

Lecture #12: Vector Architectures

*Professor: S. Lennart Johnsson**TA: Wei Ding*

1 The Generic Computer

The generic computer consists of a central processing unit, CPU, and memory, as illustrated in Figure 1. The memory contains both instructions and data. Real computers are refinements of this generic computer. The refinements depend upon the technology in which the various parts of the system are implemented and the quest for cost efficiency and speed. Supercomputers have always used the idea of pipelining to achieve high speed. Parallelism is another technique to achieve high speed. Pipelining is often both less expensive and simpler to manage than parallelism. However, in computer systems the speed improvement that can be achieved due to pipelining is typically limited to within a factor of five to ten. Parallelism can be used to achieve a speed improvement by a factor of a thousand or more. State-of-the-art supercomputers use both pipelining and parallel processing. Pipelining functional units such as adders and multipliers has lead to the notions of *vector architectures* and *vectorizing compilers*. In these notes we will focus on the idea of pipelining and the scheduling of operations for a few typical computations on vector architectures, i.e., architectures with pipelined functional units.

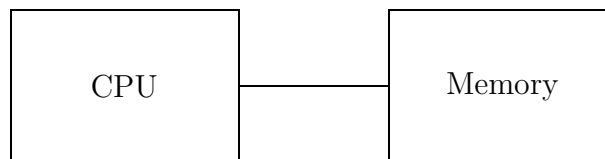


Figure 1: The generic computer

A typical instruction sequence for the generic computer is:

1. Fetch an instruction
2. Decode the instruction
3. Compute operand addresses
4. Fetch operands
5. Perform the operation
6. Store the result

7. Increment the program counter

If each step of this sequence could be performed in one clock cycle, then a typical operation would finish in exactly seven cycles. With a clock frequency of 7 GHz a performance of 1 billion operations per second is possible if all operands that are required can be moved as required in one clock cycle. For instance, in the operation $z = x \times y$, both x and y must be fetched on step four for the above performance estimate to be true. A 7 GHz clock frequency is equivalent to a cycle time of $\frac{1}{7}$ nanoseconds. In one nanosecond light travels about 0.3 m. Thus, at a clock frequency of 7 GHz, a signal can travel at most about 4 cm within a clock period. It is not feasible to fit a CPU and all of memory for large scale problems within a volume determined by a maximum distance of 4 cm between any parts.

Conversely, if the supercomputer with its memory would fit in a volume with a maximum distance of 3 m, then the clock frequency, assuming that each of the seven operations complete within one clock cycle, must be less than 50 MHz. It takes a signal at least 10 nsec to travel 6 m, which may be required for the fetch operation. It requires that at least one address be sent to memory and the content of that memory address be returned to the CPU. Hence, the clock period must be at least 20 nsec. The maximum speed of a computer designed in this fashion is $\frac{50}{7} \approx 7$ million operations/second.

The simplistic instruction execution described above has several problems. One problem is that the different elements of an instruction often requires a different amount of time. In todays computers, the memory is often the slowest part. Moreover, the time also depends upon where in memory an operand reside, as we will see later. Another problem is the poor utilization of some units. For instance, the instruction decoder is only used one out of every seven cycles. Memory, the weakest link in most current architectures, is used three out of seven cycles. If each memory operation required four cycles instead of one, then the instruction time would be 16 cycles and the decoder would be used one out of every 16 cycles whereas the memory would be used 12 out of 16 cycles.

Pipelining can improve both the instruction execution rate as well as the utilization of the functional units.

2 Pipelining

Pipelining is a common technique to increase performance in many systems. Most cafeterias are good examples thereof. Typically there are stations for salads, entres, beverages, and checkout. And, as we know, it is not necessary for a person to clear all stations before the next person can enter the first station. This assembly line for meals is based on the same idea as assembly lines for automobiles, except it is considerably shorter. The essential idea is to increase the capacity of the facility by taking advantage of the fact that many operations are localized in space. The pace at which meals can be served (units produced) is determined by the task that takes the longest time, rather than by the time to assemble your entire meal (or assemble a complete car).

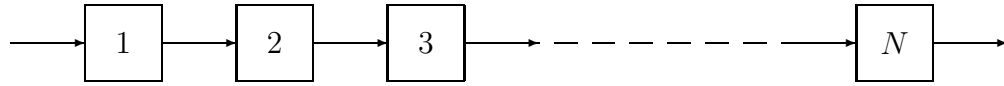


Figure 2: The nonpipelined operation of an N station assembly line.

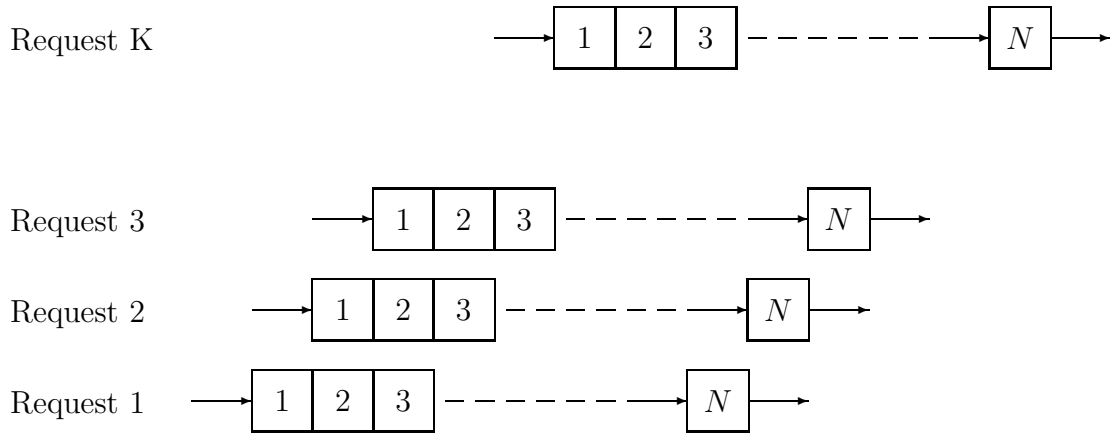


Figure 3: The pipelined operation of an N station assembly line.

Another example of a pipeline is a one-way highway with only one lane. A very conservative approach is to reserve the highway for a single driver, not allowing anybody else on it until the driver on the highway exits from it. A better way to operate the highway is clearly to let drivers onto the highway with some safe time interval separating different drivers. With drivers traveling at moderate speed and the distance between entrance and exit points being far apart, the capacity of the highway is clearly improved considerably by the second management strategy. It relies on the fact that each driver only occupies a small portion of the highway at any given time, and that each driver always travels in one direction. The time each driver spends between entrance and exit points is unaffected by the management policies.

In our cafeteria example with four stations, the stations can be designed and managed so the time at each station takes about the same time. It is clear that the time for each person to assemble the meal by passing all stations is the same whether or not there are persons at the other stations. On the other hand, the capacity of the cafeteria is increased fourfold by allowing a new person to enter a station as soon as the person currently at the station has been served. Figures 2 and 3 illustrate the difference between a nonpipelined and a pipelined operation.

We now define some concepts that are useful in analyzing pipelined systems:

- *Latency* is the time elapsed between the initiation and the completion of an operation. Latency may be measured in absolute time or in clock cycles. In our cafeteria example the latency is four time units (clock cycles).
- *Pipeline depth* is the number of different stages or stations in the pipeline. In our cafeteria

example, the pipeline depth is the same as the latency. However, if the ready made salad station was replaced by a salad bar, the pipeline depth would still be four while the latency would increase. Making your own salad requires more time than selecting a ready made one. If we assume that making your own salad takes four times as long as selecting and picking up a dish, then the latency is seven time units with a salad bar station.

- The *initiation rate* is the rate at which requests can be served measured as requests per unit time. For pipelines in which all stations takes the same amount of time, the initiation rate is one operation per unit time. In general, the initiation rate is determined by the first stage in the pipeline. In our example, the initiation rate is one request in every four time units with the salad bar, one request per time unit in the case of ready made salads. (The salad station was first in our example.)
- The *completion rate* is the rate at which requests can be finished measured in requests per unit time. The completion rate is determined by the last stage of the pipeline. In our example, the completion rate is unaffected by whether the salad station is designed for ready made salads or a salad bar.
- *Throughput* refers to the steady state of a pipeline. It is the rate at which operations can be completed, measured in requests per unit time. The throughput is determined by the slowest stage in the pipeline. Thus, the throughput in our example is either one request per unit time, or one request per four units of time depending upon whether there is no salad bar or there is one. The throughput need not be equal to either the initiation rate or the completion rate. For instance, if we add a station at the beginning of the cafeteria line in which a person picks up a tray, knives, forks, etc., and view it as a unit time operation, then both the initiation and completion rate will be one request per unit time regardless of how the salad station is designed. But, the throughput still depends on the design of the salad station.
- *Startup time* is the time required between the initiation of a pipeline operation and the time the first result is delivered. The startup time is equal to the latency. The two notions are often used interchangeably. However, startup typically refers to the pipeline itself, whereas latency refers to the entities passing through the pipeline. For a sequence of requests to a pipeline, all stages of a pipeline are busy after the startup time has passed, assuming that the number of requests exceeds the latency. However, whether a request is the first one or is being serviced after all stations in the pipeline are busy, it experiences the same latency.

The greater the pipeline depth, the greater is the gain in throughput from using a pipeline strategy. For instance, in the highway example, assume that the distance between entrance and exit is 15 km, and that the average speed along the highway is 75 km/hr. The latency is 12 minutes ($\frac{15}{75} \times 60$ minutes). Thus, the throughput without pipelining is five cars per hour. With a safety zone of 50 m for each car, cars can enter the highway at a rate of one car every 2.4 seconds ($\frac{50}{75000} \times 3600$). The pipeline depth is $\frac{15000}{50} = 300$. The throughput is $\frac{3600}{2.4} = 1500$. The capacity of the highway is increased 300 fold.

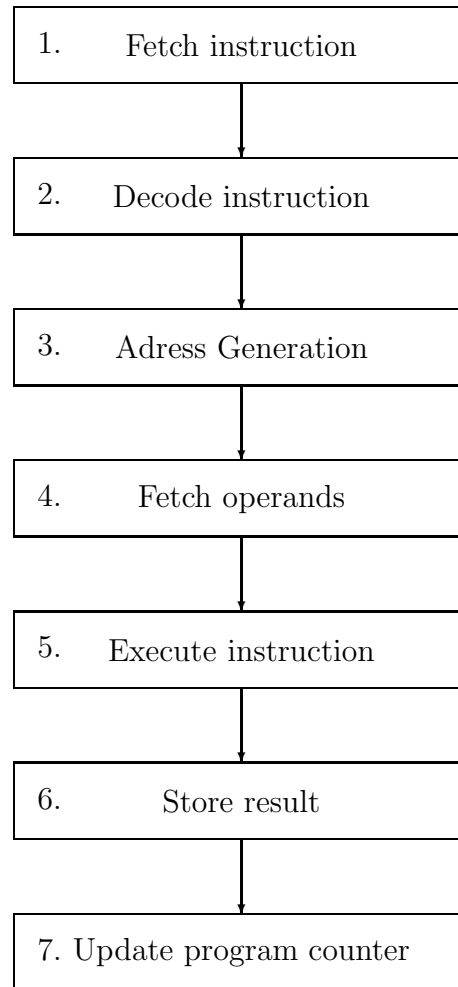


Figure 4: Elemental operations of an instruction.

In computers, the pipeline depth is often in the range 2 – 20. We will now discuss some of the issues related to pipelined architectures, in particular instruction and arithmetic pipelines.

A pipelined execution of our model instruction, as shown in Figure 4, would yield an increase in throughput by a factor of seven, assuming all operations requires the same time.

If each memory operation requires four cycles instead of one, then the latency becomes 16 cycles and the throughput is one operation every four cycles, at best. Figure 5 shows the actual operation of a pipeline in which each memory operation requires four cycles and new instructions are issued every four cycles. The width of each box is proportional to the time required to perform the stage indicated by the box number.

As Figure 5 shows, in order for the pipelining to work properly, the memory system must be capable of handling three memory requests concurrently from cycle 12 and on. If the memory system can only handle a single request at a time, then, in fact, the throughput is limited to one instruction every 15 cycles, which is an insignificant improvement over the nonpipelined

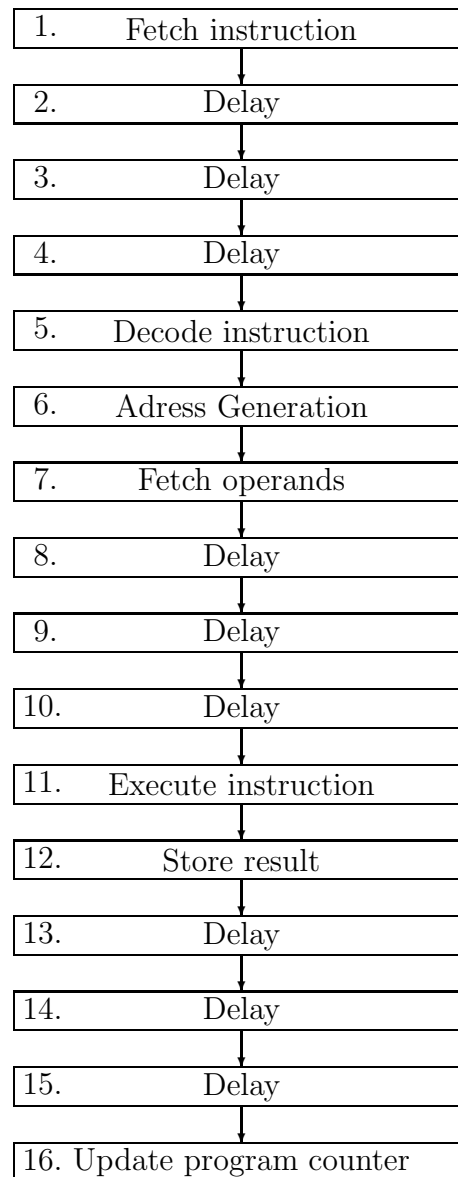


Figure 6: Partitioning slow operations into unit time operations.

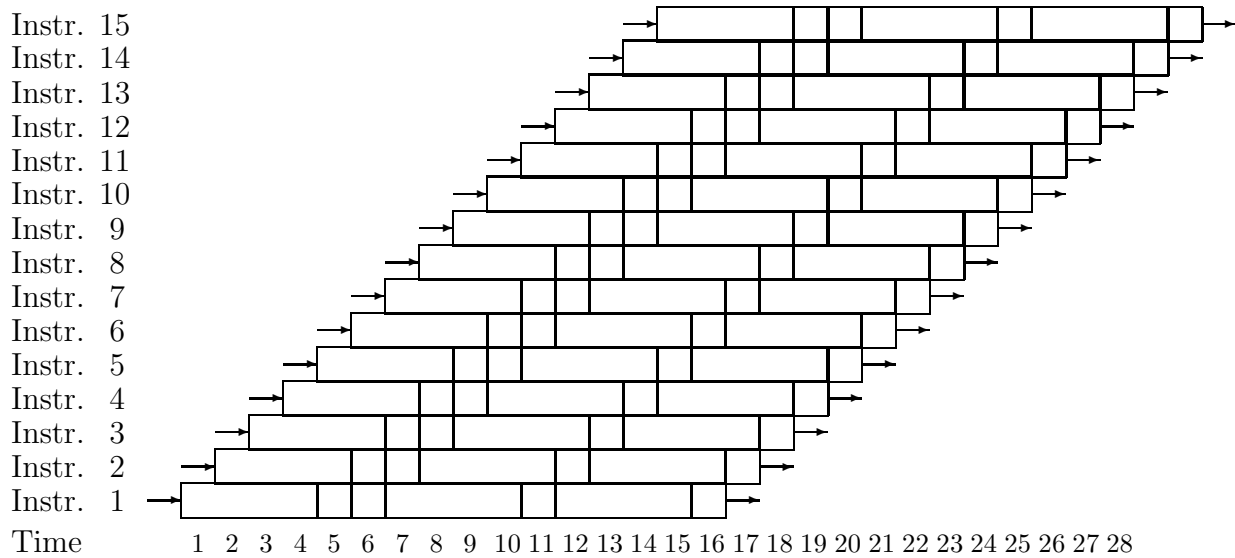
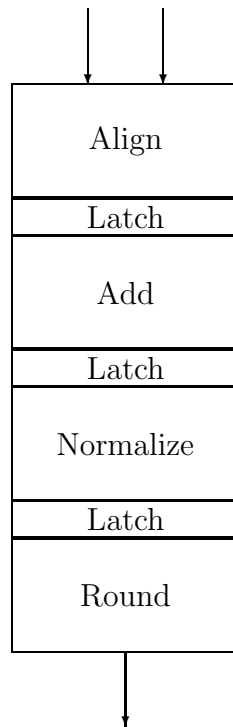


Figure 7: Pipelining of the partitioned instructions.



Let the latency for each stage be L_i time units for $i = 1, 2, 3, 4$. The total latency is $L = \sum_{i=1}^4 L_i$. The latches in the above diagram separate the stages from each other, making sure that data remains separated in space. Assuming that the latches do not introduce any delays, an adder without latches would have a throughput of one addition every L time units, while the unit with latches has a throughput of one addition every $\max_i L_i$ time units. Most computers are designed as synchronous machines, with the latency in each stage being typically the same for all stages.

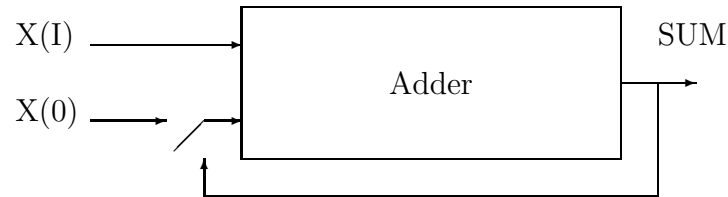


Figure 8: Addition through iteration.

For the adder above, the throughput would be four times higher with the introduction of three latches, assuming that the latency in all stages is the same.

2.1 Scheduling of Computations for Efficient Use of Pipelines

We will now study the scheduling of operations for efficient use of a pipelined adder employing an example: the summation of the elements of an array, $y = \sum_{i=0}^{N-1} x(i)$. We will refer to the different schedules as different algorithms. Of the three algorithms we are considering, all require $N - 1$ arithmetic operations.

Algorithm 1: A Simple Iterative Approach

This algorithm performs a simple iterative loop to compute the sum.

```
SUM=X(0)
FOR I=1 TO N-1 DO
    SUM=SUM+X(I)
ENDFOR
```

The sequence of operations implied by this code is depicted in Figure 8.

The time to carry out the operation for a latency of L clock cycles is $L \times (N - 1)$. The sequence of operations implied by the loop in Algorithm 1 prevents the capacity of the pipeline from being exploited. The result of one operation is required as an input for the next operation. Such a dependence is sometimes referred to as a *Read After Write* (RAW) dependence or *true data dependence*.

The schedule of operations implied by Algorithm 1 is not good for pipelined architectures. But, fortunately there exist many other schedules that compute the sum for associative operations correctly. For non-associative operations, such as floating-point addition, caution must be exercised in changing the summation order.

Algorithm 2: A Recursive Approach

This algorithm performs a recursive, pairwise addition of elements. In the first phase of the algorithm, element $X(I)$ and $X(I + \frac{N}{2})$ are added for $I = 0, 1, \dots, \frac{N}{2} - 1$, assuming N is even. Assume that the result is stored in $X(I)$, $I = 0, 1, 2, \dots, \frac{N}{2} - 1$. Then, in the second phase, elements $X(I)$ and $X(I + \frac{N}{4})$ are added, the assumption being that N is divisible by 4, etc. A sample algorithm for the case $N = 2^n$ is given below.

```

FOR I=1 TO  $n$  DO
    FOR J=0 TO  $N/2^I - 1$  DO
         $X(J)=X(J)+X(J+N/2^I)$ 
    ENDFOR
ENDFOR

```

Note that the vector X is used as a working array. The contents of the first half of the vector are altered. If that is unacceptable, temporary storage equal to half the vector length may be required.

Note also that for arbitrary N the loop bounds and indexing become more complex to guarantee the correct result.

The time for the case $N = 2^n$ is easily derived. Exploiting pipelining, the first execution of the loop on I requires $L + N/2 - 1$ clock cycles. The second execution requires $L + N/4 - 1$ clock cycles, etc. The total time is $N - 1 + (L - 1) \times \log_2 N$ cycles, assuming no overlap between operations in different iterations of the loop.

Algorithm 3: Partial Sums

Algorithm 2 required a large working storage (half the size of the input array). The working storage for Algorithm 1 was only one variable passed from the adder output to its input. The algorithm described next combines the ideas in Algorithms 1 and 2 to yield an algorithm that efficiently utilizes the pipeline, yet has very limited need for temporary storage. Algorithm 3 uses L partial sums that are added together to produce the final result:

```

FOR I=0 TO L-1 DO
     $SUM(I)=X(I)$ 
ENDFOR
FORALL J=0 TO L-1 DO
    FOR I=L TO N-1 STEP L DO
         $SUM(J)=SUM(J)+X(I+J)$ 
    ENDFOR
ENDFORALL

```

In Figure 9, the algorithm is shown for $L = 4$ and $N = 16$. The idea is that the next element of each partial sum arrives at the adder L steps later, just when the previous value has been accumulated (assuming the adder has latency L).

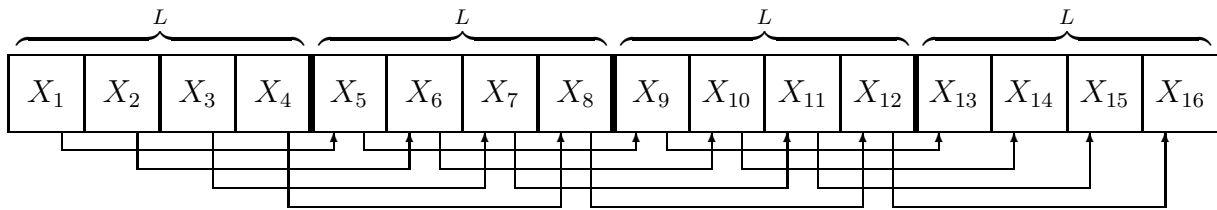


Figure 9: Addition using partial sums for $L = 4$ and $N = 16$.

At this point there are L partial sums that must be added. These partial sums can be added in $L - 1 + (L - 1) \lceil \log_2 L \rceil$ cycles using Algorithm 2, or in $L(L - 1)$ cycles using Algorithm 1.

The time required for this algorithm is $N + L$ clock cycles for the first part of the algorithm, and $(L - 1)(\lceil \log_2 L \rceil + 1)$ clock cycles for the second part using Algorithm 2. Total time is $N + 2L - 1 + (L - 1) \lceil \log_2 L \rceil$ clock cycles.

Comparison of Algorithms 1, 2, and 3

Alg.	Arith. Ops.	Cycles	Load	Store	Scalar Ops.	Vector Ops.	Vector Length
1	$N - 1$	$(N - 1) \times L$	N	1	N	0	0
2	$N - 1$	$N - 1 + (L - 1) \times \lceil \log_2 N \rceil$	$2(N - 1)$	$N - 1$	0	$\lceil \log_2 N \rceil$	$N/2, N/4, \dots$
3	$N - 1$	$N + 2L - 1 + (L - 1) \times \lceil \log_2 L \rceil$	N	1	0	$\lceil \log_2 L \rceil + 1$	$N - L, L, L/2, \dots$

Algorithm 3 both requires the fewest cycles and the least working storage.

Algorithm 3 has been used in the implementation of inner products on production supercomputers (the Cray series of computers and the Connection Machine systems CM-2/200).

In memory-to-memory architectures, operands are fetched from memory and stored in memory for each instruction. Algorithm 2 is often recommended for such architectures. However, few architectures today are memory-to-memory architectures.

In summary, pipelined units may require reordering of the specified operations to make efficient use of the pipeline. The optimal reordering depends upon several factors, such as the number of possible concurrent memory accesses and overhead in the issuance of scalar and vector instructions.

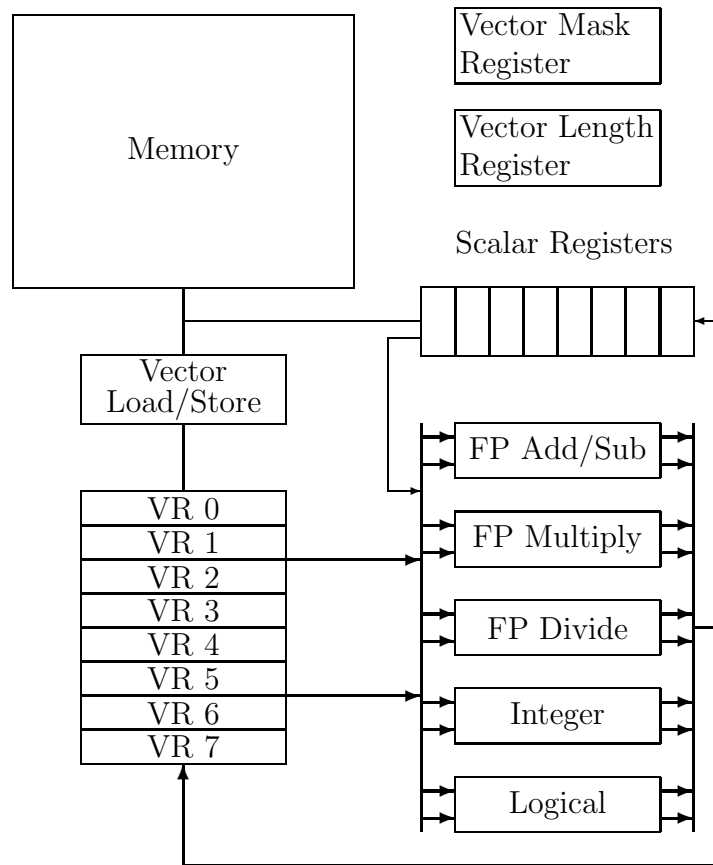


Figure 10: A generic vector-register architecture.

3 Vector Architectures

3.1 A Generic Vector Processor

In this section we define a generic *vector-register processor*. In such a processor all operations are between registers and functional units. Load/store operations are between registers and main memory. Arithmetic and logic operations are between registers and the corresponding functional units. Figure 10 shows a generic vector-register architecture.

In *vector-memory architectures*, the operations are between functional units and memory. In the following section, we will focus on vector-register processors.

The pipeline depths of the functional units vary significantly. The depth of the adder and the multiplier are often approximately the same. The depth of logical units is often similar. However, division is an inherently more difficult operation in a binary number system; the pipeline depth is often significantly higher. Some processors do not have pipelined units for division. Division in such processors has a significantly lower throughput than other operations.

Each *vector-register* contains several operands. The contents of a vector-register is referred to

as a vector. We refer to the maximum number of elements in a vector-register as the *maximum vector-length* or MVL. Each vector-register has two read-ports and one write-port. In our generic vector-register architecture, there are eight vector-registers with MVL=64.

In many vector processor architectures, the functional units are shared between the *scalar unit* and the *vector unit*. However, the scalar unit has its own *scalar-registers*. The operands for a functional unit can be taken from (and returned to) either the vector-registers for vector-instructions or to the scalar-registers for scalar-instructions. Our generic vector-register machine has eight scalar-registers.

The *load/store* unit is often rather complex, since it has to manage the sometimes large difference in speed between the main memory and the processor with its registers. Load/store units are fully pipelined on all vector processor architectures. The pipeline depth for the load/store pipe in traditional supercomputers is often large, mostly due to the difference in speed between main memory and the processor. The range for the Cray series of computers has been 14 – 50 depending upon the model.

The *Vector-Length* register stores the lengths of the vectors currently residing in the vector registers. We use VL to denote the current length of the vectors in the registers.

The *Vector-Mask* register contains a vector of logicals (bits), one for each vector element in the vector-registers. These Boolean variables are used to decide whether or not an operation shall be performed on the corresponding element of the vector.

In addition to the facilities shown in Figure 10, a vector-register processor has a set of general registers and units for address calculation plus other standard operations.

Most vector architectures have an overhead associated with the initiation and completion of a vector operation in addition to the pipeline depth of the functional unit. Hence, if a vector-instruction depends upon the completion of another vector-instruction, then it must wait until the instruction is fully complete. In our generic vector-register processor, a 4-cycle delay is incurred. We refer to this delay as the *vector-stall* time.

The *vector-loop* startup penalty accounts for the overhead in setting up each vector operation. Vectors of arbitrary length have to be divided into chunks of a length at most equal to the length of the vector-registers. The technique used to partition a vector into chunks of a length equal to the length of the vector-registers, and a remainder, is known as *strip-mining*. Strip-mining will be discussed later. The vector-loop penalty accounts for setting up vector starting addresses, setting up vector strides, incrementing counters, and executing a loop branch.

The *vector-base* penalty accounts for the overhead of the whole vector operation. This overhead consists of computing starting addresses and setting up the vector control. All of these operations are scalar.

Both the vector-loop and vector-base overheads depend upon hardware, compilers, and the actual vector-instructions to be executed. However, the dependence is small. For simplicity we will use the same values for all vector-instructions. The values are similar to those that applied to the Cray-1 [1].

Figure 10 does not show the full capability of most vector architectures. In general, there

General Characteristics	Vector Operation Costs	
	Operation	Depth
8 Vector Registers (MVL=64)	Vector Add	6
8 Scalar Registers	Vector Multiply	7
1 Load/Store Path (VP-1)	Vector Divide	20
2 Load Paths/1 Store Path (VP-3)	Vector Load	12
	Vector Store	12
	Vector Copy ²	1
	Vector Stall	4
	Vector Loop	15
	Vector Base	10

Table 1: Characteristics of the generic vector architectures VP-1 and VP-3.

are a sufficient number of data paths between the functional units and the registers such that all functional units can operate concurrently, given a sufficient number of output registers. Different vector-registers can supply different functional units with data without conflict in our vector-register architecture. A register can even serve as input to several functional units. However, a vector-register can be the destination of only one functional unit. A register can concurrently be both the source and the destination of vector operations.

We will now define our generic vector-register machines, VP-1 and VP-3. The two machines are identical except for their load/store characteristics. VP-1 has 1 load/store path. VP-3 has 2 load paths and 1 store path¹. Table 1 summarizes the pipeline depths and other overhead associated with vector operations on VP-1 and VP-3.

3.2 Performance Model

We can now define an expression for the execution time of a vector operation:

$$T_N = T_{base} + \left\lceil \frac{N}{MVL} \right\rceil \times (T_{loop} + T_{start}) + N \times T_{elem}.$$

MVL is the maximum vector length, and T_{start} is one of, or a sum of several of, the functional unit pipeline depths in Table 1. Since we assume that all units in our generic vector-register processor are fully pipelined, each vector operation contributes one unit to T_{elem} .

T_{base} and T_{loop} are the overhead costs from Table 1. Vector stalls, if any, are added into T_{start} .

We will now work through some examples using Algorithm 2 to sum the elements of an array.

¹Each path can be viewed as an independent functional unit. Since VP-1 has only 1 path, it cannot perform a load and store simultaneously. VP-3, however, can perform up to 2 loads and 1 store simultaneously.

²In our generic architectures, the task of copying all or part of one vector register to another vector register will be viewed as a functional unit of depth 1. For example, copying the last 32 positions of vector register V0 to the first 32 positions of V1 would require $1 + 32$ cycles.

T_{loop}	Load	Load	Stall	Add	Stall	Store
15	$12 + VL$	$12 + VL$	4	$6 + VL$	4	$12 + VL$

Figure 11: Total cycles used in one segment of pairwise addition on VP-1.

Example 1: Computing $y = \sum_{i=0}^{N-1} x(i)$ with Algorithm 2 on VP-1

The algorithm has a loop of $\log_2 N$ iterations, each of which performs an addition of two subarrays. The subarrays must be partitioned into segments that fit in the registers.

On VP-1, the operations on each such segment are:

Load a segment of X
 Load a second segment of X
 Perform a vector add on the two segments
 Store the result

With a vector length of VL, each vector operation requires the following time:

Load first segment	$12+VL$	
Load second segment	$12+VL$	
Vector add	$6+VL$	
Stall for vector add	4	(vector add depends on second load)
Vector store	$12+VL$	
Stall for vector store	4	(vector store depends on vector add)
Total	$\frac{50+4VL}{}$	

Figure 11 shows a single segment of this operation. In general, $VL = MVL$ for all but the first segment when $VL = N \bmod MVL$.

Hence, we find that $T_{\text{start}} = 50$ and that $T_{\text{elem}} = 4$. Hence, iteration I in the outer loop requires time:

$$T_{N/2^I} = 10 + \left\lceil \frac{N/2^I}{MVL} \right\rceil (15 + 50) + 4N/2^I.$$

The total time is:

$$\begin{aligned} T_N &= \sum_{I=1}^{\log_2 N} \left(10 + \left\lceil \frac{N/2^I}{MVL} \right\rceil (15 + 50) + 4N/2^I \right) \\ &= 10 \log_2 N + 65 \sum_{I=1}^{\log_2 N} \left\lceil \frac{N/2^I}{MVL} \right\rceil + 4(N - 1). \end{aligned}$$

Since MVL is 64, this makes the asymptotic rate approximately 0.2 operations per clock cycle. For this computation, the number of operations is equal to the number of elements. Note that the latencies have a significant impact on the asymptotic performance. The performance would be 0.25 operations per clock cycle without the 65 cycles attributed to T_{loop} and T_{start} .

First Segment	Loop	Load	Stall	Add
	15	$12 + (N \bmod MVL)$	4	$6 + (N \bmod MVL)$
Middle Segments	Loop	Load/Store	Stall	Add
	15	$12 + MVL$	4	$6 + MVL$
Final Segment	Loop	Store		
	15	$12 + MVL$		

Figure 12: Total cycles used in segments of pairwise addition on VP-3.

Example 2: Computing $y = \sum_{i=0}^{N-1} x(i)$ with Algorithm 2 on VP-3

On VP-3, the increased bandwidth to memory can be used to our advantage. For a given subarray, the segments can be manipulated as follows:

Load first two segments of X
Perform a vector addition on the two segments
<hr/>
Load next two segments of X, store previous result
Perform a vector addition on the new segments
<hr/>
Load next two segments of X, store previous result
Perform a vector addition on the new segments
<hr/>
⋮
<hr/>
Store final result

Note that no store is performed in the first iteration and no add or load is performed in the last iteration.

With a vector length of VL, each vector operation (other than the first and last) requires the following time:

Vector load, store	$12 + VL$
Vector add	$6 + VL$
Stall for vector add	4 (<i>vector add must wait for vector loads</i>)
Total	$22 + 2VL$

Because we have overlapped the $(I - 1)^{th}$ store with the I^{th} loads, the number of cycles used by the outer loop is more complicated. Figure 12 shows the three cases when $N > MVL$ and $(N \bmod MVL) \neq 0$. Note that all segments except for the first have $VL = MVL$.

The number of cycles required by iteration I in the outer loop is a bit harder to calculate because of the final store at the end. We also must allow or cases when $N < MVL$, which

affects the last segment. The time is:

$$T_{N/2^I} = 10 + \underbrace{\left\lceil \frac{N/2^I}{MVL} \right\rceil (15 + 22)}_{\text{First and Middle Segments}} + \underbrace{(15 + 12 + \min(MVL, N/2^I))}_{\text{Final Store}} + 2N/2^I.$$

The total time is:

$$\begin{aligned} T_N &= \sum_{I=1}^{\log_2 N} \left(10 + \left\lceil \frac{N/2^I}{MVL} \right\rceil (15 + 22) + (15 + 12 + \min(MVL, N/2^I)) + 2N/2^I \right) \\ &= 37 \log_2 N + 37 \sum_{I=1}^{\log_2 N} \left\lceil \frac{N/2^I}{MVL} \right\rceil + \sum_{I=1}^{\log_2 N} (\min(MVL, N/2^I)) + 2(N - 1). \end{aligned}$$

The exact expression for T_N is complicated enough that we can't expand it out. However, we can still easily determine the asymptotic rate by extracting out the terms which depend on N . The effect of other terms on the asymptotic rate diminishes as N gets large. Since MVL is 64, the asymptotic rate is nearly 0.4 operations per cycle. The asymptotic performance is almost twice that of VP-1. Again, note that the latencies have a significant impact on the asymptotic rate. The performance would be 0.5 operations per clock cycle without the 37 cycles attributed to T_{loop} and T_{start} .

The performance is clearly dependent upon the array length. Table 2 summarizes the number of operations per cycle for some different array sizes. Note that the performance drops slightly for values of N which are just slightly bigger than a multiple of MVL.

Since the time per operation depends upon the maximum vector length for the architecture, two general measures are often used to describe performance. These are r_∞ and $N_{1/2}$. They are defined as follows:

- r_∞ is the *asymptotic performance rate* (i.e., $\lim_{N \rightarrow \infty} \frac{\text{ops}}{\text{cycle}}$).
- $N_{1/2}$ is the *vector length* (i.e., number of elements) for which half of the asymptotic performance is attained, often called *half vector length*.

Note that each of these measures depends upon both the architecture and the operation being performed.

For this example, the values of r_∞ are 0.199 for VP-1 and 0.388 ops/cycle for VP-3, and the values of $N_{1/2}$ are 87 and 311 elements, for VP-1 and VP-3 respectively. A quick review of the data in Table 2 suggests that these values are, in fact, correct. In many cases, larger values of r_∞ result in larger values of $N_{1/2}$.

It is worth noting that this is *not* the best implementation of Algorithm 2 on VP-1 or VP-3. When the segment size drops below MVL, it becomes inefficient to load and store segments. It is more efficient to perform register-to-register operations, eliminating the overhead of moving elements between memory and vector registers. We will discuss such algorithms in Lecture Notes #3.

N	VP-1		VP-3	
	Cycles	Ops/Cycle	Cycles	Ops/Cycle
20	452	0.0442	428	0.0467
40	607	0.0658	562	0.0712
60	686	0.0874	621	0.0966
64	702	0.0912	633	0.1011
65	782	0.0830	711	0.0914
80	842	0.0950	756	0.1058
100	921	0.1085	815	0.1226
120	1001	0.1199	875	0.1371
128	1033	0.1239	899	0.1424
129	1178	0.1094	1013	0.1272
140	1222	0.1145	1041	0.1344
160	1302	0.1228	1091	0.1466
180	1382	0.1302	1140	0.1578
200	1461	0.1368	1190	0.1680
300	2067	0.1451	1578	0.1901
400	2531	0.1580	1839	0.2174
500	2931	0.1706	2064	0.2422
1000	5461	0.1831	3461	0.2889
1500	8057	0.1862	4864	0.3084
2000	10511	0.1903	6154	0.3250
5000	25782	0.1939	14160	0.3531
10000	50927	0.1964	27184	0.3679

Table 2: Performance of Algorithm 2 on VP-1 and VP-3.

4 Strip-Mining

Strip-mining is the name of a technique used for vectorizing loops of arbitrary length. Consider the following loop:

```
FOR I=1 TO N DO
    Y(I)=Y(I)+X(I)
ENDFOR
```

The execution of this loop on a pipelined architecture with vector-registers, both of length MVL , requires that the vectors X and Y each be partitioned into segments of length at most MVL . The actual length of a segment, VL , is kept in the *vector-length register*. The operations on each segment can then be performed as *vector-instructions*. If VL is stored in the vector-length register, then a vector-instruction operates only on the first VL elements of one or more vector-registers.

Strip-mining results in a number of loops, all but one of which are of length MVL . One segment is of length $N \bmod MVL$. Strip-mining changes the loop above to the following:

```
VL=N mod MVL
LOW=1
FOR J=0 TO N/MVL DO
    FOR I=LOW TO LOW+VL-1 DO
        Y(I)=Y(I)+X(I)
    ENDFOR
    LOW=LOW+VL
    VL = MVL
ENDFOR
```

The term N/MVL represents integer division. The outer loop partitions the arrays X and Y into segments processed in the inner loop. Figure 13 shows the segmentation of the arrays X and Y . The segment length is VL which is equal to MVL for all but the first segment. The time T_{loop} accounts for the loading of the vector-length register, the scalar arithmetic, and the address calculation. In the above example, the vector-length register is set twice. Scalar arithmetic is performed on the variable LOW and on the loop bounds. The overhead of strip-mining is accounted for by the T_{loop} cost.

5 Chaining of Operations

In many computers, the different functional units can operate concurrently. For instance, loading data into registers can take place concurrently with performing additions.

Chaining is the concurrent operation of functional units as a single pipeline.

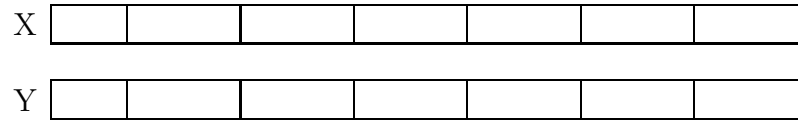


Figure 13: Strip-mining of two vectors.

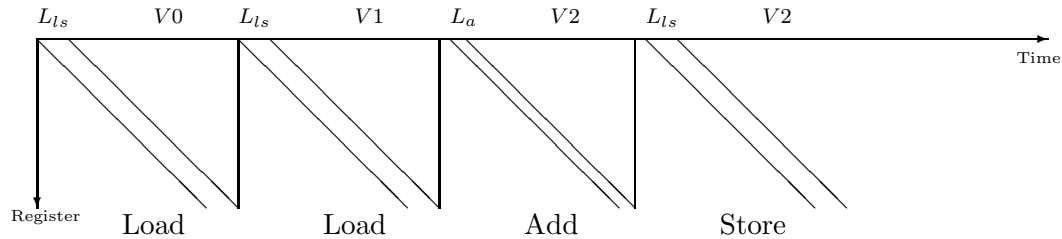


Figure 14: Timing diagram for vector loading and addition without chaining on VP-1.

Consider again the example of adding the elements of an array (i.e., $y = \sum_{i=0}^{N-1} x(i)$). Using Algorithm 2 on our vector processor VP-1, we gave the following instruction sequence:

Load a segment of X
 Load a second segment of X
 Perform a vector add on the two segments
 Store the result

The timing analysis was:

Load first segment	12+VL	
Load second segment	12+VL	
Vector add	6+VL	
Stall for vector add	4	(vector add depends on second load)
Vector store	12+VL	
Stall for vector store	4	(vector store depends on vector add)
Total	$\frac{50+4VL}{}$	

The asymptotic rate, r_∞ , was 0.199 operations/cycle for this sequence.

For the purposes of this example, assume the first vector-instruction loads a segment into vector-register V0, the second loads a segment into V1, and the third adds the contents of V0 and V1, placing the result in V2. The result is then stored, and the process is repeated.

The timing diagram in Figure 14 illustrates the first few vector operations.

For each set of VL elements, $3L_{ls} + L_a + 4VL + 8$ cycles are required, where L_{ls} is the latency of the load/store pipeline, L_a is the latency of the adder pipeline and 8 accounts for the 2 vector stalls. Assuming that the load/store unit and the adder can operate concurrently, chaining a

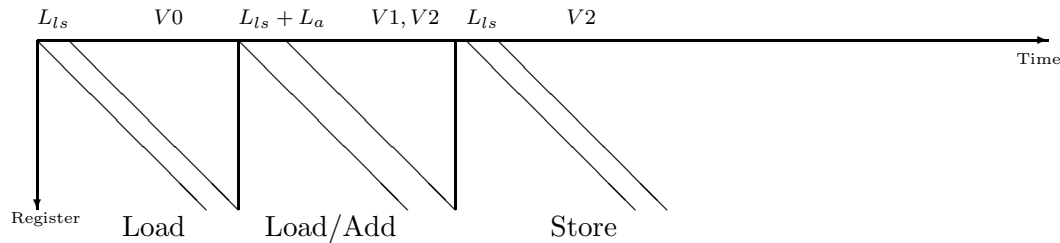


Figure 15: Timing diagram for vector loading and addition with chaining on VP-1.

load-and-add sequence together reduces the number of cycles for the addition of VL elements by VL cycles as shown in Figure 15. It also avoids the vector stall for the add operation because chaining units together eliminates the need for a stall. The total startup time now becomes $3L_{ls} + L_a + 3VL + 4$. The new operations are:

- Load a segment of X
- Load a second segment of X, and add to previous segment
- Store the result

The new timing analysis is:

Load first segment	12+VL	
Load second segment and add	12+6+VL	(chain load and add)
Vector store	12+VL	
Stall for vector store	4	(vector store depends on vector add)
Total	<u>46+3VL</u>	

The asymptotic rate, r_∞ , is now 0.253 operations per second, a 27% improvement over the earlier rate of 0.199 operations per second.

Note that once we have chained the load and addition, it is not possible to chain the store as well. This is because VP-1 has only one load/store path which means that the loads must finish before the stores can begin, making chaining impossible. Chaining a load and a store on VP-1 would be like chaining an add with an add when there is only one addition unit.

If we use VP-3, we can do the loads simultaneously. The additional data paths makes it possible to chain the load-add-stores operations. The new operations are:

- Load two segments of X
- Add segments of X
- Store the result

The new timing analysis is:

Load two segments	12+VL	
Vector add	6	(chained with load)
Vector store	12	(chained with add)
Total	<u>30+VL</u>	

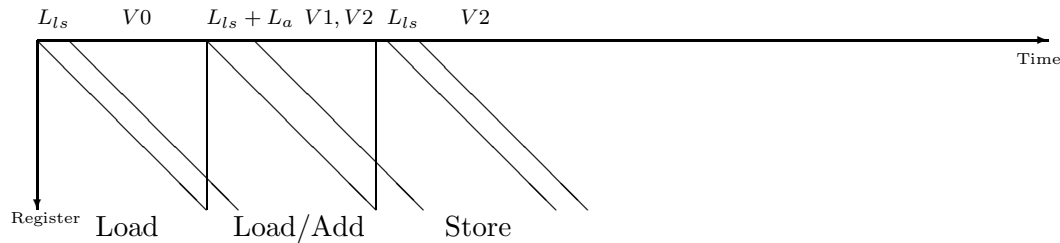


Figure 16: Timing diagram for vector loading and addition with chaining and instruction overlap on VP-1.

The asymptotic rate, r_∞ , is now 0.587 operations per second, a 132% improvement over the VP-1 chained rate of 0.253 operations per second. Note that performance can be further improved by performing the $(N - 1)^{th}$ store concurrently with the N^{th} pair of loads. Table 3 summarizes the performance rates for the various architectures we have analyzed.

	VP-1	VP-3
No Chaining	0.199 ops/cycle	0.388 ops/cycle
Chaining	0.253 ops/cycle	0.587 ops/cycle

Table 3: Asymptotic performance values (r_∞) for computation of $y = \sum_{i=0}^{N-1} x(i)$ using Algorithm 2.

With *instruction overlapping*, or pipelining of the issuing of instructions, the time for the operation can be further reduced as shown in Figure 16.

6 An Example Vector–Instruction Set

The table below describes the vector–instructions which would be found in a typical vector–register machine.

Instruction	Operands	Function
ADDV	V1,V2,V3	Add the elements of V2 to V3; put result in V1
ADDSV	V1,F0,V2	Add F0 to the elements of V2; put result in V1
SUBV	V1,V2,V3	Subtract the elements of V3 from V2, put result in V1
SUBSV	V1,F0,V2	Subtract elements of V2 from F0; put result in V1
SUBVS	V1,V2,F0	Subtract F0 from the elements of V2; put result in V1
MULT	V1,V2,V3	Multiply the elements of V2 to V3; put result in V1
MULTSV	V1,F0,V2	Multiply F0 by the elements of V2; put result in V1
DIVV	V1,V2,V3	Divide the elements of V2 by V3; put result in V1
DIVSV	V1,F0,V2	Divide F0 by the elements of V2; put result in V1
DIVVS	V1,V2,F0	Divide the elements of V2 by F0; put result in V1
LV	V1,R1	Load vector reg. V1 from mem. starting at addr. R1
SV	R1,V1	Store vector reg. V1 into mem. starting at addr. R1
LVWS	V1,(R1,R2)	Load vector reg. V1 from mem. starting at addr. R1 w/stride in R2
SVWS	(R1,R2),V1	Store vector reg. V1 into mem. starting at addr. R1 w/stride in R2
LVI	V1,(R1+V2)	Load vector reg. V1 from mem. with elements at addr. R1+V2(i)
SVI	(R1+V2),V1	Store vector reg. V1 into mem. at addr. R1+V2(i)
CVI	V1,R1	Create an index vector 0, R1, 2R1,..., 63 R1 in V1
S_V	V1,V2	Compare V1 and V2 with result in VM
S_SV	F0,V1	Compare F0 with V1, result in VM
POP	R1,VM	Count the number of 1's in VM, result in R1
CVM		Set VM to all 1's
MOVI2S	VLR,R1	Move the contents of R1 to vector length register
MOVS2I	R1,VLR	Move contents of VLR to R1
MOVF2S	VM,F0	Move the contents of F0 to vector mask register
MOVS2F	F0,VM	Move the contents of VM to F0

7 More Examples

In Section 2.1, three algorithms were introduced for summing the elements of an array. In Section 3.2, the performance of Algorithm 2 was studied. In the next two subsections, a similar analysis is performed for Algorithms 1 and 3. In the final subsection, we consider a problem whose result is a vector.

7.1 Computing $y = \sum_{i=0}^{N-1} x(i)$ with Algorithm 1

Algorithm 1, introduced in Section 2.1, is the naive iterative approach to summation. As shown earlier, this approach is not conducive to pipelining. In fact, Algorithm 1 prevents the use of vector additions.

Instead, scalar additions are performed. Each element of the vector register must be copied to a scalar register and used as input to the adder. We assume that the running sum is kept in another scalar register. For each load of MVL elements, each element will be copied to a scalar register and added to the running sum. Since the operation is strictly scalar, each of the additions requires 6 cycles and is not pipelined. Assuming that register-to-register copies work like vector copies, the time per segment on VP-1 will be:

Load a segment of X	12+VL	
Copies to scalar registers	1×VL	(not pipelined – must perform VL times)
Perform scalar additions	6×VL	(not pipelined – must perform VL times)
Total	$\frac{12+8VL}{12+8VL}$	

Vector stalls are not required because no vector operation depends on the result of another vector operation. The only pipelined vector operations are the loads; everything else is scalar.

At the end, the finally accumulated value must be stored, at a cost of 12+1 for the single store. Since $MVL = 64$, the total time is:

$$\begin{aligned} T_N &= 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 12) + 8N + (12 + 1) \\ &= 23 + 27 \times \left\lceil \frac{N}{64} \right\rceil + 8N \end{aligned}$$

The asymptotic performance, r_∞ , is 0.119 operations/cycle.

If instruction overlap were allowed, the copy cost could be hidden in the 6 cycle addition latency, but we will not complicate the example further by analyzing this possibility.

Since there is no pipelining, VP-3 cannot provide much improvement over VP-1. However, it is possible to load twice as much data per segment, reducing the number of loads by a factor of 2. This would result in the following times:

Load two segments of X	12+VL	
Copy to scalar register	1×2×VL	(not pipelined – must perform 2×VL times)
Perform scalar additions	6×2×VL	(not pipelined – must perform 2×VL times)
Total	$\frac{12+15VL}{12+15VL}$	

At the end, the final value must be stored at a cost of 12+1 cycles. However, this time, the total time is computed based on half as many elements. The total time is:

$$\begin{aligned} T_N &= 10 + \left\lceil \frac{N/2}{64} \right\rceil (15 + 12) + 15(N/2) + (12 + 1) \\ &= 23 + 27 \times \left\lceil \frac{N}{128} \right\rceil + 7.5N \end{aligned}$$

The asymptotic performance, r_∞ , is 0.130 operations/cycle an improvement of 8.5% over the previous result of 0.119 operations/cycle. As predicted, the performance is poor, since very little pipelining is possible. Algorithm 1 would not be used to compute the sum of an array on a vector-register architecture like VP-1 or VP-3.

7.2 Computing $y = \sum_{i=0}^{N-1} x(i)$ with Algorithm 3

Algorithm 3 computes partial sums and then adds them using Algorithm 1 or 2. For the purpose of this example, we will assume Algorithm 2 is used. On VP-1, the time per segment is:

Load first segment of X	12+VL	
Load second segment of X	12+VL	
Vector add	6+VL	
Stall for vector add	4	(vector add must wait for vector load)
Total	$\frac{34+3VL}{}$	

After execution over all segments, Algorithm 2 is used to sum the MVL partial sums. Then a final store is required. The additional one-time cost is:

Algorithm 2 on MVL Elements	C_{alg2}	
Store final result	12+1	
Stall for store	4	(vector store must wait for final vector add)
Total	$\frac{17 + C_{alg2}}{}$	

where C_{alg2} is the number of cycles to sum MVL elements using Algorithm 2. The total cost is:

$$\begin{aligned} T_N &= 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 34) + 3N + 17 + C_{alg2} \\ &= 27 + 49 \times \left\lceil \frac{N}{64} \right\rceil + 3N + C_{alg2} \end{aligned}$$

The cost of performing Algorithm 2 on MVL elements is left as an exercise and appears in Problem Set 1.

If we allow chaining, the second load can be chained with the add, reducing the cost to:

$$T_N = 27 + 45 \times \left\lceil \frac{N}{64} \right\rceil + 2N + C_{alg2}.$$

On VP-3, we can perform the two loads simultaneously. The time per segment is:

Load first and second segments of X	12+VL	
Vector add	6+VL	
Stall for vector add	4	(vector add must wait for vector load)
Total	$\frac{22+2VL}{}$	

The total cost is:

$$\begin{aligned} T_N &= 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 22) + 2N + 17 + C_{alg2} \\ &= 27 + 37 \times \left\lceil \frac{N}{64} \right\rceil + 2N + C_{alg2} \end{aligned}$$

If the load and the add are chained, the cost is reduced to:

$$\begin{aligned} T_N &= 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 22) + N + 17 + C_{alg2} \\ &= 27 + 33 \times \left\lceil \frac{N}{64} \right\rceil + N + C_{alg2} \end{aligned}$$

Since the value of C_{alg2} is independent of N , we can compute r_∞ for all of these techniques, as shown below in Table 3.

	VP-1		VP-3	
	Time	r_∞	Time	r_∞
No Chaining	$27 + 49 \times \lceil N/64 \rceil + 3N + C_{alg2}$	0.266	$27 + 37 \times \lceil N/64 \rceil + 2N + C_{alg2}$	0.388
Chaining	$27 + 45 \times \lceil N/64 \rceil + 2N + C_{alg2}$	0.369	$27 + 33 \times \lceil N/64 \rceil + N + C_{alg2}$	0.660

Table 4: Asymptotic performance values (r_∞) for computation of $y = \sum_{i=0}^{N-1} x(i)$ using Algorithm 3.

7.3 Computing $z(i) = x(i) + y(i)$, $i = 1, 2, \dots, N$

In this example, each segment is complete after it is processed. There is no value to accumulate. On VP-1, the time per segment is:

Load a segment of X	12+VL	
Load a segment of Y	12+VL	
Vector add	6+VL	
Stall for add	4	(vector add depends on load of Y)
Vector Store	12+VL	
Stall for store	4	(vector store depends on vector add)
Total	<u>50+4VL</u>	

The total is:

$$\begin{aligned} T_N &= 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 50) + 4N \\ &= 10 + 65 \times \left\lceil \frac{N}{64} \right\rceil + 4N \end{aligned}$$

If chaining is allowed, then the second load can be chained with the add, reducing the total (by a VL and a stall) to:

$$\begin{aligned} T_N &= 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 46) + 3N \\ &= 10 + 61 \times \left\lceil \frac{N}{64} \right\rceil + 3N \end{aligned}$$

On VP-3, the two loads can be performed simultaneously:

Load segment of X and Y	12+VL	
Vector add	6+VL	
Stall for add	4	(vector add depends on load)
Vector Store	12+VL	
Stall for store	4	(vector store depends on vector add)
Total	<u>38+3VL</u>	

If chaining is allowed, we can chain the load, add, and store because we are using VP-3. This eliminates two VL's and the stalls. The total is reduced to 30+VL. Through some cleverness, it is possible to

improve the time further, but we will not do so here³.

The performance of the various techniques are summarized in Table 5.

	VP-1		VP-3	
	Time	r_∞	Time	r_∞
No Chaining	$10 + 65 \times \lceil N/64 \rceil + 4N$	0.199	$10 + 53 \times \lceil N/64 \rceil + 3N$	0.261
Chaining	$10 + 61 \times \lceil N/64 \rceil + 3N$	0.253	$10 + 45 \times \lceil N/64 \rceil + N$	0.587

Table 5: Performance values for computation of $z(i) = x(i) + y(i)$.

References

- [1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann Publishers, Inc, 1990.
- [2] Peter M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.
- [3] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, 1993.
- [4] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.

³We can actually reduce the time to $22+VL$ (with chaining), if we store the previous result when we load the next two segments. This method was shown during the analysis of Algorithm 2 in Section 3.2 and requires special treatment of the first and last iterations.