6.0 Objective

In this lesson we will about various types of concurrent processor. To study vector processor how pipelining is implemented in vector processor through the instruction format, functional unit. To provides a general overview of the architecture of a vector computer which includes an introduction to vectors and vector arithmetic, a discussion of performance measurements used to evaluate this type of machine. Various models for memory organization for the vector processor are also discussed. We will also study about multiple instruction issue machine which include VLIW, EPIC etc .

6.1 Introduction

The Concurrent Processors must be able to execute multiple instructions at the same time. Concurrent processors must be able to make simultaneous accesses to memory and to simultaneously execute multiple operations. Concurrent processors depend on sophisticated compilers to detect various types of instruction level parallelism that exist within a program. They are classified as

- Vector processors
- SIMD and small clustered MIMD
- Multiple instruction issue machines

    Superscalar (run time schedule)

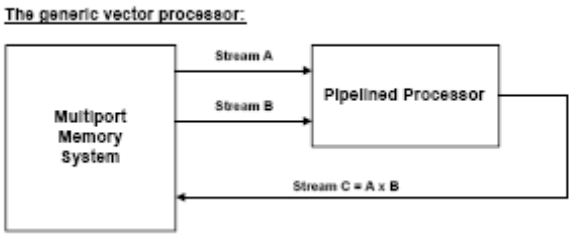
    VLIW (compile time schedule)

    EPIC

    Hybrids

A Vector processor is a processor that can operate on an entire vector in one instruction. The operands to the instructions are complete vectors instead of one element.

Vector processors reduce the fetch and decode bandwidth as the numbers of instructions fetched are less.



They also exploit data parallelism in large scientific and multimedia applications. Based on how the operands are fetched, vector processors can be divided into two categories - in memory-memory architecture operands are directly streamed to the functional units from the memory and results are written back to memory as the vector operation proceeds. In vector-register architecture, operands are read into vector registers from which they are fed to the functional units and results of operations are written to vector registers.

Many performance optimization schemes are used in vector processors. Memory banks are used to reduce load/store latency. Strip mining is used to generate code so that vector operation is possible for vector operands whose size is less than or greater than the size of vector registers.

Various techniques are used for fast accessing these include

— Vector chaining - the equivalent of forwarding in vector processors - is used in case of data dependency among vector instructions.

— Special scatter and gather instructions are provided to efficiently operate on sparse matrices.

Instruction set has been designed with the property that all vector arithmetic instructions only allow element N of one vector register to take part in operations with element N from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel lanes. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. The reason behind the declining popularity of vector processors is their cost as compared to multiprocessors and superscalar processors. The reasons behind high cost of vector processors are

• Vector processors do not use commodity parts. Since they sell very few copies, design cost dominates overall cost.

• Vector processors need high speed on-chip memories which are expensive.

• It is difficult to package the processors with such high speed. In the past, vector manufactures have employed expensive designs for this.

• There have been few architectural innovations compared to superscalar processors to improve performance keeping the cost low.

Vector processing has the following semantic advantages.

• Programs size is small as it requires less number of instructions. Vector instructions also hide many branches by executing a loop in one instruction.

• Vector memory access has no wastage like cache access. Every data item requested by the processor is actually used.

• Once a vector instruction starts operating, only the functional unit(FU) and the register buses feeding it need to be powered. Fetch unit, de-code unit, ROB etc can be powered off. This reduces the power usage.

6.2 Vector processor

The vector computer or **vector processor** is a machine designed to efficiently handle arithmetic operations on elements of arrays, called *vectors*. Such machines are especially useful in high-performance scientific computing, where matrix and vector arithmetic are quite common. The Cray Y-MP and the Convex C3880 are two examples of vector processors used today.

Vectors and vector arithmetic

A *vector*, *v*, is a list of elements

v = (v1, v2, v3, ..., vn),

transposed. The *length* of a vector is defined as the number of elements in that vector; so the length of *v* is *n*. As far as a vector to a computer program, we declare it as an 1-D array. In Fortran, we declare *v* by the statement

```
DIMENSION  V(N)
```

where N is an integer variable holding the value of the length of the vector.

Arithmetic operations may be performed on vectors. Two vectors are added by adding corresponding elements:

s = x + y = (x1+y1, x2+y2, ..., xn+yn).

In Fortran, vector addition could be performed by the following code

```
DO  I=1,N
  S(I) = X(I) + Y(I)
ENDDO
```

where *s* is the vector representing the final sum and S, X, and Y have been declared as arrays of dimension N. This operation is sometimes called *elementwise* addition. Similarly, the subtraction of two vectors, *x – y*, is an elementwise operation.

The stages of a floating-point operation

Consider the steps or stages involved in a floating-point addition on a sequential machine with IEEE arithmetic hardware: *s = x + y*.

- [A:] The exponents of the two floating-point numbers to be added are compared to find the number with the smallest magnitude.
- [B:] The significand of the number with the smaller magnitude is shifted so that the exponents of the two numbers agree.
- [C:] The significands are added.
- [D:] The result of the addition is normalized.
- [E:] Checks are made to see if any floating-point exceptions occurred during the addition, such as overflow.
- [F:] Rounding occurs.

Step	**A**	**B**	**C**	**D**	**E**	**F**
x	0.1234E4	0.12340E4				
y	-0.5678E3	-0.05678E4				
s			0.066620E4	0.66620E3	0.66620E3	0.6662E3

Figure 6.2 : An example showing the stages of a floating-point addition: $s = x + y$.

Figure 6.2 shows the step-by-step example of such an addition. The numbers to be added are `x = 1234.00` and `y = -567.8`. Now consider this scalar addition performed on all the elements of a pair of vectors (arrays) of length `n`. Each of the six stages needs to be executed for every pair of elements. If each stage of the execution takes `t` units of time, then each addition takes `6*t` units of time (not counting the time required to fetch and decode the instruction itself or to fetch the two operands). So the number of time units required to add all the elements of the two vectors in a serial fashion would be `Ts = 6*n*t`. These execution stages are shown in figure 6.3 with respect to time.

Time:	***t***	**2 *t***	**3 *t***	**4 *t***	**5 *t***	**6 *t***	**7 *t***	**8 *t***
Step								
A	*x1* + *y1*						*x2* + *y2*	

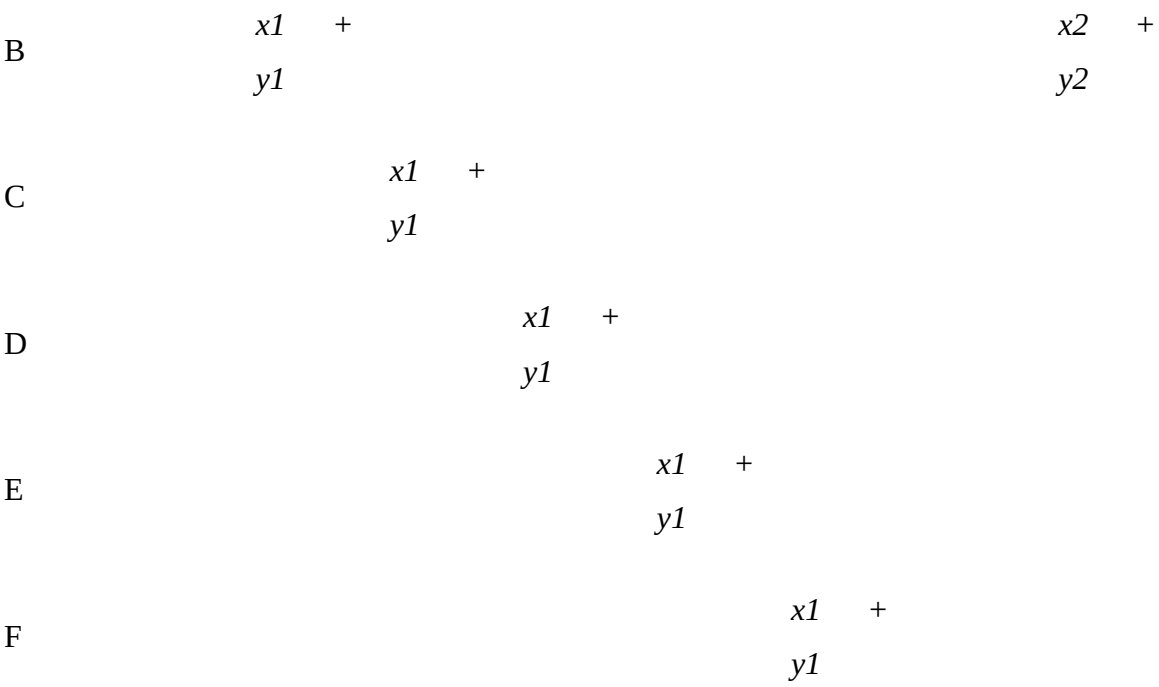
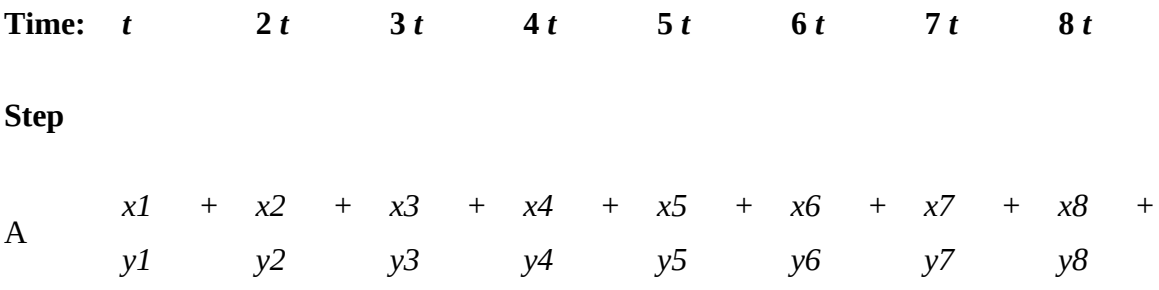


Figure 6.3 : Scalar floating-point addition of vector elements.

An arithmetic pipeline

Suppose the addition operation described in the last subsection is pipelined; that is, one of the six stages of the addition for a pair of elements is performed at each stage in the pipeline. Each stage of the pipeline has a separate arithmetic unit designed for the operation to be performed at that stage. Once stage A has been completed for the first pair of elements, these elements can be moved to the next stage (B) while the second pair of elements moves into the first stage (A). Again each stage takes t units of time. Thus, the flow through the pipeline can be viewed as shown in figure 6.4



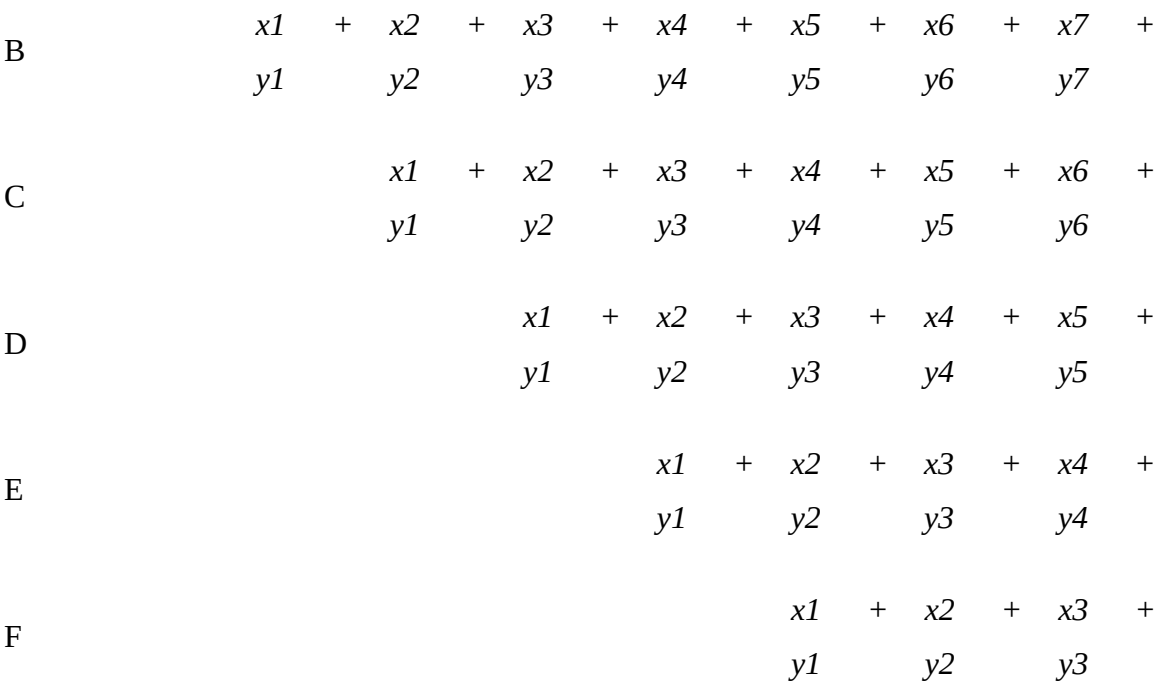


Figure 6.4: Pipelined floating-point addition of vector elements.

Observe that it still takes `6*t` units of time to complete the sum of the first pair of elements, but that the sum of the next pair is ready in only `t` more units of time. And this pattern continues for each succeeding pair. This means that the time, `Tp`, to do the pipelined addition of two vectors of length `n` is

`Tp = 6*t + (n-1)*t = (n + 5)*t`.

The first `6*t` units of time are required to *fill the pipeline* and to obtain the first result. After the last result, `xn + yn`, is completed, the pipeline is emptied out or *flushed*.

Comparing the equations for `Ts` and `Tp`, it is clear that

`(n + 5)*t < 6*n*t`, for `n > 1`.

Thus, this pipelined version of addition is faster than the serial version by almost a factor of the number of stages in the pipeline. This is an example of what makes vector processing more efficient than scalar processing. For large `n`, the pipelined addition for this sample pipeline is about six times faster than scalar addition.

6.2.1 Vector Functional unit

Vector Processing Requirements

A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each element in a vector is a scalar quantity, which may be a floating point number, an integer, a logical value or a character. A vector processor consists of a scalar processor and a vector unit, which could be thought of as an independent functional unit capable of efficient vector operations.

Vector Hardware

Vector computers have hardware to perform the vector operations efficiently. Operands can not be used directly from memory but rather are loaded into registers and are put back in registers after the operation. Vector hardware has the special ability to overlap or pipeline operand processing.

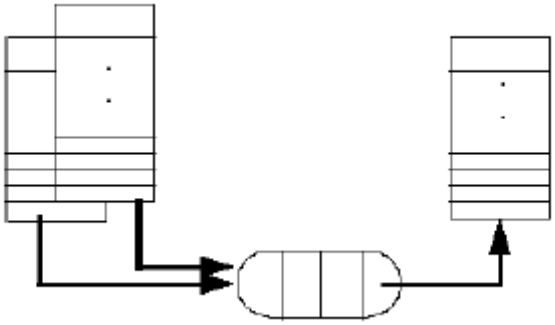


Figure 6.5 Vector Hardware

Vector functional units pipelined, fully segmented each stage of the pipeline performs a step of the function on different operand(s) once pipeline is full, a new result is produced each clock period (cp).

Pipelining

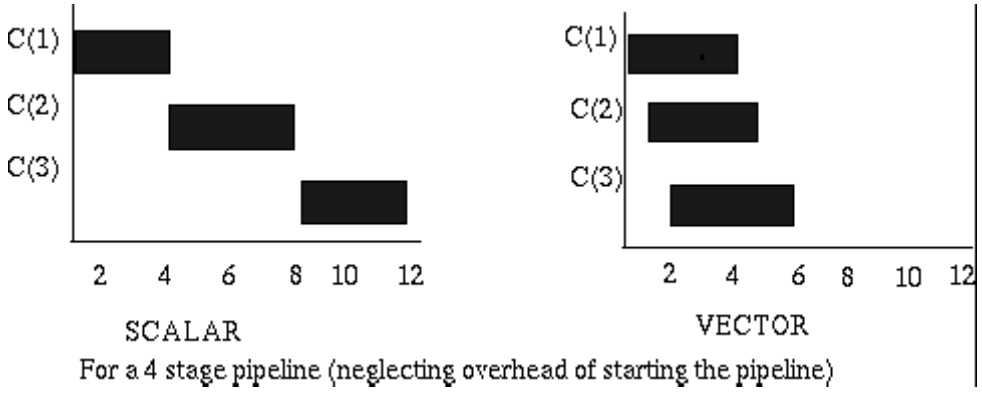
The pipeline is divided up into individual segments, each of which is completely independent and involves no hardware sharing. This means that the machine can be working on separate operands at the same time. This ability enables it to produce one result per clock period as soon as the pipeline is full. The same instruction is obeyed repeatedly using the pipeline technique so the vector processor processes all the elements of a vector in exactly the same way. The pipeline segments arithmetic operation such as floating point multiply into stages passing the output of one stage to the next stage as input. The next pair of operands may enter the pipeline after the first stage has processed

the previous pair of operands. The processing of a number of operands may be carried out simultaneously.

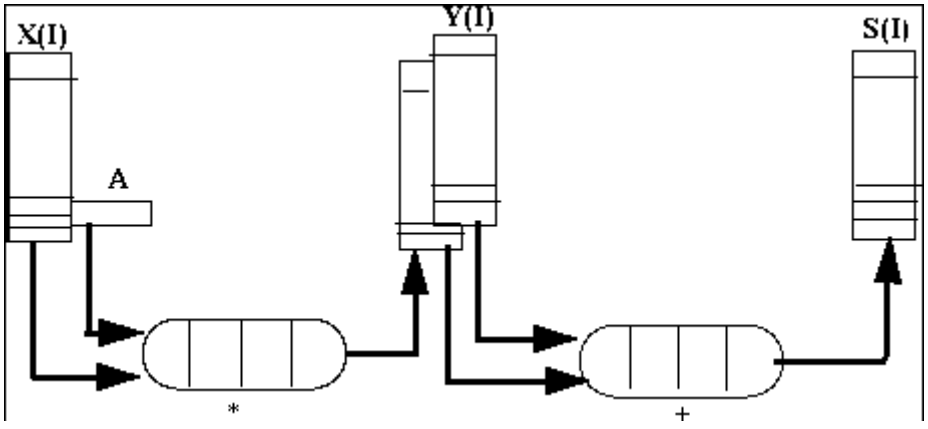
The loading of a vector register is itself a pipelined operation, with the ability to load one element each clock period after some initial startup overhead.

Chaining

Theoretical speedup depends on the number of segments in the pipeline so there is a direct relationship between the number of stages in the pipeline you can keep full and the performance of the code. The size of the pipeline can be increased by chaining thus the Cray combines more than one pipeline to increase its effective size. Chaining means that the result from a pipeline can be used as an operand in a second pipeline as illustrated in the next diagram



For a 4 stage pipeline (neglecting overhead of starting the pipeline)



S(I) = A * X(I) + Y(I)

Figure Pipeline Chaining

This example shows how two pipelines can be chained together to form an effectively single pipeline containing more segments. The output from the first segment is fed directly into the second set of segments thus giving a resultant effective pipeline length of 8. Speedup (over scalar code) is dependent on the number of stages in the pipeline. Chaining increases the number of stages

Most vector architectures have more than one pipeline; they may also contain different types of pipelines. Some vector architectures provide greater efficiency by allowing the output of one pipeline to be *chained* directly into another pipeline. This feature is called *chaining* and eliminates the need to store the result of the first pipeline before sending it into the second pipeline. Figure 14.5 demonstrates the use of chaining in the computation of a *saxpy* vector operation:

a*x + y,

where *x* and *y* are vectors and **a** is a scalar constant.

Vector Chaining used to compute a*x + y

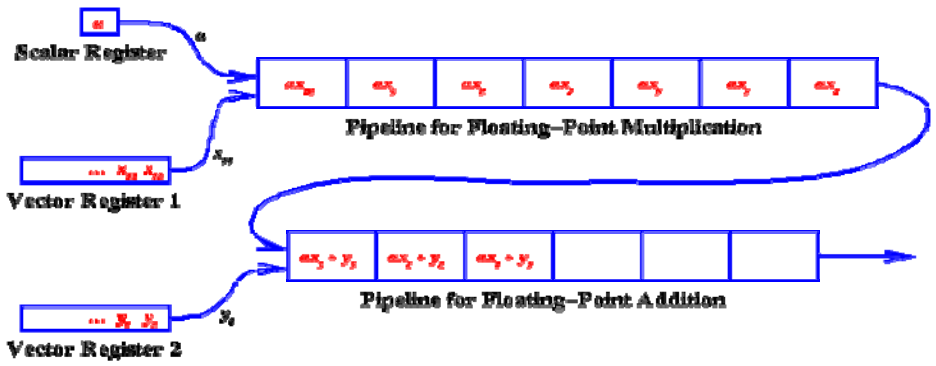


Figure 6.9 Vector chaining used to compute a scalar value **a** times a vector *x*, adding the elements the resultant vector to the elements of a second vector *y* (of the same length).

Chaining can double the number of floating-point operations that are done in x units of time. Once both the multiplication and addition pipelines have been filled, one floating-point multiplication and one floating-point addition (a total of two floating-point operations) are completed every x time units. Conceptually, it is possible to chain more than two functional units together, providing an even greater speedup. However this is rarely (if ever) done due to difficult timing problems.

6.2.2 Vector instruction /operation

Vector Instructions

The ISA of a scalar processor is augmented with vector instructions of the following types:

Vector-vector instructions:

f1: Vi -> Vj (e.g. MOVE Va, Vb)

f2: Vj x Vk -> Vi (e.g. ADD Va, Vb, Vc)

Vector-scalar instructions:

f3: s x Vi -> Vj (e.g. ADD R1, Va, Vb)

Vector-memory instructions:

f4: M -> V (e.g. Vector Load)

f5: V -> M (e.g. Vector Store)

Vector reduction instructions:

f6: V -> s (e.g. ADD V, s)

f7: Vi x Vj ->s (e.g. DOT Va, Vb, s)

Scatter and gather operations

Sometimes, only certain elements of a vector are needed in a computation. Most vector processors are equipped to pick out the appropriate elements (a *gather* operation) and put them together into a vector or a vector register. If the elements to be used are in a regularly-spaced pattern, the spacing between the elements to be gathered is called the *stride*. For example, if the elements

```
x1, x5, x9, x13, ..., x[4*floor((n-1)/4)+1]
```

are to be extracted from the vector

```
( x1, x2, x3, x4, x5, x6, ..., xn )
```

for some vector operation, we say the stride is equal to `4`. A *scatter* operation reformats the output vector so that the elements are spaced correctly. Scatter and gather operations may also be used with irregularly-spaced data.

f8: M x Va -> Vb (e.g. gather)

f9: Va x Vb -> M (e.g. scatter)

Gather and scatter are used to process sparse matrices/vectors. The gather operation, uses a base address and a set of indices to access from memory "few" of the elements of a

large vector into one of the vector registers. The scatter operation does the opposite. The masking operations allows conditional execution of an instruction based on a "masking" register.

Masking instructions:

fa: Va x Vm->Vb (e.g. MMOVE V1, V2, V3)

Gather and scatter are used to process sparse matrices/vectors. The gather operation, uses a base address and a set of indices to access from memory "few" of the elements of a large vector into one of the vector registers. The scatter operation does the opposite. The masking operation allows conditional execution of an instruction based on a "masking" register.

- A Boolean vector can be generated as a result of comparing two vectors, and can be used as a masking vector for enabling and disabling component operations in a vector instruction.
- A compress instruction will shorten a vector under the control of a masking of vector.
- A merge instruction combines two vectors under the control of a masking vector.

In general machine operation suitable for pipelining should have the following properties:

- Identical Processes (or functions) are repeatedly invoked many times, each of which can be subdivided into subprocesses (or sub functions)
- Successive Operands are fed through the pipeline segments and require as few buffers and local controls as possible.
- Operations executed by distinct pipelines should be able to share expensive resources, such as memories and buses in the system.
- The **operation code** must be specified in order to select the functional unit or to reconfigure a multifunctional unit to perform the specified operation.
- For a memory reference instruction, the **base addresses** are needed for both source operands and result vectors. If the operands and results are located in the vector register file, the designated vector registers must be specified.
- The **address increment** between the elements must be specified.
- The **address offset** relative to the base address should be specified. Using the base address and the offset the relative effective address can be calculated.

- The **Vector length** is needed to determine the termination of a vector instruction.
- The Relative Vector/Scalar Performance and Amdahl Law

The major hurdle for designing a vector unit is to ensure that the flow of data from memory to the vector unit will not pose a bottleneck. In particular, for a vector unit to be effective, the memory must be able to deliver one datum per clock cycle. This is usually achieved using pipelining using the C-access memory organization (concurrent access) or the S-access memory organization (simultaneous access), or a combination thereof.

Vector-register vector processors

If a vector processor contains vector registers, the elements of the vector are read from memory directly into the vector register by a *load vector* operation. The vector result of a vector operation is put into a vector register before it is stored back in memory by a *store vector* operation; this permits it to be used in another computation without needing to be reread, and it allows the store to be overlapped by other operations. On these machines, all arithmetic or logical vector operations are register-register operations; that is, they are only performed on vectors that are already in the vector registers. For this reason, these machines are called *vector-register* vector processors.

Memory-memory vector processors

Another type of vector processor allows the vector operands to be fetched directly from memory to the different vector pipelines and the results to be written directly to memory; these are called *memory-memory* vector processors. Because the elements of the vector need to come from memory instead of a register, it takes a little longer to get a vector operation started; this is due partly to the cost of a memory access. One example of a *memory-memory* vector processor is the CDC Cyber 205.

Because of the ability to overlap memory accesses and the possible reuse of vector processors, vector-register vector processors are usually more efficient than memory-memory vector processors. However as the length of the vectors in a computation increase, this difference in efficiency between the two types of architectures is diminished. In fact, the memory-memory vector processors may prove more efficient if the vectors are long enough. Nevertheless, experience has shown that shorter vectors are more commonly used.

Comparison - Vector and Scalar Operations

A scalar operation works on only one pair of operands from the S register and returns the result to another S register whereas a vector operation can work on 64 pairs of operands together to produce 64 results executing only one instruction. Computational efficiency is achieved by processing each element of a vector identically eg initializing all the elements of a vector to zero.

A vector instruction provides iterative processing of successive vector register elements by obtaining the operands from the first element of one or more V registers and delivering the result to another V register. Successive operand pairs are transmitted to a functional unit in each clock period so that the first result emerges after the start up time of the functional unit and successive results appear each clock cycle.

Vector overhead is larger than scalar overhead, one reason being the vector length which has to be computed to determine how many vector registers are going to be needed (i.e., the number of elements divided by 64).

Each vector register can hold up to 64 words so vectors can only be processed in 64 element segments. This is important when it comes to programming as one situation to be avoided is where the number of elements to be processed exceeds the register capacity by a small amount e.g., a vector length of 65. What happens in this case is that the first 64 elements are processed from one register, the 65th element must then be processed using a separate register, after the first 64 elements have been processed. The functional unit will process this element in a time equal to the start up time instead of one clock cycle hence reducing the computational efficiency.

There is a sharp decrease in performance at each point where the vector length spills over into a new register.

The Cray can receive a result by a vector register and retransmit it as an operand to a subsequent operation in the same clock period. In other words a register may be both a result and an operand register which allows the chaining of two or more vector operations together as seen earlier. In this way two or more results may be produced per clock cycle.

Parallelism is also possible as the functional units can operate concurrently and two or more units may be co-operating at once. This combined with chaining, using the result of one functional unit as the input of another, leads to very high processing speeds.

Scalar and vector processing examples

```
DO 10 I = 1, 3
JJ(I) = KK(I)+LL(I)
10 CONTINUE
```

A generic vector processor

Vector registers

Some vector computers, such as the Cray Y-MP, contain *vector registers*. A general purpose or a floating-point register holds a single value; vector registers contain several elements of a vector at one time. For example, the Cray Y-MP vector registers contain 64 elements while the Cray C90 vector registers hold 128 elements. The contents of these registers may be sent to (or received from) a vector pipeline one element at a time.

Scalar registers

Scalar registers behave like general purpose or floating-point registers; they hold a single value. However, these registers are configured so that they may be used by a vector pipeline; the value in the register is read once every `tau` units of time and put into the pipeline, just as a vector element is released from the vector pipeline. This allows the elements of a vector to be operated on by a scalar. To compute

```
y = 2.5 * x,
```

the `2.5` is stored in a scalar register and fed into the vector multiplication pipeline every `tau` units of time in order to be multiplied by each element of `x` to produce `y`.

6.2.4 Vector computing performance

For typical vector architectures, the value of `tau` (the time to complete one pipeline stage) is equivalent to one clock cycle of the machine On some machines, it may be equal to two or more clock cycles.. Once a pipeline like the one shown in figure 3 has been filled, it generates one result for each `t` units of time, that is, for each clock cycle. This means the hardware performs one floating-point operation per clock cycle.

Let `k` represent the number of `t` time units the same sequential operation would take (or the number of stages in the pipeline). Then the time to execute that sequential operation on a vector of length `n` is

```
Ts = k*n*t,
```

and the time to perform the pipelined version is

$Tp = k*t + (n-1)*t = (n + k - 1)*t$.

Again for $n > 1$, $Ts > Tp$.

A *startup time* is also required; this is the time needed to get the operation going. In a sequential machine, there may some overhead required to set up a loop to repeat the same floating-point operation for an entire vector; the elements of the vector also need to be fetched from memory. If we let Ss be the number of t time units for the sequential startup time, then Ts must include this time:

$Ts = (Ss + k*n)*t$.

In a pipelined machine, the flow from the vector registers or from memory to the pipeline needs to be started; call this time quantity Sp. Another overhead cost, $k*t$ time units, is the time needed to initially fill the pipeline. Hence, Tp must include the startup time for the pipelined operation; thus,

$Tp = (Sp + k)*t + (n - 1)*t$

or

$Tp = (Sp + k + n - 1)*t$.

As the length of the vector gets larger (as n goes to infinity), the startup time becomes negligible in both cases. This means that

$Ts \dashrightarrow k*n*t$

while

$Tp \dashrightarrow n*t$.

Thus, for large n, Ts is k times larger than Tp.

There are a number of other terms to describe the performance of vector processors or vector computers. The following list introduces some of these:

- Rn: For a vector processor, the number of Mflops obtainable for a vector of length n.
- $R\_infinity$: The asymptotic number of Mflops for a given vector computer as the length of the vectors gets large. This means that the startup time would be completely negligible. When the vectors are very long, there should be a result from the pipeline at every tau units of time or every clock cycle. So the number

of floating-point operations that can be completed in one second is *1.0/tau*; dividing this result by one million produces the result in Mflops.

- *n_1/2*: The length, *n*, of a vector such that *Rn* is equal to *R_infinity / 2*. Again for very large vectors, there should be a result from the pipeline at every *tau* units of time. So, *n_1/2* represents the vector length needed to get a result at every *2*tau* units of time or every two clock cycles.
- *n_v*: The length, *n*, of a vector such that performing a vector operation on the *n* elements of that vector is more efficient than executing the *n* scalar operations instead.

Vector Computer Performance

Performance Characteristics	Year	Clock Cycle (nsec)	Peak Perf (Mflops)	R_infinity (x * y) (Mflops)	n_1/2 (x * y)
Cray-1	1976	12.5	160	22	18
CDC Cyber 205	1980	20.0	100	50	86
Cray X-MP	1983	9.5	70	53	
210 ... with 4 Procs	---	---	---	---	
840 Cray-2	1985	4.1	56	83	
488 ... with 4 Procs	---	---	---	---	
1951 IBM 3090	1985	18.5	54	high 20's	
108 ... with 8 Procs	---	---	---	---	
432 ETA 10	1986	10.5	---	---	
1250 ... with 8 Procs	---	---	---	---	
10,000 Alliant FS/8	1986	170.0	1	151	
6 ... with 8 Procs	---	---	47	1	23
Cray C90	1990	4.2	---	---	
952 ... with Procs	---	---	---	650	
			15,238	---	---
Convex C3880	---	---	960		

Performance Characteristics	Year	Clock Cycle (nsec)	Peak Perf (Mflops)	R_infinity (x * y) (Mflops)	n_1/2 (x * y)
Cray 3-128	1993	2.1	948	---	---
... with 4 Procs	---	---	3972	---	---

Table 1: Performance characteristics of vector processing computers using 64-bit floating-point numbers. The expression *(x * y)* refers to the element wise multiplication of two vectors, *x* and *y*

Table 1 provides some performance characteristics for some of the vector computers discussed later in this section. The values of *R_infinity* and *n_1/2* are for the elementwise multiplication of two vectors.

The pipeline vector computers can be divided into 2 architectural configurations according to where the operands are received in a vector processor. They are :

- Memory -to- memory Architecture, in which source operands, intermediate and final results are retrieved directly from the main memory.
- Register-to-register architecture, in which operands and results are retrieved indirectly from the main memory through the use of large number of vector or scalar registers.

Pipelined Vector Processing Methods

Vector computations are often involved in processing large arrays of data. By ordering successive computations in the array, the vector array processing can be classified into three types :

Horizontal Processing, in which vector computations are performed horizontally from left to right in row fashion.

Vertical processing, in which vector computations are carried out vertically from top to bottom in column fashion.

Vector looping, in which segmented vector loop computations are performed from left to right and top to bottom in a combined horizontal and vertical method.

A simple vector summation computation illustrate these vector processing methods

Let { ai for 1 <= i<= n) be n scalar contstants, Xj = (X1j,X2j…… Xmj)T for j = 1,2,3 ….n be n column vectors and Yj = (Y1j,Y2j…… Ym)T be a column vector of m components. The computation to be performed is

Y = ai.x1 + a2.x2 + …. an.xn

Y1 = Z11 + Z12 + …..Z1n

Y2 = Z21 + Z22 + …..Z2n

.

.

. Ym = Zm1 +Zm2+…..Zmn

Horizontal Vector Processing

In this method all components of the vector y are calculated in sequential order, yi for i = 1,2,….m. Each summation involving n-1 additions must be completed before switching to the evaluation the next summation.

Vertical Vector Processing :

The sequence of additions in this method are, compute the partial sum sequentially through the pipeline (in row wise z11+z12…)

Computer the partial sum in the column format repeatedly.

Vector Looping Method:

It combines the horizontal and vertical approaches into a block approach.

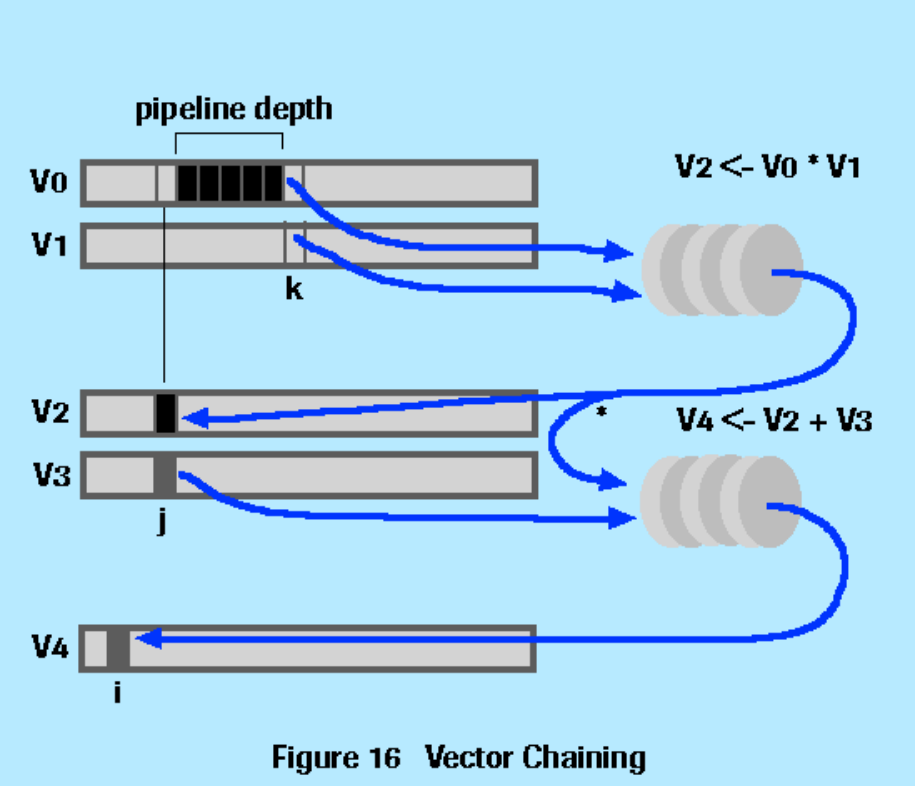


Figure 16 Vector Chaining

The Relative Vector/Scalar Performance and Amdahl Law

Let r be the vector/scalar speed ratio and f be the vectorization ratio. For example, if the time it takes to add a vector of 64 integers using the scalar unit is 10 times the time it takes to do it using the vector unit, then r = 10. Moreover, if the total number of operations in a program is 100 and only 10 of these are scalar (after vectorization), then f=90 (i.e. 90% of the work is done by the vector unit). It follows that the achievable speedup is:

Time without the vector unit

Time with the vector unit

For our example, assuming that it takes one unit of time to execute one scalar operation, this ratio will be:

100x1

--- = 100/19 (approx 5).

10x1 + 90x0.1

In general, the speedup is:

$$\frac{r}{(1-f)r + f}$$

So even if the performance of the vector unit is extremely high (r = oo) we get a speedup less than 1/(1-f), which suggests that the ratio f is crucial to performace since it poses a limit on the attainable speedup. This ratio depends on the efficiency of the compilation, etc... This also suggests that a scalar unit with a mediocre performance (even if coupled with the fastest vector unit), will yield mediocre speedup.

Strip-mining

If a vector to be processed has a length greater than that of the vector registers, then strip-mining is used, whereby the original vector is divided into equal size segments (equal to the size of the vector registers) and these segments are processed in sequence. The process of strip-mining is usually performed by the compiler but in some architectures (like the Fujitsu VP series) it could be done by the hardware.

Compound Vector Processing

A sequence of vector operation may be bundled into a "compound" vector function (CVF), which could be executed as one operation (without having to store intermediate results in register vectors, etc..) using a technique called chaining, which is an extension of bypassing (used in scalar pipelines). The purpose of "discovering" CVFs is to explore opportunities for concurrent processing of linked vector operations.

Notice that the number of available vector registers and functional units imposes limitations on how many CVFs can be executed simulataneously (e.g. Cray 1 CVP of SAXPY code leads to a speedup of 5/3. The X-MP results in a speadup of 5).

6.3 Vector memory

Interleaved memory banks

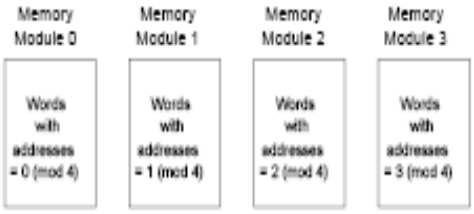
To allow faster access to vector elements stored in memory, the memory of a vector processor is often divided into *memory banks*. *Interleaved* memory banks associate successive memory addresses with successive banks cyclically; thus word 0 is stored in bank 0, word 1 is in bank 1, ..., word $n-1$ is in bank $n-1$, word n is in bank 0, word $n+1$ is in bank 1, ..., etc., where n is the number of memory banks. As with many other computer architectural features, n is usually a power of 2:

$n = 2^k$,

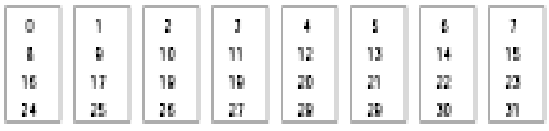
where $k = 1, 2, 3,$ or 4.

One memory access (load or store) of a data value in a memory bank takes several clock cycles to complete. Each memory bank allows only one data value to be read or stored in a single memory access, but more than one memory bank may be accessed at the same time. When the elements of a vector stored in an interleaved memory are read into a vector register, the reads are staggered across the memory banks so that one vector element is read from a bank per clock cycle. If one memory access takes n clock cycles, then n elements of a vector may be fetched at a cost of one memory access; this is n times faster than the same number of memory accesses to a single bank.

The figure below is an interleaved memory as it can be seen it places consecutive words of memory in different memory modules:



Since a read or write to one module can be started before a read/write to another module finishes, reads/writes can be overlapped. Only the leading bits of the address are used to determine the address within the module. The least-significant bits (in the diagram above, the two least-significant bits) determine the memory module. Thus, by loading a single address into the memory-address register (MAR) and saying “read” or “write”, the processor can read/write *M* words of memory. We say that memory is *M*-way interleaved. *Low-order interleaving* distributes the addresses so that consecutive addresses are located within consecutive modules. For example, for 8-way interleaving:



The Low end machine use the interleaved memory

• Memory banks take turns being connect to bus

• Interleaved memory access improves available bandwidth and may reduce latency for concurrent accesses.

High end machine use the multiple concurrent banks

• Might use crossbar switch (instead of bus, not instead of VDS) to connect several memory banks to the VDS simultaneously

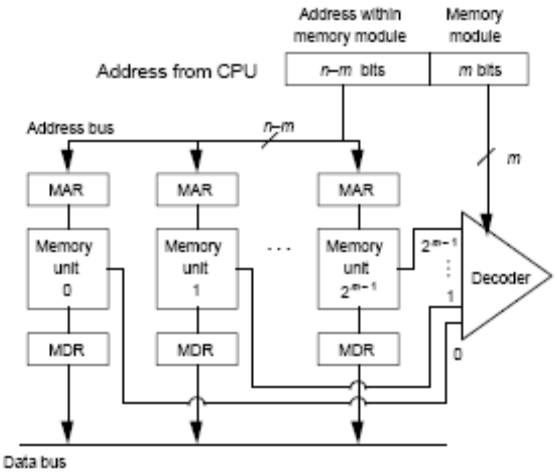
• Might be interleaved and assume different subsets of banks connected each clock

Interleaved-memory designs: Interleaved memory divides an address into two portions: one selects the module, and the other selects an address within the module.

Each module has a separate MAR and a separate MDR.

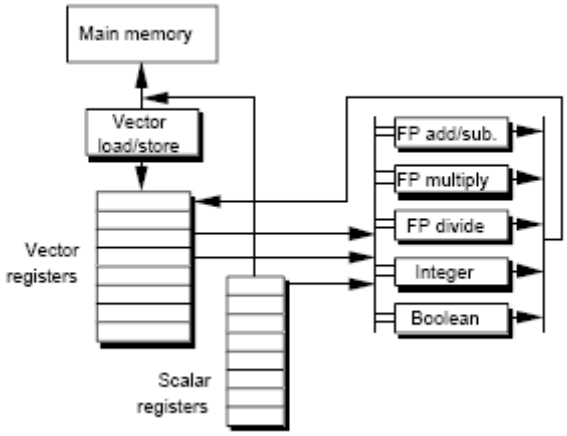
· When an address is presented, a decoder determines which MAR should be loaded with this address. It uses the low-order m — log2M bits to decide this.

· The high-order $n–m$ bits are actually loaded into the MAR. They select the proper location within the module.



An alternative to feeding a vector processor directly from external storage is to provide a hierarchical memory system similar to cache memory. Memory on the processor chip is called ***register storage*** rather than L1 cache, and is managed directly by the programmer rather than automatically by the hardware.

A vector processor with high-speed register storage:



The vector registers are large – 64 to 256 floating point numbers each. 256 floating point numbers at 64 bits each times 8 registers is equivalent to a 16k byte internal data cache.

6.3.1 Vector Memory Modeling

In vector processor when vector operate the parallel execution the memory access can be overlapped with vector execution the problem arise if the memory cannot keep up with vector execution rate.

Gamma (©) **ℜinomial model**

This model request is based on the principal to use vector request buffer to bypass waiting requests. An associated issue is the degree of bypassing or out-of-order requests that a source can make to the memory system. Suppose a conflict arises: a request is directed to a busy module. How many subsequent requests can the source make before it must wait? Assume each of *s* access ports to memory has a buffer of size *T BE / s* (Fig 7.19). This buffer holds requests (element addresses) to memory that are being held due to a conflict. For each source, the degree of bypassing is defined as the allowable number of requests waiting before stalling of subsequent requests occurs.

From a modeling point of view, this is different from the simple binomial or the δ-binomial models. The basis difference is that the queue awaiting service from a module is larger by an amount ϒ, where ϒ is the man queue size of bypassed requests awaiting service. Note that the average queue size (ϒ) is always less than or equal to the buffer size:

$$\gamma \leq TBF / s,$$

Since r cannot exceed the size of the physically implemented buffer. (Although, depending on the organization of the *TBF*, one source buffer could "borrow" from another)

With or without request bypassing, there ids a buffer between the *s* request sources and the *m* memory modules (Figure 7.19). This must be large enough to accommodate denied requests (no bypassing) i.e.:

Buffer = $TBF.\ mQ_c$

Where *Qc* is the expected number of denied requests per module, and *m* is the number of modules. The $m\ .\ Qc = n - B$, as discussed in chapter 6. If we allow bypassing, we will require additional buffer entries and additional control. Typically, an entry could include:

- Request source id.
- Request source tag (i.e., VR number).
- Module id.
- Address for request to a module
- Entry priority id (assuming more than one request can be bypassed).

While some optimization is possible, it is clear that large bypassed request buffers can be complex.

7.3.3 **Gamma(ɣ)-Binomial Model**

We now develop the ɣ -binomial model of bypassed vector memory behavior. Assume that each vector sources issues a request each cycle ($\delta = 1$), and that each physical requestor in the vector processor has the same buffer capacity and characteristic. If the vector processor can make *s* requests per cycle, and there are *t cycles per Tc*, we have:

Total requests per $Tc = t\ .\ s = n.$

This is the same as our *n* requests per *Tc* in the simple binominal model, but the situation in the vector processor is more complex. We assume that each of the sources s makes a request each cycle and *each of its* ɣ *-buffered requests* also makes a request.

Depending on the buffer control, these buffer requests are made only implicitly. The controller "knows" when a target module will be free and therefore schedules the actual request for that time. From a memory modeling point of view, this is equivalent to the buffer requesting service each cycle until the module is free.

Thus, we now have:

Total requests per $Tc = \underline{t \cdot s + t \cdot s \cdot \lambda}$

$= \underline{t \cdot s (1 + \lambda)}$

$= \underline{n(1 + \lambda)}$

Vector computation model not as compelling as it once was

• Multi-issue, latency-tolerant architectures reduce cost of loop overhead

– Instruction concurrency is available, and can substitute for data concurrency

• Improved compiler technology reduces value of programmer using vectors to give hints to hardware

– Improved algorithms to exploit cache

– Smart pre-fetching hardware, cache bypass, latency tolerance

• Commodity networked computing can often achieve comparable performance to a supercomputer

– Single-chip CPUs now have very high clock rates

– Improved infrastructure for parallel computing makes it accessible

But, desktop CPUs can benefit from supercomputer tricks

• Strided prefetching to reduce latency and better use memory bandwidth

• Selective bypassing of cache to avoid cache pollution

• Intel i860 was an experiment in this direction; but it was a poor compiler target

6.4 Multiple issue machines

The alternative to vector processors is multiple-issue machine. There are two broad classes of multiple-issue machines: statically scheduled and dynamically scheduled. In principle, these two classes are quite similar. Dependencies among groups of instructions are evaluated, and groups found to be independent are simultaneously dispatched to multiple execution units. For statically scheduled processors, this detection process is done by the compiler, and Instructions are assembled into *instruction packets*, which are decoded and executed at run time. For dynamically scheduled processors, the detection of independent instructions may also be done at compiler time and the code suitably arranged to optimize execution patterns, but the ultimate selection of instructions (to be executed or dispatched) is done by the hardware in the decoder at run time. In principle, the dynamically scheduled processor may have an instruction representation and form

that is indistinguishable from slower pipeline processors. Statically scheduled processors must have some additional information either implicitly or explicitly indicating instruction packet boundaries.

The extensive use of register ports provides simultaneous access to data as required by a VLIW processor. This suggests the register set as a processor bottleneck. Dynamic multiple-issue processors usually use multiple buses connecting the register set and functional units, and each bus services multiple functional units. This may limit the maximum degree of concurrency, but it can also significantly reduce the required number of register ports.

6.4.1 Very Long Instruction Words

Another approach to the parallelism problem is to exploit instruction level parallelism by having the compiler create bundles of instructions that take advantage of the chip's known functional units. For instance, if the processor is capable of executing 2 ALU operations, 1 load/store operation, and one multiply operation simultaneously, the compiler can do its best to arrange the instructions in such a way that groups consisting of all these elements will be formed. Together, the group will be issued as a very long instruction.

This technique is not as popular as superscalar because of the high dependency on compiler support, and the initial lack thereof. VLIW avoids the chip complexity issues that are present in superscalar, but it is hindered by the fact that if there is no compiler capable of efficiently created very long instructions, the architecture is basically useless. The VLIW technique is probably most useful in certain implementations of high-performance computers where the types of programs that will be executed are known in advance and that extensive compiler support is not needed.

VLIW Machines

As superscalar machines become more complex, the difficulties of scheduling instruction issue become more complex. The on-chip hardware devoted to resolving dependencies and deciding on instruction issue is growing as a proportion of the total. In some ways, the situation is reminiscent of the trend towards more complex CISC processors - eventually leading to the radical change to RISC machines.

Another way of looking at superscalar machines is as dynamic instruction schedulers - the hardware decides on the fly which instructions to execute in parallel, out of order, etc. An alternative approach would be to get the compiler to do it beforehand - that is, to *statically* schedule execution. This is the basic concept behind *Very Long Instruction Word*, or VLIW machines.

VLIW machines have, as you may guess, very long instruction words - in which a number of 'traditional' instructions can be packed. (Actually for more recent examples, this is arguably not really true but it's a convenient mental model for now.) For example, suppose we have a processor which has two integer operation units; a floating point unit; a load/store unit; and a branch unit. An 'instruction' for such a machine would consist of [up to] two integer operations, a floating point operation, a load or store, and a branch. It is the compilers responsibility to find the appropriate operations, and pack them together into a very long instruction - which the hardware can execute simultaneously without worrying about dependencies (because the compiler has already considered them).

Pros and Cons

VLIW has both advantages and disadvantages. The main advantage is the saving in hardware - the compiler now decides what can be executed in parallel, and the hardware just does it. There is no need to check for dependencies or decide on scheduling - the compiler has already resolved these issues. (Actually, as we shall see, this may not be entirely true either.) This means that much more hardware can be devoted to useful computation, bigger on-chip caches etc., meaning faster processors.

Not surprisingly, there are also disadvantages.

- **Compilers.** First, obviously compilers will be harder to build. In fact, to get the best out of current, dynamically scheduled superscalar processors it is necessary for compilers to do a fair bit of code rearranging to 'second guess' the hardware, so this technology is already developing. It is observed that building good compilers for VLIW is non-trival.
- **Code Bigger.** Secondly, programs will get bigger. If there are not enough instructions that can be done in parallel to fill all the available slots in an instruction (which will be the case most of the time). There will consequently be empty slots in instructions. It is likely that the majority of instructions, in typical

applications, will have empty code slots, meaning wasted space and bigger code. (It may well be the case that to ensure that all scheduling problems are resolved at compiler time, we will need to put in some *completely empty* instructions.) Memory and disk space is cheap - however, memory bandwidth is not. Even with the large and efficient caches, we would prefer not to have to fetch large, half-empty instructions.

- **One Stalls, all Stall.** Unfortunately, it is not possible at compile time to identify all possible sources of pipeline stalls and their durations. For example, suppose a memory access causes a *cache miss*, leading to a longer than expected stall. If other, parallel, functional units are allowed to continue operating, sources of data dependency may *dynamically* emerge. For example, consider two operations which have an output dependency. The original scheduling by the compiler would ensure that there is no consequent WAW hazard. However, if one stalls and the other 'runs ahead', the dependency may turn into a WAW hazard. In order to get the compiler to do *all* dependency resolution, it is required to stall *all* pipeline elements together. This is another performance problem.
- **Hardware Shows Through** A significant issue is the break in the barrier between architecture and implementation which has existed since the IBM 360 in the early/mid 60s. It will be necessary for compilers to know exactly what the capabilities of the processor are - for example, how many functional units are there?
- VLIW instruction sets are not backward compatible between implementations. As wider implementations (more execution units) are built, the instruction set for the wider machines is not backward compatible with older, narrower implementations.
- Load responses from a memory hierarchy which includes CPU caches and DRAM do not give a deterministic delay of when the load response returns to the processor. This makes static scheduling of load instructions by the compiler very difficult.

6.4.2 Moving beyond VLIW

EPIC architectures add several features to get around the deficiencies of VLIW:

- Each group of multiple software instructions is called a *bundle*. Each of the bundles has information indicating if this set of operations is depended upon by the subsequent bundle. With this capability, future implementations can be built to issue multiple bundles in parallel. The dependency information is calculated by the compiler, so the hardware does not have to perform operand dependency checking.
- A *speculative* load instruction is used as a type of data prefetch. This prefetch increases the chances for a primary cache hit for normal loads.
- A check load instruction also aids speculative loads by checking that a load was not dependent on a previous store.

The *EPIC* architecture also includes a *grab-bag* of architectural concepts to increase *ILP*:

- Predicated execution is used to decrease the occurrence of branches and to increase the speculative execution of instructions. In this feature, branch conditions are converted to predicate registers which are used to kill results of executed instructions from the side of the branch which is not taken.
- Delayed exceptions (using a Not-A-Thing bit within the general purpose registers) also allow more speculative execution past possible exceptions.
- Very large architectural register files avoid the need for register renaming.
- Multi-way branch instructions

The IA-64 architecture also added register_rotation - a digital signal processing concept useful for loop unrolling and software pipelining.

6.5 Summary

Vector supercomputers are not viable due to cost reason, but vector instruction set architecture is still useful. Vector supercomputers are adapting commodity technology like SMT to improve their price-performance. Superscalar microprocessor designs have begun to absorb some of the techniques made popular in earlier vector computer systems (Ex - Intel MMX extension). Vector processors are useful for embedded and multimedia applications which require low power, small code size and high performance.

Vector Processor vs Multiple Issue processor

Advantage of Vector Processor

— good Sp on large scientific problems

— mature compiler technology.

Disadvantage of **Vector Processor**

— limited to regular data and control structures

— Vector Registers and buffers

— memory BW

Advantage of multiple issue processor

— general-purpose

— good Sp on small problems

— developing compiler technology

Advantage of multiple issue processor

— instruction decoder H/W

— large D cache

— inefficient use of multiple ALUs

6.6 Keywords

vector an ordered list of items in a computer's memory. A simple vector is defined as having a starting address, a length, and a stride. An indirect address vector is defined as having a relative base address and a vector of values to be applied as indices to the base.

vector processor A computer designed to apply arithmetic operations to long vectors or arrays. Most vector processors rely heavily on *pipelining* to achieve high performance.

vector register a storage device that acts as an intermediate memory between a computer's functional units and main memory

interleaved memory memory divide into a number of modules or banks that can be accessed simultaneously.

VLIW Very Long Instruction Word; the use of extremely long instructions (256 bits or more) in a computer to improve its ability to chain operations together.

6.7 Self assessment Question

1. What are vector?
2. Why vector processors popular in scientific calculations
3. Drawback of vector processor
4. Drawback of VILW processor
5. Write problems in implementing VILW processor?

6.8 Reference:

Advance computer architecture by Kai Hwang

Computer Architecture by Michael J. Flynn