicates additional register number and register stride information, allowing specification of complete 7-bit register numbers and register strides for the rLS, rD, and rS2 operands. This enables complex regular patterns of register access. Code 01 indicates a variety of control fields for such mechanisms as changing the vector length, controlling use of the vector mask, indirect addressing, S1 operand register striding, and population count.

The arithmetic operations that can be specified by the ALU-F instruction field are summarized in Table 1. Note the large set of three-operand multiply-add instructions. These come in three different addressing patterns: accumulative, which adds a product into a destination register (useful for dot products); inverted, which multiplies the destination by one source and then adds in the other (useful for polynomial evaluation and integer subscript computations); and full triadic, which takes one operand from the load/store register so that the destination register may be distinct from all three sources. The triadic multiply-add operations are provided for signed and unsigned integers as well as for floating-point operands, in both 32-bit and 64-bit sizes. Unsigned 64-bit multiply-boolean operations are also provided. (Note that multiplying by a power of two has the effect of a shift.)

Copyright © 1993 Thinking Machines Corporation

Table 1. Summary of vector unit arithmetic instructions (Part I).

imove	dimove	umove	dumove	fmove	dfmove	Move: D = S1 + 0
itest	ditest	utest	dutest	ftest	dftest	Move and generate status
icmp	dicmp	ucmp	ducmp	fcmp	dfcmp	Compare
iadd	diadd	uadd	duadd	fadd	dfadd	Add
isub	disub	usub	dusub	fsub	dfsub	Subtract
isubr	disubr	usubr	dusubr	fsubr	dfsubr	Subtract reversed
imul	dimul	umul	dumul	fmul	dfmul	Multiply (low 64 bits for integers)
	dimulh		dumulh			Integer multiply (high 64 bits)
				fdiv	dfdiv	Divide
				finv	dfinv	Invert: D = 1.0/S1
				fsqrt	dfsqrt	Square root
				fisqt	dfisqt	Inverse square root: D = 1.0/SQRT(S2)
ineg	dineg			fneg	dfneg	Negate
iabs	diabs			fabs	dfabs	Absolute value
iaddc	diaddc	uaddc	duaddc			Integer add with carry
isubc	disubc	usubc	dusubc			Integer subtract with borrow
isbrc	disbrc	usbrc	dusbrc			Integer subtract reversed with borrow
		ushl	dushl			Integer shift left
		ushlr	dushlr			Integer shift left reversed
		ushr	dushr			Integer shift right logical
		ushrr	dushrr			Integer shift right logical reversed
ishr	dishr					Integer shift right arithmetic
ishrr	dishrr					Integer shift right arithmetic reversed
		uand	duand			Bitwise logical AND
		uandc	duandc			Bitwise logical AND with complement
		unand	dunand			Bitwise logical NAND
		uor	duor			Bitwise logical OR
		unor	dunor			Bitwise logical NOR
		uxor	duxor			Bitwise logical XOR
		unot	dunot			Bitwise logical NOT
		umrg	dumrg			Merge: D = (if mask then S2 else S1)
		uffb	duffb			Find first 1-bit

Copyright © 1993 Thinking Machines Corporation

Table 1. Summary of vector unit arithmetic instructions (Part II).

imada	dimada	umada	dumada	fmada	dfmada	rD = (rS1 * rS2) + rD
imsba	dimsba	umsba	dumsba	fmsba	dfmsba	rD = (rS1 * rS2) – rD
imsra	dimsra	umsra	dumsra	fmsra	dfmsra	rD = – (rS1 * rS2) + rD
inmaa	dinmaa	unmaa	dunmaa	fnmaa	dfnmaa	rD = – (rS1 * rS2) – rD
imadi	dimadi	umadi	dumadi	fmadi	dfmadi	rD = (rS2 * rD) + rS1
imsbi	dimsbi	umsbi	dumsbi	fmsbi	dfmsbi	rD = (rS2 * rD) – rS1
imsri	dimsri	umsri	dumsri	fmsri	dfmsri	rD = – (rS2 * rD) + rS1
inmai	dinmai	unmai	dunmai	fnmai	dfnmai	rD = – (rS2 * rD) – rS1
imadt	dimadt	umadt	dumadt	fmadt	dfmadt	rD = (rS1 * rLS) + rS2
imsbt	dimsbt	umsbt	dumsbt	fmsbt	dfmsbt	rD = (rS1 * rLS) – rS2
imsrt	dimsrt	umsrt	dumsrt	fmsrt	dfmsrt	rD = – (rS1 * rLS) + rS2
inmat	dinmat	unmat	dunmat	fnmat	dfnmat	rD = – (rS1 * rLS) – rS2
			dumsa			rD = lower(rS1 * rS2) AND rD
			dumhsa			rD = upper(rS1 * rS2) AND rD
			dumma			rD = lower(rS1 * rS2) AND NOT rD
			dumhma			rD = upper(rS1 * rS2) AND NOT rD
			dumoa			rD = lower(rS1 * rS2) OR rD
			dumhoa			rD = upper(rS1 * rS2) OR rD
			dumxa			rD = lower(rS1 * rS2) XOR rD
			dumhxa			rD = upper(rS1 * rS2) XOR rD
			dumsi			rD = lower(rS2 * rD) AND rS1
			dumhsi			rD = upper(rS2 * rD) AND rS1
			dummi			rD = lower(rS2 * rD) AND NOT rS1
			dumhmi			rD = upper(rS2 * rD) AND NOT rS1
			dumoi			rD = lower(rS2 * rD) OR rS1
			dumhoi			rD = upper(rS2 * rD) OR rS1
			dumxi			rD = lower(rS2 * rD) XOR rS1
			dumhxi			rD = upper(rS2 * rD) XOR rS1
			dumst			rD = upper(rS1 * rLS) AND rS2
			dumhst			rD = upper(rS1 * rLS) AND rS2
			dummt			rD = lower(rS1 * rLS) AND NOT rS2
			dumhmt			rD = upper(rS1 * rLS) AND NOT rS2
			dumot			rD = lower(rS1 * rLS) OR rS2
			dumhot			rD = upper(rS1 * rLS) OR rS2
			dumxt			rD = lower(rS1 * rLS) XOR rS2
			dumhxt			rD = upper(rS1 * rLS) XOR rS2

Copyright © 1993 Thinking Machines Corporation

Table 1. Summary of vector unit arithmetic instructions (Part III).

			`fclas`	`dfclas`	Classify operand
			`fexp`	`dfexp`	Extract exponent
			`fmant`	`dfmant`	Extract mantissa with hidden bit
	`uenc`	`duenc`			Make float from exponent (S1) and mantissa (S2)
			`fnop`		No arithmetic operation
`cvtfi`					Convert integer to float*
`cvtf`					Convert float to float*
`cvtir`					Convert float to integer (round)*
`cvti`					Convert float to integer (truncate)*
`trap`					Generate debug trap
`etrap`					Generate trap on enabled exception
`ldvm`					Load vector mask
`stvm`					Store vector mask

* The rS2 field encodes the source and result sizes and formats for these instructions.

The LS-F instruction field specifies one of 5 load/store operations:

- no operation
- 32-bit load
- 64-bit load
- 32-bit store
- 64-bit store

The load/store size (32 or 64 bits) need not be the same as the arithmetic operand size. They should be the same, however, if load chaining is used. There is no distinction between integer and floating-point loads and stores. A 64-bit load or store may be used to load or store an even-odd 32-bit register pair.

Executing Vector Code

All instruction fetching and control decisions for the vector units are made by the node microprocessor. When vector units are present, all instructions and data reside in the memory banks associated with the vector units. A portion of each memory bank is conventionally reserved for instruction and data areas for the

Copyright © 1993 Thinking Machines Corporation

microprocessor. The memory management hardware of the microprocessor is used to map pages from the four memory banks so as to make them appear contiguous to the microprocessor.

While the microprocessor does not have its own memory, it does have a local cache that is used for both instruction and data references. Thus, the microprocessor and vector units can execute concurrently so long as no cache misses occur.

When a cache block must be fetched from memory, the associated vector unit may be in one of three states. If it is not performing any local operations, then the cache block is fetched immediately. If it is performing a local load or store operation, then the block fetch is delayed until the operation completes. If the vector unit is doing an operation that does not require the memory bus, then the block fetch proceeds immediately, concurrently with the executing vector operation.

The microprocessor issues VU instructions by storing to a specially constructed address: the microprocessor fetches the instruction itself from its data memory, calculates the special vector-unit destination address for issuing the instruction, and executes the store. The time it takes the microprocessor to do this is generally less than the time it takes a vector unit to execute an instruction with a vector length of 4. Moreover, the tail end of one vector instruction may be overlapped in time with the beginning of the next, thus eliminating memory latency and vector start-up overhead. With careful programming, therefore, the microprocessor can sustain delivery of vector instructions so as to keep the vector units continuously busy.

The vector unit is optimally suited for a vector length of 8; with vectors this long, the timing requirements are not so critical, and the microprocessor has time to spare for bookkeeping operations. The short vector-unit instruction format supports addressing of length-8 register blocks for either 32-bit or 64-bit operands. This provides 8 vector registers for 64-bit elements or 16 vector registers for 32-bit elements, with the first two such register blocks also addressable as 16 scalar registers. This is only a conventional arrangement, however; long-format instructions can address the registers in arbitrary patterns.

Flow control of instructions to the vector units is managed using the hardware protocol of the node bus. When a vector instruction is issued by the microprocessor, any addressed vector unit may stall the bus if it is busy. A small write buffer and independent bus controller within the microprocessor allows it to continue local execution of its own instructions while the bus is stalled by a vector unit.

Copyright © 1993 Thinking Machines Corporation

If the microprocessor gets far enough ahead, the small write buffer becomes full, causing the microprocessor to stall until the vector unit(s) catch up.

Each vector instruction either completes successfully or terminates in a hard error condition. Exceptions and other non-fatal conditions are signaled in sticky status registers that may be either polled or enabled to signal interrupts. Hard errors and enabled exception conditions are signaled to the microprocessor as interrupts via the Network Interface.

The memory addresses on the node bus are physical addresses resulting from memory-map translation in the microprocessor. The memory map provides the necessary protection to ensure that the addressed location itself is in fact within a user's permitted address space, but cannot prevent accesses to other locations by execution of vector instructions that use indirect addressing or memory strides. Additional protection is provided in each vector unit by bounds-checking hardware that signals an interrupt if specified physical address bounds are exceeded.

Certain privileged vector unit operations are reserved for supervisor use. These include the interrupt management and memory management features. The supervisor can interrupt a user task at any time for task-switching purposes and can save the state of each vector unit for transparent restoration at a later time.

Copyright © 1993 Thinking Machines Corporation