Arranging vectors to avoid memory conflicts

A vector instruction written as

\[ C := A + B \]

is interpreted to mean

\[ c_i := a_i + b_i, \quad 0 \leq i \leq N-1 \]

To implement this instruction, we can use an organization like this:

- The memory system supplies one element of A and B on each clock cycle, one to each input stream.
- The arithmetic unit produces one element of C on each clock cycle.

The challenge is to avoid memory conflicts. This is a challenge because—

- A conventional memory module can perform only a single read or write in a cycle.
- Additional bandwidth is needed for I/O operations.
- Still more bandwidth is needed for instruction fetches.

Why aren’t these necessarily serious problems?
Instead of interleaved memory, what other technique could be used to increase bandwidth?

Consider our example from the previous page, and suppose that a memory access takes two processor cycles.

What memory bandwidth is needed to service the pipeline at full speed? ___ words per cycle.

Suppose our multiport memory system is an 8-way interleaved memory.

Consider an ideal case—
• **A[0]** is in module 0.
• **B[0]** is in module 2.
• **C[0]** is in module 4.

Successive elements lie in successive memory modules.

If the pipeline has four stages, the diagram on the next page shows which stages and memory modules are in use in each cycle.

Let us see what is happening at each time period.
At time period 0, accesses to \( A[0] \) and \( B[0] \) are initiated.

At time period 1, these accesses are still in progress, and accesses to \( A[1] \) and \( B[1] \) are initiated.

At time period 2,
- accesses to \( A[1] \) and \( B[1] \) are still in progress, and
- accesses to \( A[0] \) and \( B[0] \) have finished, and these operands begin to flow through the pipeline.

At time period 6, the first set of operands has finished flowing through the pipeline.

Now the result is written, to memory module 4, which has just finished reading \( A[4] \).

Notice that there are no conflicts for any memory module.

However, it is not always possible to arrange vectors in memory like this.

- Vector \( C \) cannot begin in modules 0, 1, 5, 6, or 7, or there will be conflicts.
- But \( C \) might be an operand of other vector instructions that, similarly, prevent it from beginning in modules 2, 3, or 4.

Then conflicts are inevitable.

One way of overcoming these conflicts is through the use of variable delays.

If such delays are inserted in one input stream and the output stream, then it is possible to avoid conflicts by

- prefetching one stream of operands, and
- delaying the storing of the result.
For example, if all vectors start in module 0, conflicts can be avoided by—

- delaying the fetching of \( B \) by _____ cycles, and
- delaying the storing of \( C \) by _____ cycles.

A timing diagram is given below.

How should variable delays be implemented?

One way is to use a tapped delay line:

- The \( D \) modules are unit delays.
• The Delay amount is decoded and used to select the output of one of the delays.

Another way is to use a FIFO.
• One cell can be read while another is being written.

\[ \text{Read/write registers} \]

![Diagram of a FIFO](image)

Two registers hold the addresses within the memory where accesses take place.

• The \textit{write address} register is initialized to 0.
• The \textit{read address} register is initialized to \(-d\), where \(d\) is the required delay.
• Each register is incremented by 1 in each clock cycle.
• Accesses to negative addresses are ignored.

If a delay of four cycles is desired, what is the read-address register initialized to?

When is the first datum retrieved from the memory?

When is the \(i\)th datum retrieved from the queue?

Special action is required when the specified delay is

This organization has advantages over the delay stages, since only two addresses change state each clock period.
Handling long vectors

There is one disadvantage incurred by any variable-delay strategy.

Changing operations in a pipeline imposes a certain pipeline overhead time $t_0$, consisting of startup and flushing delays.

What do variable delays do to this penalty?

The time for a pipeline operation is $t_0 + L t$, where $L$ is the number of operand pairs and $t$ is the latency between two successive pairs.

Therefore, it is most efficient to operate on long vectors.

But some algorithms, like Gaussian elimination produce vectors of successively decreasing length. Hence, long pipelines exact a large penalty.

Vector length and stride

[H&P, §B.3] How can a program handle a vector length that is not the same as the length of the vector registers? E.g., in DLXV, suppose that the vector length isn’t 64.

Another problem is what happens when the vector length is unknown at compile time. For example, in the code below, it is not obvious what the vector length would be.

```
for i := 1 to N do
  Y[i] := a × X[i] + Y[i];
```

The value of $N$ may not be known until run time. It might also be a parameter to a procedure, and thus be subject to change during execution.

The vector-length register, or VLR, is used to handle these problems. It controls the length of any vector operation, including loads and stores.

The VLR can contain any value $\leq$ the maximum vector length (MVL) of the machine.
If the value of \( N \) is unknown at compile time, and thus may be > MVL, *strip mining* is used. It divides the vectors into chunks with length \( \leq \) MVL.

Here is a strip-mined version of the SAXPY loop.

```plaintext
var low: integer;    { Low bound for this chunk }
VL: 1 . . MVL;      { Length of this chunk }
begin
  low := 1;
  VL := N mod MVL;  { Measure the odd-size chunk. }
  for j := 0 to N/MVL do    { Outer loop }
    begin
      for i := low to low + VL – 1 do    { Runs for length VL }
        Y[i] := a \times X[i] + Y[i];
      low := low + VL;    { Start of next chunk }
      VL := MVL;          { All chunks but the 1st are max. length }
    end;
end;
```

The term \( N/MVL \) is calculated by (truncating) integer division.

The length of the first chunk is and the lengths of all other chunks are

Here is a diagram of how the vector is divided up.

<table>
<thead>
<tr>
<th>Value of ( j )</th>
<th>N/MVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td></td>
</tr>
<tr>
<td>( 1 )</td>
<td></td>
</tr>
<tr>
<td>( 2 )</td>
<td></td>
</tr>
<tr>
<td>( 3 )</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
1 & . . m & \quad (m+1) & . . & (m+MVL+1) & . . & (m+2MVL+1) & . . & (N – MVL + 1). \\
\text{Range of } i & \quad m + MVL & \quad m + 2MVL & \quad m + 3MVL & \quad . . & \quad N \\
& \quad (m = N \mod MVL)
\end{align*}
\]

If multiple vector operations execute in parallel, the hardware must copy the value of VLR when
What are the time penalties for strip mining? There are two kinds:

- 
- 

For this code,

\[ \text{for } i := 1 \text{ to } N \text{ do } A[i] := B[i]; \]

the compiler will generate two nested loops. The inner loop contains a sequence of two operations,

What is the startup overhead?

The store latency can be ignored, since nothing depends on it.

If the vector is short, the startup cost is very high. For a vector length of 2, it is

For a vector length of 64, it is

**Vector stride**

Suppose the elements of a vector are not sequential. This is bound to happen in the case of a matrix multiply, either for the rows or the columns.

Here is the straightforward code for multiplying \(100 \times 100\) matrices:

\[
\text{for } i := 1 \text{ to } 100 \text{ do }
  \begin{align*}
  &\text{for } j := 1 \text{ to } 100 \text{ do } \\
  &\quad \text{begin} \\
  &\quad \quad C[i, j] := 0; \\
  &\quad \quad \text{for } k := 1 \text{ to } 100 \text{ do } \\
  &\quad \quad \quad C[i, j] := C[i, j] + A[i, k] \times B[k, j]; \\
  &\quad \text{end; }
  \end{align*}
\]

In the inner loop, we could vectorize the multiplication of a row of \(A\) with a column of \(B\), and strip-mine the loop with ____ as the index variable.
To do this, we need to know whether the array is stored in row-major or column-major order.

- In row-major order, used by most languages except Fortran, elements of a row (e.g., $B[i, k]$ and $B[i, k+1]$) are adjacent.

- In column-major order, used by Fortran, elements of a column (e.g., $B[i, k]$ and $B[i+1, k]$) are adjacent.

**Row-major order:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

**Column-major order:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1)</td>
<td>(1, 2)</td>
<td>...</td>
<td>(1, 100)</td>
<td>(2, 1)</td>
<td>(2, 2)</td>
<td>...</td>
<td>(100, 1)</td>
<td>(100, 2)</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>(2, 2)</td>
<td>...</td>
<td>(100, 1)</td>
<td>(1, 2)</td>
<td>(2, 2)</td>
<td>...</td>
<td>(1, 100)</td>
<td>(2, 100)</td>
</tr>
</tbody>
</table>

**Nonunit stride and memory conflicts**

The distance separating elements that are to be loaded into a vector register is called the *stride*.

In the example above, if row-major order is used,

- matrix $B$ has a stride of
- matrix $A$ has a stride of
A stride greater than one is called a nonunit stride.

Vector machines typically have instructions for loading vectors of nonunit strides.

Like the vector length, the stride may not be known at compile time.

The DLXV instruction LVWS (load vector with stride) can be used to load the vector into a register.

The counterpart of LVWS is SVWS.

In some vector machines, the value of the stride is taken from a special register, so there needn’t be special LVWS/DVWS instructions.

*Memory conflicts:* When discussing how many modules were necessary to allow a vector operation to proceed at full speed, we saw that the number of modules had to be \( \geq \) memory-access time in clock cycles.

However, if nonunit strides are used, the operation may nonetheless slow down due to memory-bank conflicts, if operands are requested from the same module at a higher rate than permitted by the memory-access time.

*Example:* Suppose we have—

- 16 memory banks,
- with an access time of 12 clock cycles.

How long will it take to complete a 64-element vector load with a stride of 1?

How long will it take to complete a 64-element vector load with a stride of 32?