available in high-level language vs. speed of access.

Trading off: less flexible addressing and smaller length (not
of length) less than or equal to VT.

Addressed in a monolithic fashion which can contain a vector
addressed in a set of locations which are

Definition: A vector register is a set of locations which are

memory-to-memory architectures

able to exploit other optimizations more easily than

can have less contention

less memory intensive: requires fewer ports to memory,

(sometimes)

Motivation:

Register-based Vector Processors
end do

(\lambda_0 + \lambda_1 \lambda_1) = \lambda_2 (\lambda_1) \lambda_1 = \lambda_2

do \lambda_1 = 1 \lambda_1

Example: \lambda_0 + \lambda_1 = \lambda_2

is equivalent to

assumed to be on chip in the vector register file.

Restriction that length is less than \lambda_1 and with all data

Operations work in the same way as defined earlier with the
Loop index computations.

- Often specific instructions exist to support address and
  responsibility of the user (compiler).

- Explicit loop control and address computation is the
  partitioned and operated on one piece at a time

- Large vectors (\( T \Lambda < u \)) from memory must be
These are some of the most successful architectures in history:

Example: Cray 1, Cray 2, Cray X-MP, Cray X-MP, C90.

Operations are not included in the instruction set. Often vector operands have vector register operands. Often vector reduction load/store instructions can access memory. All others

Vector LOAD/STORE (Vector RISC): only vector

Two broad classes of vector instruction sets:
Note that this is equivalent to a three instruction series.

\[ \forall \in W \ast 0 \vdash f + 0 \rightarrow \forall I \]

and IBM 360/370, typical instruction:

\[ \text{ Alliant FX/80} \]

much richer and more complex. Example: Alliant FX/80

operands come directly from memory. Instruction set

Vector CISC: Vector instructions may have one or their

Vector LOAD/STORE set.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVS</td>
<td>V1, P0, V2</td>
</tr>
<tr>
<td>DIVS</td>
<td>V1, V2, P0</td>
</tr>
<tr>
<td>DIVS</td>
<td>V1, V2, V3</td>
</tr>
<tr>
<td>MULTSV</td>
<td>V1, P0, V2</td>
</tr>
<tr>
<td>MULTV</td>
<td>V1, V2, V3</td>
</tr>
<tr>
<td>SUBVS</td>
<td>V1, P0, V2</td>
</tr>
<tr>
<td>SUBSV</td>
<td>V1, V2, P0</td>
</tr>
<tr>
<td>SUBV</td>
<td>V1, V2, V3</td>
</tr>
<tr>
<td>ADDSV</td>
<td>V1, P0, V2</td>
</tr>
<tr>
<td>ADDV</td>
<td>V1, V2, V3</td>
</tr>
<tr>
<td>SWVS</td>
<td>R1, R2, V1</td>
</tr>
<tr>
<td>SY</td>
<td>R1, V1</td>
</tr>
<tr>
<td>SS</td>
<td>R1, R2</td>
</tr>
<tr>
<td>LWS</td>
<td>V1, R1, R2</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
</tr>
<tr>
<td>LS</td>
<td>R1, R2</td>
</tr>
<tr>
<td>Instruction</td>
<td>Function</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>R1 → VLR</td>
<td>MOVLSL</td>
</tr>
<tr>
<td>VLR → R1</td>
<td>MOVLSL</td>
</tr>
<tr>
<td>VI → VI, V2</td>
<td>EXPAND</td>
</tr>
<tr>
<td>VI, V2 → VI</td>
<td>COMPRESS</td>
</tr>
<tr>
<td>VI → VI, V1</td>
<td>T-LOGIC</td>
</tr>
<tr>
<td>VI, V1 → VI</td>
<td></td>
</tr>
<tr>
<td>VI → VI, V2</td>
<td>MOVMSR</td>
</tr>
<tr>
<td>VI, V2 → VI</td>
<td></td>
</tr>
<tr>
<td>VI → I</td>
<td>GVN</td>
</tr>
<tr>
<td>I, V1 → VI</td>
<td>POP</td>
</tr>
<tr>
<td>V1, R1, R2</td>
<td>MVSET</td>
</tr>
<tr>
<td>R1 → V1 (R2)</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Instructions</td>
</tr>
</tbody>
</table>
Instructions are not listed.

extensions throughout the course (e.g., scalar-scalar
by Hennessy and Patterson. It will be used with some
This instruction set is a slight modification of that used

\[ \text{detected later. For now it is used with all bits set to 1}. \]

(\text{We will discuss the use of VM in more
operations. VM are assumed to control all vector
register, VM, and the vector mask

\text{Logical operation.}

In the logical operations the – can be replaced by any

\text{Logical operation.} \]
(Note we do not assume any loop index support instructions.)

the vector registers are assumed to have 64 elements each. A, Rb is the base address of B and \( P \) is the value of \( P \) and R0 is assumed to have the value 0, Ra is the base address of R.

\[
(I)B * s = (I)A
\]

\[
DO \quad 0 = i \leq 256
\]

Simple Example
parallel.

Chaining allows these two instructions to proceed in

and \( V_4(i) \) to be available.

Actually, the computation of \( V_5(i) \) needs only for \( V_3 \) to

be computed.

enforced at the register level, i.e., it waits for all of \( V_3 \) to

instruction 2 cannot proceed when dependences are

\[
\begin{align*}
2. & \quad V_5 \rightarrow V_3 \rightarrow V_4 \\
1. & \quad V_3 \rightarrow V_1 + V_2
\end{align*}
\]

which allows functional unit parallelism:

Consider the following two operations on a vector processor:

Chaining
When \( V_3(i) \) has been loaded from memory, \( V_3 \) from memory and the multiplication would start. For example, the first instruction could be a load of units. Memory ports can also be chained with functional operand type. Note that the dependence is destination-to-source continue as each element is produced. The second multiplication will begin to compute when the first element of \( V_3 \) is available from the addition and
dependence at the element level.

• Tailoring enforces this type of source-to-destination

soon as it has been used in the addition.

• Actually, the load from memory can overwrite \( V_1(i) \) as

computed since the addition requires \( V_1 \),

are enforced at the register level. It must wait until \( V_3 \) is

The second operation cannot proceed when dependences

\[
\begin{align*}
&1. \ V_3 \rightarrow V_1 + V_2 \\
&2. \ V_1 \rightarrow \text{MEM}
\end{align*}
\]

Consider the following two operations:

Tailoring
\[ V_1 \rightarrow V_1 + V_2 \]

*the same operation as in*

seen when an operand is both a source and a destination for

Most machines allow a limited type of tailgating. This type is
and how they interact to affect performance.

- Tailgating is not relevant to this example.
- Chaining
- Functional unit parallelism within a vector processor

Consider
Three registers used, time = \( k \times (4 \text{ chimneys}) \)

<table>
<thead>
<tr>
<th>( \vdots )</th>
<th>+</th>
<th>( A_2 )</th>
<th>B_2</th>
<th>C_2</th>
<th>A_1</th>
<th>B_1</th>
<th>C_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdots )</td>
<td>+</td>
<td>( \text{adder} )</td>
<td>( \text{port} )</td>
<td>( \text{time} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I. No chaining or tailoring, no parallelism.

Processor with one port to memory.

Assume a register-based vector partitioned into substrectors of length \( \forall \text{ denoted } A, B, \text{ and } C \) are vectors.

Example: \( A \rightarrow B + C \), where \( A, B, \text{ and } C \) are vectors.
Four registers used, time $\approx K^*$ (3 chimneys). Port saturated.

<table>
<thead>
<tr>
<th>...</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>: adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>A</td>
<td>V</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1</td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

$\leftarrow$ time

2. no chaining or tailgate, parallelism
Three registers used, time \( \approx k \cdot 3 \text{ cycles} \), Port saturated.

<table>
<thead>
<tr>
<th></th>
<th>( + )</th>
<th>( + )</th>
<th>( \text{adder} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( A_1 )</td>
<td>( B_1 )</td>
<td>( C_1 )</td>
</tr>
<tr>
<td>( \cdots )</td>
<td>( A_2 )</td>
<td>( B_2 )</td>
<td></td>
</tr>
<tr>
<td>( \cdots )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \leftarrow \text{time} \)

3. chaining and parallelism
Note that computation is free here! Computation and data transfer (communication) have been overlapped.

- Non-chained processor such as a memory-to-memory one.
- Operations that would require a specific instruction in a chaining dynamically creates efficient versions of.
- Chains and parallelism together can often achieve the complex than expected.
- Often parallelism of functional units and ports is enough to saturate key resources but the coding can be more
pipeline, but how about memory to/from register?

data supply rate – registers can keep up with the

determining performance limits

performance models for vector processors

(model (conditionals and nonlinear addresses)
delegation from vector/data parallel computation

Reductions (subject of first homework)

What's Next?