Architecture and Operation of Vector Computer CONVEX 3820

Kandó Kálmán Polytechnic
Institute of Computer Technology, Székesfehérvár
and
University of Athens

Etelka Kovács, Senior Lecturer
Ágnes Burián, Senior Lecturer

BACKGROUND

BASICS VECTOR ARCHITECTURE

A vector processor typically consists of ordinary pipelined scalar unit plus a vector unit. All functional units within the vector unit have a latency of several clock cycles. This allows a shorter clock cycle time and is compatible with long-running vector operations that can be deeply pipelined without generating hazards. Most vector processors allow the vectors to be dealt with as floating-point numbers, as integers, or as logical data. Here we will focus on floating point. The scalar unit is basically no different from the type of advanced pipelined CPU.

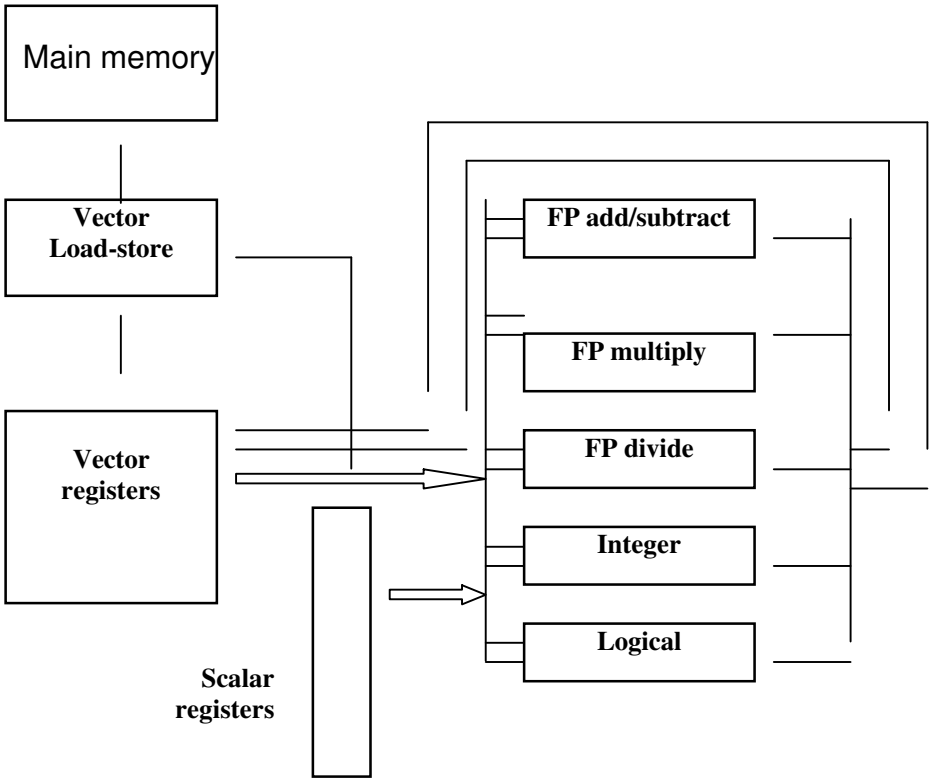
There are two primary types of architectures for vector processors:

- **vector-register** processor,
- **memory-memory vector** processor.

In a **vector-register** processor, all vector operations –except load and store – are among the vector registers. These architectures are vector counterpart of a load-store architecture. All major vector computers shipped since the late 1980s use a vector-register architecture; these include the Cray Research processors (CRAY-1, CRAY-2, X-MP, Y-MP, and C-90), the Japanese supercomputers (NEC SX/2 and NEC SX/3, Fujitsu VP200 and VP400, and the Hitachi S820), as well as **the mini-supercomputers** (**Convex C-1 and C-2**).

In a **memory-memory vector** processor, all vector operations are memory to memory. The first vector computers were of this type, as were CDC's vector computers.
From this point on we will focus on vector-register architectures only; we will briefly return to memory-memory vector architecture later to discuss why they have not been as successful as vector-register architectures.

The basic structure of a vector-register architecture:



This processor has a scalar architecture, vector registers, and all the functional units are vector functional units. Special vector instructions are defined both for arithmetic and for memory accesses. We show vector units for logical and integer operations. These are included a standard vector processor, which usually includes these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. These ports are connected to the inputs and outputs of vector functional units by a set of crossbars.

The primary components of instruction set architecture of vector processors are:

A./ Vector registers:
Each vector registers is a fixed-length bank holding a single vector. The vector processor has eight vector registers /and each vector register holds 64 elements/. Each vector register must have at least two read ports and one write ports. This will allow a high degree of overlap among vector operations to different vector registers. /The read and write ports, which total at least 16 read ports and eight write ports, are connected to the functional unit input or outputs by a pair of crossbars./

B./ Vector functional units:
Each unit fully pipelined and can start a new operation on every clock cycle. A control unit is needed to detect hazards, both from conflicts for the functional units (structural hazards) and from conflicts for register accesses (data hazards). This vector processor has five functional units, as shown above. For simplicity, we will focus exclusively on floating-point functional units. Depending on the vector processor, scalar operations either use the vector functional units or use a dedicated

set. We assume the functional units are shared, but again, for simplicity, we ignore potential conflicts.

C./ Vector load-store unit:
This is a vector memory unit that loads or stores a vector to or from memory. It is fully pipelined, so that words can be moved between the vector registers and memory with a bandwith of one word per clock cycle, after initial latency. This unit would also normally handle scalar loads and stores.

D./ A set of scalar registers:
Scalar registers can also provide data as input to the vector functional units, as well as compute addresses to pass to the vector load-store unit. These are the normal 32 general-purpose registers and 32 floating-point registers, though more read and write ports are needed. The scalar registers are also connected to the functional units by the pair of crossbars.

The following shows the characteristics some typical vector processors, including the size and count of the registers, the number and types of functional units, and the number of load-store units:

Processor	Year announced	Clock rate (MHz)	Registers	Elements per register	Functional units	Load-store units
CRAY-1	1976	80	8	64	**6: add, multiply, reciprocal, integer add, logical, shift**	1
CRAY-X-MP **CRAY Y-MP**	1983 1988	120 166	8	64	**8: FP add, FP multiply, FP reciprocal,** **integer add, 2 logical, shift, population count/parity**	2 loads 1 store
CRAY-2	1985	166	8	64	**5: FP add, FP multiply, FP reciprocal/sqrt,** **integer (add, shift, population count), logical**	1
Fujitsu VP100/200	1982	133	8-256	32-1024	**3: FP or integer add/ logical multiply, divide**	2
Hitachi S810/820	1983	71	32	256	**3: 2 integer add/logical, 1 multiply-add, and 1 multiply/ divide-add unit**	4
CONVEX C-1	***1985***	***10***	***8***	***128***	***4: multiply, add, divide, integer/logical***	***1***
NEC SX/2	1984	160	8+8192	256 variable	**16: 4 integer add/logical, 4 FP multiply/ divide, 4 FP add, 4 shift**	8
DLXV	1990	200	8	64	**5: multiply, add, divide, integer add, logical**	1
Cray C-90	1991	240	8	128	**8: FP add, FP multiply, FP reciprocal, integer add, 2 logical shift, population count/parity**	4
CONVEX C-4	***1994***	***135***	***16***	***128***	***3: each is full integer, logical, and FP (including multiply add)***	
NEC SX/4	1995	400	8+8192	256 variable	**16: 4 integer add/logical, 4 FP multiply/ divide, 4 FP add, 4 shift**	8
CRAY J-90	1995	100	8	64	**4: FP add, FP multiply, FP reciprocal, integer/logical**	
CRAY T-90	1996	500	8	128	**8: FP add, FP multiply, FP reciprocal, integer add, 2 logical shift, population count/parity**	4

The vector functional units include all operation units used by the vector instruction. The functional units are floating point unless stated otherwise. If the processor is a multiprocessor, the entries correspond to the characteristics of one processor. Each vector load-store unit represents the ability to do an independent, overlapped transfer to or from the vector registers.

Note that the most processors use the vector FP multiply and divide units for vector integer multiply and divide, and several of the processors use the same units for FP scalar and FP vector operations. Several of the machines have different clock rates

in the vector and scalar units; the clock rates shown are for the vector units.

The vector instruction take as their input either a pair of vector registers (ADDV) or vector register and scalar register, designated by appending „SV" (ADDSV). In the latter case, the value in the scalar register is used as the input for all operations –the operation ADDSV will add the contents of a scalar register to each element in a vector register. Most vector operations have a vector destination registers, though a few (population count) produce a scalar value, which is stored to a scalar register. The names LV and SV denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is vector register to be loaded or stored; the other operand, is the starting address of vector in memory. In addition to the vector registers, we need to additional special-purpose: the vector-length and vector-mask registers.

Vector Execution Time

The execution time of a sequence of vector operations primarily, depends on three factors: the length of the vectors being operating on, structural hazards among the operations, and the data dependence. Given the vector length and initiation rate, which is the rate at which a vector unit consumes new operands and produces new results, we can compute the time for a single vector instruction. The initiation rate is usually one per clock cycle for individual operations. However, some supercomputers have vector instructions that can produce two or more results per clock, and others have units that may not be fully pipelined .

THE CONVEX C SERIES ARCHITECTURE

We present the specifications of the central processing unit (CPU) of the CONVEX supercomputers.

Within this description, an architectural specification defines the following attributes, as perceived by the programmer and the hardware designer:

- Data representations
- Data operations
- Register sets
  - General registers
  - Communication registers
- Address space
  - Physical address space
  - I/O address space
  - Logical (virtual) address space
- Process structures
- Timers
- Memory management
  - Memory protection mechanisms
  - Caches
- Multiprocessing mechanisms
  - Automatic self-allocating processors
  - Parallel processing mechanisms
  - Forking and spawning mechanisms
- Exception system mechanisms
- Interrupt system mechanisms
- Instruction set

The CONVEX C Series architecture incorporates the following features:

- An integrated vector processor incorporated within the system for high speed operation
- A full range of fixed floating point data types
- A total capacity of four Gbytes of virtual memory. Two Gbytes are available to support large user programs and data, and two Gbytes are available to the operating system.
- Large, high-speed register sets (address, scalar, and vector) that support high-performance operation for address calculations in parallel with scalar and vector calculations.
- Communication registers and multiprocessing structures (in the multiprocessing C Series CPUs).
- Multilevel protection systems that support and separate users, thereby enhancing system reliability and increasing the performance of operating system functions.

All CONVEX C Series CPUs share a common architecture, in most respects. However, some software, such as the Convex OS operating system, use features whose implementation varies among different CONVEX CPUs.

Brief summaries about CONVEX CPUs

❶ Data representations and operations

❷ General registers

❸ Physical address space

❹ Logical (virtual) address space

❺ Communication registers

❻ Process structures

❼ Memory and cache management

❽ Multiprocessor management

❾ Timers

❿ Operating system interrupts

❶❶ Operating system exceptions

❶ Data representations and operations

There are three binary numeric data representations:

- Signed fixed point integer
- Unsigned fixed point integer
- Floating point

The CONVEX processors support four fixed-point integer precision. Signed fixed-point numbers are interpreted by the architecture as two two's complement representations. Integer quantities exist in four lengths:

- Byte – 8 bits
- Halfword – 16 bits
- Word – 32 bits
- Longword – 64 bits

The CONVEX CPUs support both native and LEEE standard floating point number representations in two formats:

- Single-precision word (32 bits)
- Double-precision longword (64 bits)

Both formats are interpreted as binary normalized fractions with an implicit 1 value in the most significant bit of the fraction. The exponent is a biased power of two scales factor.

An address or logical value is treated as an unsigned value. Addresses are 32 bits long and are usually contained in the address registers. For numeric purposes, an address register may be treated as a signed or unsigned 32 bit integer.

Logical (virtual) addresses are byte granular, Instruction operands in memory may begin on any byte boundary that allows all byte locations within a given data type to be used, even though the operands may be unrelated.

Operations are performed in integer and floating point, Floating point operations are performed in native and IEEE modes. The differences are delineated for add, subtract, multiply, divide, square root, compare, and conversion operations.

❷ General registers

There are **three general register sets** and several status registers. The three register sets are partitioned according to the type of operand to be manipulated:

- **Address registers** (eight 32-bit registers: A0-A7);
  - A0 is the stack pointer /SP/
  - A3, A4, and A5 are implicitly used by some trap handlers (for example, page fault, system exception, and so forth)
  - A5 are implicitly used by some additional instructions
  - A6 is the argument pointer /AP/
  - A7 is the frame pointer /FP/
- **Scalar registers** (eight 64-bit registers: S0-S7);
  The S register can contain logical, fixed point integer, or floating point operands.
- **Vector registers** (8 vectors; each containing 128 64-bit register elements)
  (see in the following section)

There are two general status register and three privileged flags.
The **two status registers** are:

- Program counter (PC)
- Processor status word (PSW)

The **three privileged flags** are

- Interrupt on (ION)
- Real-time interrupt on (RT_ION)
- Vector valid flag (VV)

❸ Physical address space

The physical address space in all C Series CPUs spans physical memory. Data access at the program level uses virtual addresses, however, these virtual addresses are translated to physical addresses in hardware. The possible range of these physical addresses describes the physical address space.

All I/O data references by the CPU are memory mapped. There are no explicit I/O instructions. The I/O registers and memory status bits are referenced through the appropriate logical-to-physical address mapping.

Referenced and modified bits determine whether a valid memory read or write has occurred.

❹ Logical (virtual) address space

The memory management unit (MMU) supports the operating system in providing a versatile and reliable virtual memory programming environment, The CONVEX C Series architecture provides 4 Gbytes of virtual memory in its virtual address space partitioned into eight 512-Mbyte segments. Four segments are allocated to the operating system and four segments to the user. The maximum size of a user program (instruction and data) is limited to 2 Gbytes. The operating system data structures and instructions necessary to manage the user program occupy the remaining 2 Gbytes of virtual storage.

Because the address space of the C Series architecture is virtual, an address may be a valid logical address, but the referenced data may or may not be in physical memory. Memory is managed as pages on a fixed-size basis.

To manage memory, the CONVEX system architecture defines and supports these features:

- **Segment** – A logically contiguous 512-Mbyte block of memory.
- **Segment descriptor register (SDR)** – A 32-bit register that contains the pointer to the first-level page table.
- **Page** – A 4-kbyte contiguous block of memory. The bytes within a page are both virtually and physically contiguous.
- **Page frame** – A page that is stored in physical memory.
- **Page table entry (PTE)** – 32 bit entry interpreted by the memory management unit (MMU) or operating system software to determine whether or not a page is resident in main memory.
- **Page table** – A page that contains page table entries (PTEs). A page table begins on an integral page boundary and is contained in one page frame or less. First-level page tables contain PTEs that have pointers to second-level page tables. Second-level page tables contain pointers to physical page frames or thread-level pages.
- **Address translation units (ATU)** – A programmer-invisible address cache that is maintained in hardware. The ATU contains the most recently used virtual-to-physical address translations.

Since the operating system is embedded within the user-virtual address space, it must be protected from the user. The memory protection system protects the user's programs from other users' programs, while supporting time-sharing and operating system structures. This system is based on hierarchical structures called **rings** and:

- Supports embedding the operating system in the user's virtual address space
- Contains certain access violation to the user's process
- Permits implementing the operating system efficiently
- Enhances operating system call processing by reducing the time for context switching

❺ Communication registers

The communication registers set implements CPU communication and synchronization for the multiprocessing C Series architecture:

- 1 024 locations by 64 bits for C200/C3200 Series CPUs
- 1 024 locations by 64 bits for C3400 Series CPUs
- 4 096 locations by 64 bits for C3800 Series CPUs

❻ Process structures

A CONVEX C Series process consists of one or more threads. A process is controlled by maintaining a process stack. These stack entries are called return blocks and contain vital information for controlling the execution of a process.

Resource structures are pre-determined memory locations used to store specific registers, flags, and lock bits. The shared resource structures are memory dual of the communication registers. The stack resource structures are an extension of the shared resource structures, used to push and pop word resource structures. System resource structures manage allocation of available stacks in ring 0 when ring crossings occur.

❼ Memory and cache management

C Series CPUs have implementation-specific cache management mechanisms. These mechanisms purge the instruction, data, ATU, and PTE caches under certain conditions.

Memory interleave is the process of swapping address bits to determine which physical memory bank to use. Through interleaving, a different bank may be accessed on each clock, allowing sequential request to ascending banks to proceed at full speed. The C Series memory subsystems may be interleaved in variety of ways.

❽ Multiprocessor management

Multiprocessing is defined as the creation and scheduling of individual processes on any subcomplex. The multiprocessor management hardware incorporated in each C Series architecture CPU provides the operating system and user with a simple and flexible set of instructions for dynamic CPU allocation, deallocation, and communication.

In order to manage multiple CPUs, the CONVEX C Series architecture defines and supports several attributes:

CPU – One physical central processing unit
CPU complex – The entire set of physical CPUs in a configuration
CPU subcomplex – Any subset of a complex
Thread – Any single instruction stream executing within a process
Process – One or more threads executing within a single logical address space

Each CPU in a C Series architecture complex operates independently as a 64-bit supercomputer. The multiprocessor management hardware binds these CPUs into a tightly coupled set with shared memory that implements a multi-instruction multi-data (MIMD) architecture that provides a parallel execution environment for user applications.

❾ Timers

The multiprocessing C Series architecture provides four timers to permit accurate accounting of CPU execution time and to assist in process scheduling:

- Interval timer – Used interrupt the CPU at a programmable rate (count-down timer)
- Time of century clock – This clock keeps „wall clock time", not user time.
- CPU execution timer – Timer for each process executing.
- Thread timer – Timer for each thread executing.

❿ Operating system interrupts

Interrupts are results of events that occur asynchronously and belong to the system, not to the executing process. When an interrupt occurs, the processor jumps to a particular interrupt handler determined by the interrupt source.

❶❶ Operating system exceptions

Exceptions are invoked when problems occur in a currently executing program, such as arithmetic inconsistencies, address translation faults, or some asynchronous event (such as an interrupt). When an exception occurs, control is transferred to a predetermined address whose value is a function of the exception.

VECTOR OPERATIONS

VECTOR HARDWARE

Four types of register are used in vector operations:

- Vector-accumulator (V) register
- Vector-length (VL) register
- Vector-stride (VS) register
- Vector-merge (VM) register

Vector-accumulator (V) register

A vector register can hold up to 128 64-bit elements. These elements can be integer, logical or floating-point data. Data must be of uniform size and precision. The vector register is used to store arrays of operands. CONVEX C Series machines have eight vector registers (V0-V7).

Vector-length (VL) register

The CONVEX C Series machines have one vector-length register.
The value in a VL register is the number of elements used in subsequent vector operations. The VL may contain any value from 0 –128.
Even thought the VL register has a maximum value of 128, a vector can be any arbitrary length up to the user virtual address space limit of 2 Gbytes. Arrays longer than 128 elements, or arrays with a variable length that could exceed 128, are handled in software by coding a loop around a group of vector instruction that handles up to 128 elements at a time. This is called a **strip mining** and is generated automatically by CONVEX vectorizing compilers.

Vector-stride (VS) register

The 32-bit vector-stride register is used by the load and store instructions. The value in the VS register is the number of bytes from one element of an array in memory to the next sequential element. Strides can be either positive or negative.

Vector-merge (VM) register

The vector-merge register holds a 128-bit mask used for compress, expand, operate-under-mask, and merge instructions. The VM register also stores the results of a vector comparison. If the comparison of corresponding elements in two vector registers is true, the corresponding bit in the VM register is set. Otherwise, the corresponding bit is cleared.

The VM register is often used for this operation:

- Population count (number of successful compares)
- Sparse vector manipulation
- Array compression, expansion and merging
- Vector clipping

How the CONVEX architecture works

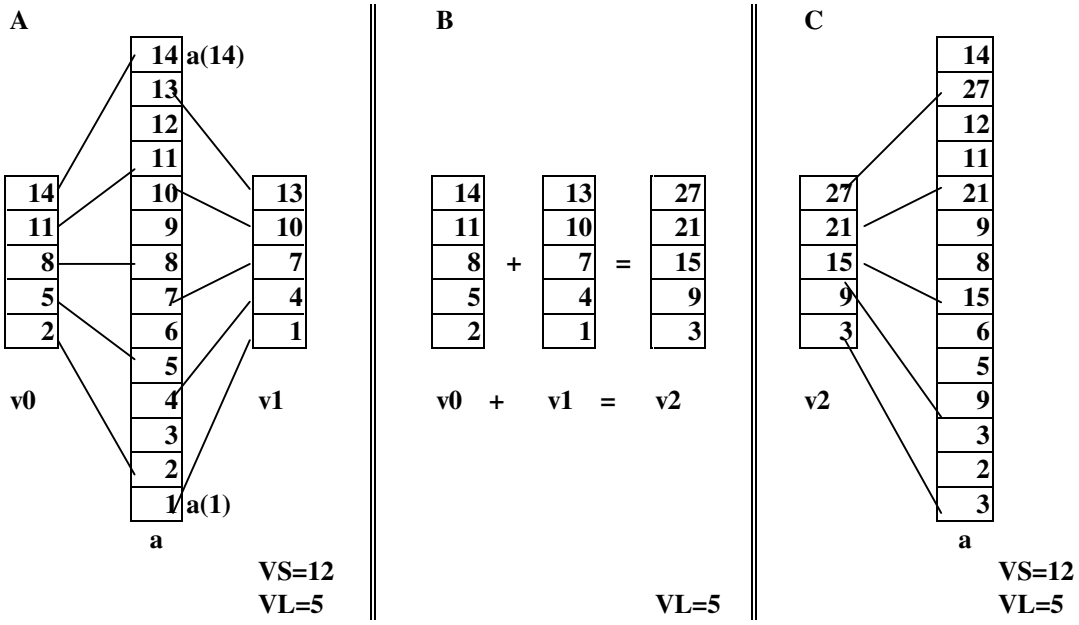
To see how to the vector hardware works, consider the vector operation in the following example.

Example:

```
INTEGER A(14), I
DO I=1, 14, 3
   A(I)=A(I+1)+A(I)
ENDDO
```

The code increments every third elements of array and uses the VS, VL and V registers. The next figure shows the vector operations on array A.

Figure x
Vector operation for A(I)=A(I+1)+A(I)



In A, the CPU sets the vector-stride register (VS) to 12 (the number of bytes between elements of array). In B, the CPU has set the vector-length register (VL) controlling the operation to 5. The VL register controls the loading of elements from array A into vector registers V0 and V1.
The CPU adds the contents of vector registers V0 and V1 and stores the result in V2.
In C, the CPU stores elements of V2 back into array A.

VECTOR INSTRUCTION SET

This section describes some of the assembly-language instructions used vector operations. Assembly-language listings are provided to show how certain FORTRAN statements are vectorized in assembly language. A complete list of the vector instruction set is in the CONVEX Architecture Reference Manual (C Series).

Vector load

The vector load instruction loads the contents of an array stored in memory into a vector register, The data types are byte, half-word, word and long-word. The VS register contains the byte separation of each element that is loaded into the vector register, and the VL register contains the number of array elements to be loaded.

For example, suppose the FORTRAN code on the left in the following example is vectorized.

FORTRAN

```
INTEGER*4 A(25)

DO I=1, 25
    A(I)=A(I) + 4
ENDDO
```

Assembly language

```
ld.w    #25,VL
ld.w    #4,VS
ld.w    A,v0
```

The assembly-language code required to load A into a vector register appears on the right.

The first statement loads the length of array A, which is 25, into VL. The second statement loads 4 into VS because each element in the array requires four bytes for storage and the loop's stride is one. The last statement loads the contents of array A into vector register V0.

The following example shows another example of vector load.

FORTRAN	**Assembly language**
REAL*8 B(100)	ld.w #5,VL
	ld.w #160,VS
DO I=1, 100, 20	ld.l B,v0
B(I)=B(I) + 8.0	
ENDDO	

The FORTRAN code on the left generates the vectorized assembly-language code on the right.
VL contains 5 because only five elements of array B are modified. It is necessary to load all the elements of B into the vector.
Similarly, VS contains 160, because each element requires eight bytes of storage and the loop's stride is 20. The last statement loads five elements of array B into vector register V0.

Some operations that appear to require a load statement use other instruction instead, as shown in the example below.

FORTRAN	**Assembly language**
INTEGER*4 C(100)	ld.w #5,s0
	ld.w #100,VL
DO I=1, 100	ld.w #4,VS
C(I)=5	ste.w s0,C
ENDDO	

The result is a repeated store of a scalar register. The assembly-language instruction for store scalar extended is ***ste***. This instruction uses VS and VL registers in the same way the vector load instruction does: VL specifies the length of array, and VS specifies the number of bytes between each array element that is accessed.

Vector store

The vector store instruction stores the contents of a vector register into an array in memory. The VS register contains the byte separation of each element stored, and VL register contains the number of array elements in the vector register.

Following is an example of vector store.

FORTRAN		**Assembly language**
INTEGER*4 B(100),C(100)	ld.w	#100,VL
		ld.w #4,VS
DO I=1, 100		ld.w B,v0
C(I)=B(I)		st.w v0,C
ENDDO		

The VL register contains 100 because 100 element loaded and stored. The VS register contains 4 because each element requires four bytes for storage and the loop's stride is one. In the example, the vector store and vector load operations use the same VL and VS values.

Following is an example shows another vector store.

FORTRAN		**Assembly language**
INTEGER*4 B(100),C(100)	ld.w	#50,VL
		ld.w #4,VS
DO I=1, 50		ld.w B,v0
C(I*2)=B(I)		ld.w #8,VS
ENDDO		st.w v0,C+4

In this example, the VS register is increased to 8 because only every other element of array C is modified. The loop's stride is 2, and each element of array C requires four bytes for storage. Because the destination of the vector store operation starts at the second element of array C, its base address is increased by one element, or four bytes.

Binary vector operations

Four binary operations used in vector arithmetic are
- addition,
- division,
- multiplication,
- subtraction.

Additional binary operators used for logical operations are
- and,
- or,
- xor.

Both operands of these operators can be vectors, or one can be a scalar and the other a vector. All operators use the VL register to determine the number of vector elements to use computations.

The following example shows the use of the vector add operator.

FORTRAN		**Assembly language**
INTEGER*4 B(100),C(100)	ld.w	#50,VL
		ld.w #4,VS
DO I=1, 50		ld.w C,v0
C(I)=C(I)+B(I)		ld.w B,v1
ENDDO		add.w v0,v1,v2
		st.w v2,C

Arrays B and C are loaded into vector registers, wich are added together, The result is stored in a third vector register, The fourth and fifth or sixth statements can be chained together because they map to different functional units.

The FORTRAN code in the following example computes the product of a vector and a scalar.

FORTRAN	**Assembly language**
INTEGER*4 C(100)	ld.w #100,VL
	ld.w #8,s0
DO I=1, 100	ld.w #4,VS
C(I)=C(I) * 8	ld.w C,v0
ENDDO	mul.w v0,s0,v1
	st.w v1,C

Array C is loaded into v0, and v0 is multiplied by the scalar register s0. The result is stored in v1 and then returned to array C.

Vector reductions

Reduction operations reduce a vector to a scalar, A reduction operation requires two inputs: ***a scalar register and vector register***. A scalar input is provided so that reduction operator can be performed for vectors greater than 128 elements.

Mathematically, reduction operations are the sum reduction (sum) and multiply or product reduction (prod). Additional reduction operations are provided to implement the FORTRAN MAX and MIN intrinsics, as well as reduction using logical operators such as .AND., .OR., and .NEQV.

The example below generates a sum reduction.

FORTRAN	**Assembly language**
INTEGER*4 C(100)	ld.w #0,s0
	ld.w #100,VL
ISUM = 0	ld.w #4,VS
DO I=1, 100	ld.w C,v0
C(I)=C(I) * 8	sum.w v0
ENDDO	st.w s0,ISUM

During a vector reduction, a vector register is paired with a scalar register (Vi is paired with Si). In this example, s0 is the scalar register that corresponds to ISUM, and v0 is the vector that is reduced. This statement

```
sum.w v0
```

can be replaced with

```
sum.w s0
```

Both statement produce the same results.

The FORTRAN code in the following example computes vector's maximum.

FORTRAN	**Assembly language**
INTEGER*4 C(100), C(MAX)	ld.w C(1),s0
	ld.w #100,VL
CMAX = C(1)	ld.w #4,VS
DO I=1, 100	ld.w C,v0
CMAX=JMAX(0) (CMAX, C(I))	max.w v0
ENDDO	st.w s0,CMAX

This code performs the way the code in the sum reduction shown above does, except the vector's maximum is returned.

CHAINING

By chaining vector operations, the CPU can use the output of one vector instruction as input for the next. Addition and multiplication can be chained so that an addition begins while the products of two vectors are being computed. These concurrent, or pipelined, events greatly improve performance.

In the following example, a dot-product operation requires the sum of a series of products.

Example

FORTRAN	**Assembly language**
INTEGER D(100), N(100), A(100), I. SUM	ld.w #0,s0
	ld.w #100,VL
SUM = 0	ld.w #4,VS
DO I=N(I)*A(I)	ld.w A,v1
SUM = SUM + D(I)	ld.w N,v2
ENDDO	mul.w v2,v1,v0
	st.w v0,D
	sum.v v0
	st.w s0,SUM

In this example, the summation is chained with the multiplication. Pipelining uses multiple functional units of the CPU to perform a specific set of operations, and the functional units allow the multiplication and addition operations to overlap.

Vector comparisons

The three vector comparison instructions are less-than, less-than-or-equal, and equal. All other logical operators are obtained by taking the complement of these three instructions. For example, greater-than is the complement of less-than-or-equal.

The result of a vector comparison is stored in vector-merge register. This register has 128 bits, each one corresponding to an element in a vector register. If the comparison of two elements is true, the corresponding bit in the VM register is set; otherwise the bit is cleared.

Consider the vector comparison in the following example.

FORTRAN	**Assembly language**
INTEGER*4 A(100), B(100)	ld.w #100,VL
	ld.w #4,VS
DO I=1, 100	ld.w A,v0
IF (A(I) .LE. B(I))J = I	ld.w B,v1
ENDDO	le.w v0,v1

The arrays are loaded into vectors, and the vectors are compared.

Vector operations under mask

The CONVEX C2 and C3 Series computers can perform vector operations under mask. The CONVEX C1 Series computers can perform vector operations and mask operations, but multiple vector instructions must replace an individual vector operation under mask on the CONVEX C2 and C3 Series computers.

Most vector operations can operate under mask. A vector-merge register bit is associated with each vector register element. When an operation is performed under mask, each element is either included or excluded from the operation based on the state of its corresponding VM bit.

In this mode, the bit of the VM register corresponding to each vector element is examined to either enable or disable that vector element from the operation.

There are two forms of vector operations under mask:

- ✓ ***True-Elements*** with VM bit equal to one are included.
  Instructions of this type have a .t suffix, such as:
  `add.w.t`
- ✓ ***False-Elements*** with VM bit equal to zero are included.
  Instructions of this type have a .f suffix, such as:
  `div.b.f`

The following statement:

add.w v0,v1,v2

adds all elements (restricted by vector length) of v0 and v1, placing the results in v2.

The following statement:

add.w.t v0,v1,v2

adds only elements, whose corresponding VM bits are one. Elements of v2 whose corresponding VM bits are zero remain unmodified.

The complement of VM bits is used for .f, as in the statement:

add.w.f v0,v1,v2

This version operates only on vector elements whose corresponding VM bits are zero.

For remaining examples of operations under mask, assume these values before instructions are executed:

v0 = 0 1 2 3 4 5 VL = 6
v1 = 6 7 8 9 2 3 VM = 0 1 1 0 0 1
v2 = 5 5 5 5 5 5

The following statement:

add.w.t v0,v1,v2
produces:

v2 = 5 8 10 5 5 8

The following statement:

add.w.f v0,v1,v2
produces:

v2 = 6 5 5 12 6 5

The following FORTRAN code is an example of using operations under mask.

FORTRAN	**Assembly language**
DO I=1, 100	ld.w A,v0
IF(A(I) .EQ. B(I)) THEN	ld.w B,v1
C(I) = D(I)	eq.w v0,v1
ENDIF	ld.w D,v0
ENDDO	st.w.t v0,C

Ignoring the length and stride setup, the code on the left can be vectorized with the assembly-language code shown on the right.

Vector-merge register operations

The merge, mask, compress and expand operations use the vector-merge register to control the selection of elements in the vector operands.

Merge and mask

The merge and mask instructions take two operands and produce a vector as the result. The two operands can be two vectors or a vector and a scalar. The merge and mask instructions differ only in the way the indexes of the operands are used to create the result vector. For merge, the indexes of the operands are incremented only if that particular register is selected by VM. For mask, element **n** of the result vector is element **n** of either the left or right operand.

Compress

The compress instruction uses the VM register to extract elements selectively from one vector register and place the elements in another vector register. Either zeros or ones of VM can be used by specifying the instruction's **`.f`** (false) or **`.t`** (true) version, respectively. Only elements with the corresponding VM bit set (clear for **`.f`**) are moved from the source vector to the destination vector. This creates a destination vector with a number of elements equal to the number of bits set (or cleared) in VM.

Expand

The expand instruction is only available on the CONVEX C2 and C3 Series computers. This instruction uses the VM register to extract elements from one vector register and selectively place the elements in another vector register. Either zeros or ones of VM can be used by specifying the instruction's **`.f`** or **`.t`** (false or true) version, respectively. Only elements with the corresponding VM bit set (clear for **`.f`**) are loaded into destination vector. Other elements in the destination vector corresponding to clear VM bits (set for .**`f`**) are skipped over. The expand instructions creates a destination vector with VL elements, including a number of elements of the source vector equal to the number of bits set (or clear) in the VM register.

Examples

The examples below show how this instructions work.

The vector merge, mask, compress and expand instructions have either a single true version, or both **.t** and **.f** (true and false) versions. You can use either the ones or the zeros (**.t** or **.f**) of VM. If you use .t, when the appropriate bit of VM is one, the second operand is selected.

Assume these values before the instructions of each example are executed:

```
V0 = 1 2 3 4 5 6          V5 = 7 7 7 7 7 7          VL = 6

V1 = a b c d e f          VM = 0 1 1 0 0 1          S1 = 8
```

Compressing V0 produces:

```
cprs.t V0,V5 = 2 3 6 7 7 7
cprs.f V0,V5 = 1 4 5 7 7 7
```

Expanding V0 produces:

```
xpnd.t V0,V5 = 7 1 2 7 7 3
xpnd.f V0,V5 = 1 7 7 2 3 7
```

Masking V0 produces:

```
mask.t V0,V1,V5 = 1 b c 4 5 f
mask.t V1,V0,V5 = a 2 3 d e 6
mask.t V0,S1,V5 = 1 8 8 4 5 8
```

Merging V0 and V1 produces:

```
VL = 12       VM = 0 1 1 0 0 1 0 0 0 1 1 1
merg.t V0,V1,V5 = 1 a b 2 3 c 4 5 6 d e f
```

Merging V0 and S1 with the previous VL and VM produces:

```
merg.t V0,S1,V5 = 1 8 8 2 3 8 4 5 6 8 8 8
merg.f V0,S1,V5 = 8 1 2 8 8 3 8 8 8 4 5 6
```

Examples of vector operations

This section shows examples of common vector operations.
In the examples, if is an array, then Va is the vector in which a is stored; Vb(5) is the fifth element of vector Vb; and VM<6> is the sixth bit of the VM register.

Embedded IF statement

The vector operations used in this example are conditional test and masking.

Vector operations often use conditional tests. The logical test are and, equal, less-than-or-equal, less-than, greater-than-or-equal, greater-than, or, and exclusive-or. The CPU places the results of a vector comparison in the VM register, with the corresponding bit set if the result is true. If the comparison is equal and V0(5) is the same as V1(5), then VM<5> equals 1.
The vector mask operation restricts the elements altered by a vector assignment operation to those specified by bits set in the VM register. In the vector mask sum V1=V2+V3, for example, V1(5) is assigned a value only if VM<5> is set.

The results of the conditional in the loop in the following example cannot be determined until the program is executed.

Example

```
SUBRUTINE EMBED(A, B, C)
INTEGER A(100), B(100), C(100), D(100), I

DO I = 1, 100
   D(I) = I ! initialize array
ENDDO

DO I = 1, 100
   IF(A(I) .EQ. B(I)) D(I)=C(I)
ENDDO

PRINT *, D(I), I=1, 100)
END
```

Array A is loaded into Va, and array B is loaded into Vb. The two vectors are compared, and the result is strored in VM. The VM register controls the assignment operation. Vc(i) is assigned to Vd(i). Finally, when VM<i> is one, Vd is stored in D.

Indirect array addressing

The vector operations used in this example are gather and scatter.
Gather loads values from an array into a vector register. The operands come from various locations in the array. For example, in random gather moves elements from A to Va, A(5) may be placed in Va(10) while A(10) is copied into Va(2).
Scatter copies elements from a vector register into various location in an array.

The gather and scatter vector operations are used when the elements of an array are indirectly addressed. The code in the following example uses indirect addressing.

Example

```
SUBRUTINE GATHER(A, IA, B, IB)
INTEGER I, A(100), IA(100), B(100), IB(100)

DO I = 1, 100
   A(IA(I)) = B(IB(I))+1
ENDDO

PRINT *, (A(I), I=1, 100)
RETURN
END
```

Bibliography

1. CONVEX Architecture Reference Manual (C Series)
/Sixth Edition, April, 1992 – Convex Computer Corporation/
/Convex Press, Richardson, Texas United States of America – DHW-300/

1. CONVEX FORTRAN Optimization Guide
/Fourth Edition, November, 1992 – Convex Computer Corporation/
/Convex Press, Richardson, Texas United States of America – DSW-034/

CONVEX CXpa User's Guide 3rd edition March 1993
CONVEX Press
Richardson, Texas USA

CONVEX CXpa Reference 1st edition March 1993

Dr. Rezső Nagy - Magda Kovács:
Optimization and Profiling on Vector Computer CONVEX 3820
TEMPUS Project S_JEP 8333-94

Dr. Rezső Nagy - Etelka Kovács:
Digital Image Processing Exercises in the Spatial Domain on Vector Computer CONVEX 3820
TEMPUS Project S_JEP 8333-94

Magda Kovács - Ágnes Burián:
Digital Image Processing Exercises in the Fourier Domain using VECLIB on Vector Computer CONVEX 3820
TEMPUS Project S_JEP 8333-94