Problems with Superscalar approach

• Limits to conventional exploitation of ILP:
  1) *Pipelined clock rate*: Each increase in clock rate has corresponding CPI increase (branches, other hazards). (pipeline latency)
  2) *Instruction fetch and decode*: At some point, its hard to fetch and decode more instructions per clock cycle. (ILP)
  3) *Cache hit rate*: Some long-running (scientific) programs have very large data sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality. (memory latency).
X86 CPU Cache/Memory Performance Example:
AMD Athlon T-Bird Vs. Intel PIII, Vs. P4

AMD Athlon T-Bird 1GHz
L1: 64K INST, 64K DATA (3 cycle latency),
   both 2-way
L2: 256K 16-way 64 bit
   Latency: 7 cycles
   L1,L2 on-chip

Intel P4, 1.5 GHz
L1: 8K INST, 8K DATA (2 cycle latency)
   both 4-way
   96KB Execution Trace Cache
L2: 256K 8-way 256 bit, Latency: 7 cycles
   L1,L2 on-chip

Intel PIII 1 GHz
L1: 16K INST, 16K DATA (3 cycle latency)
   both 4-way
L2: 256K 8-way 256 bit, Latency: 7 cycles
   L1,L2 on-chip


From 551
Increasing Instruction-Level Parallelism

- A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop
  - (i.e. Loop Level Parallelism, LLP).
- This is accomplished by **unrolling the loop** either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present.
- In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

```plaintext
for (i=1; i<=1000; i=i+1;)
    x[i] = x[i] + y[i];
```

- In vector machines, utilizing vector instructions is an important alternative to exploit loop-level parallelism,
- Vector instructions operate on a number of data items. The above loop would require just four such instructions.

From 551
Loop-Level Parallelism (LLP) Analysis

- LLP analysis is normally done at the source level or close to it since assembly language and target machine code generation introduces a loop-carried dependence, in the registers used for addressing and incrementing.

- Instruction level parallelism (ILP) analysis is usually done when instructions are generated by the compiler.

- Analysis focuses on whether data accesses in later iterations are data dependent on data values produced in earlier iterations.

  e.g. in
  
  ```
  for (i=1; i<=1000; i++)
      x[i] = x[i] + s;
  ```

  the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of \( X[i] \) twice is within a single iteration.
LLP Analysis Examples

• In the loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];  /* S1 */
    B[i+1] = B[i] + A[i+1];}  /* S2 */
```

- **S1** uses a value computed in an earlier iteration, since iteration i computes \( A[i+1] \) read in iteration \( i+1 \) (loop-carried dependence, prevents parallelism).

- **S2** uses the value \( A[i+1] \), computed by **S1** in the same iteration (not loop-carried dependence).

From 551
LLP Analysis Examples

- In the loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];    /* S1 */
    B[i+1] = C[i] + D[i];  /* S2 */
}
```

- S1 uses a value computed by S2 in a previous iteration (loop-carried dependence)
- This dependence is not circular (neither statement depend on itself; S1 depends on S2 but S2 does not depend on S1.
- Can be made parallel by replacing the code with the following:

```c
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

From 551
LLP Analysis Example

Original Loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];  /* S1 */
    B[i+1] = C[i] + D[i];  /* S2 */
}

B[100] = C[99] + D[99];
B[101] = C[100] + D[100];
```

Modified Parallel Loop:

```
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

From 551
Alternative Model: Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

**SCALAR**

(1 operation)

\[ \text{Add.d F3, F1, F2} \]

**VECTOR**

(N operations)

\[ \text{addv.d v3, v1, v2} \]
Vectors vs. Single-issue Scalar

Single-issue Scalar

- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access in whole cache lines

Vector

- One instruction fetch, decode, dispatch per vector
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
- Bypass cache
- One TLB lookup per group of loads or stores
- Move only necessary data across chip boundary
Vector vs. Superscalar

**Superscalar**
- Control logic grows quadratically with issue width
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy

**Vector**
- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
  - Whether to use vector mask or compress/expand for conditionals
Properties of Vector Processors

- Each result in a vector operation is independent of previous results
  - long pipelines used, compiler ensures no dependencies
  - higher clock rate

- Vector instructions access memory with known pattern
  - highly interleaved memory with multiple banks used.
  - amortize memory latency of over 64 elements
  - no (data) caches required! (Do use instruction cache)

- Reduces branches and branch problems in pipelines

- Single vector instruction implies lots of work (- loop)
  - fewer instruction fetches
Changes to scalar processor to run vector instructions

- Decode vector instructions.
- Send scalar registers to vector unit (vector-scalar ops).
- Synchronization for results back from vector register, including exceptions.
- Things that don’t run in vector don’t have high ILP, so can make scalar CPU simple.
## Operation & Instruction Count:
### RISC v. Vector Processor

<table>
<thead>
<tr>
<th>Program</th>
<th>Spec92fp Operations (Millions)</th>
<th>Spec92fp Instructions (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RISC</td>
<td>Vector</td>
</tr>
<tr>
<td>swim256</td>
<td>115</td>
<td>95</td>
</tr>
<tr>
<td>hydro2d</td>
<td>58</td>
<td>40</td>
</tr>
<tr>
<td>nasa7</td>
<td>69</td>
<td>41</td>
</tr>
<tr>
<td>su2cor</td>
<td>51</td>
<td>35</td>
</tr>
<tr>
<td>tomcatv</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>wave5</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>mdljdp2</td>
<td>32</td>
<td>52</td>
</tr>
</tbody>
</table>

Vector reduces ops by 1.2X, instructions by 20X
Basic Vector Architecture

- A vector processor typically consists of an ordinary pipelined scalar unit plus a vector unit.

- The scalar unit is basically no different advanced pipelined CPUs, commercial vector machines have included both out-of-order scalar units (NEC SX/5) and VLIW scalar units (Fujitsu VPP5000).

- Types of architecture for vector processors:
  - *Memory-memory vector processors*: all vector operations are memory to memory
  - *Vector-register processors*: all vector operations between vector registers (except load and store)
    - Vector equivalent of load-store architectures
    - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC
    - We assume vector-register for rest of lecture
Basic Structure of Vector Register Architecture

Main memory

Vector load-store

Vector registers

Scalar registers

FP add/subtract
FP multiply
FP divide
Integer
Logical

EECC722 - Shaaban
Components of Vector Processor

- **Vector Register**: fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-32 vector registers, each holding 64-128 64-bit elements

- **Vector Functional Units (FUs)**: fully pipelined, start new operation every clock
  - typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit

- **Vector Load-Store Units (LSUs)**: fully pipelined unit to load or store a vector; may have multiple LSUs

- **Scalar registers**: single element for FP scalar or address

- Cross-bar to connect FUs, LSUs, registers
# Example Vector-Register Architectures

<table>
<thead>
<tr>
<th>Processor (year)</th>
<th>Clock rate (MHz)</th>
<th>Vector registers</th>
<th>Elements per register (64-bit elements)</th>
<th>Vector arithmetic units</th>
<th>Vector load-store units</th>
<th>Lanes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray-1 (1976)</td>
<td>80</td>
<td>8</td>
<td>64</td>
<td>6: FP add, FP multiply, FP reciprocal, integer add, logical, shift</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cray X-MP (1983)</td>
<td>118</td>
<td>8</td>
<td>64</td>
<td>8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity</td>
<td>2 loads 1 store</td>
<td>1</td>
</tr>
<tr>
<td>Cray Y-MP (1988)</td>
<td>166</td>
<td>8</td>
<td>64</td>
<td>5: FP add, FP multiply, FP reciprocal/sqrt, integer add/shift/population count, logical</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cray-2 (1985)</td>
<td>244</td>
<td>8</td>
<td>64</td>
<td>5: FP add, FP multiply, FP reciprocal/sqrt, integer add/shift/population count, logical</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fujitsu VP100/ VP200 (1982)</td>
<td>133</td>
<td>8–256</td>
<td>32–1024</td>
<td>3: FP or integer add/logical, multiply, divide</td>
<td>1</td>
<td>1 (VP100) 2 (VP200)</td>
</tr>
<tr>
<td>Hitachi S810/ S820 (1983)</td>
<td>71</td>
<td>32</td>
<td>256</td>
<td>4: FP multiply-add, FP multiply/divide-add unit, 2 integer add/logical</td>
<td>3 loads 1 store</td>
<td>1 (S810) 2 (S820)</td>
</tr>
<tr>
<td>Convex C-1 (1985)</td>
<td>10</td>
<td>8</td>
<td>128</td>
<td>2: FP or integer multiply/divide, add/logical</td>
<td>1</td>
<td>1 (64 bit) 2 (32 bit)</td>
</tr>
<tr>
<td>NEC SX/2 (1985)</td>
<td>167</td>
<td>8 + 32</td>
<td>256</td>
<td>4: FP multiply/divide, FP add, integer add/logical, shift</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Cray C90 (1991)</td>
<td>240</td>
<td>8</td>
<td>128</td>
<td>8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity</td>
<td>2 loads 1 store</td>
<td>2</td>
</tr>
<tr>
<td>Cray T90 (1995)</td>
<td>460</td>
<td>8</td>
<td>128</td>
<td>4: FP or integer add/logical, divide</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>NEC SX/5 (1998)</td>
<td>312</td>
<td>8 + 64</td>
<td>512</td>
<td>4: FP or integer add/logical, multiply, divide, logical</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Fujitsu VPP5000 (1999)</td>
<td>300</td>
<td>8–256</td>
<td>128–4096</td>
<td>3: FP or integer multiply, add/logical, divide</td>
<td>1 load 1 store</td>
<td>16</td>
</tr>
<tr>
<td>Cray SV1 (1998)</td>
<td>300</td>
<td>8</td>
<td>64</td>
<td>8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity</td>
<td>1 load-store 1 load</td>
<td>2 8 (MSP)</td>
</tr>
<tr>
<td>SV1ex (2001)</td>
<td>500</td>
<td>8</td>
<td>64</td>
<td>5: FP multiply, FP divide, FP add, integer add/shift, logical</td>
<td>1 load-store</td>
<td>1</td>
</tr>
<tr>
<td>VMIPS (2001)</td>
<td>500</td>
<td>8</td>
<td>64</td>
<td>5: FP multiply, FP divide, FP add, integer add/shift, logical</td>
<td>1 load-store</td>
<td>1</td>
</tr>
</tbody>
</table>
# Vector Linpack Performance (MFLOPS)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Clock</th>
<th>100x100</th>
<th>1kx1k</th>
<th>Peak (Procs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray 1</td>
<td>1976</td>
<td>80 MHz</td>
<td>12</td>
<td>110</td>
<td>160 (1)</td>
</tr>
<tr>
<td>Cray XMP</td>
<td>1983</td>
<td>120 MHz</td>
<td>121</td>
<td>218</td>
<td>940 (4)</td>
</tr>
<tr>
<td>Cray YMP</td>
<td>1988</td>
<td>166 MHz</td>
<td>150</td>
<td>307</td>
<td>2,667 (8)</td>
</tr>
<tr>
<td>Cray C-90</td>
<td>1991</td>
<td>240 MHz</td>
<td>387</td>
<td>902</td>
<td>15,238 (16)</td>
</tr>
<tr>
<td>Cray T-90</td>
<td>1996</td>
<td>455 MHz</td>
<td>705</td>
<td>1603</td>
<td>57,600 (32)</td>
</tr>
<tr>
<td>Conv. C-1</td>
<td>1984</td>
<td>10 MHz</td>
<td>3</td>
<td>--</td>
<td>20 (1)</td>
</tr>
<tr>
<td>Conv. C-4</td>
<td>1994</td>
<td>135 MHz</td>
<td>160</td>
<td>2531</td>
<td>3240 (4)</td>
</tr>
<tr>
<td>Fuj. VP200</td>
<td>1982</td>
<td>133 MHz</td>
<td>18</td>
<td>422</td>
<td>533 (1)</td>
</tr>
<tr>
<td>NEC SX/2</td>
<td>1984</td>
<td>166 MHz</td>
<td>43</td>
<td>885</td>
<td>1300 (1)</td>
</tr>
<tr>
<td>NEC SX/3</td>
<td>1995</td>
<td>400 MHz</td>
<td>368</td>
<td>2757</td>
<td>25,600 (4)</td>
</tr>
</tbody>
</table>
How To Pick Vector Length?

• Longer good because:
  1) Hide vector startup
  2) lower instruction bandwidth
  3) tiled access to memory reduce scalar processor memory bandwidth needs
  4) if know max length of app. is < max vector length, no strip mining overhead
  5) Better spatial locality for memory access

• Longer not much help because:
  1) diminishing returns on overhead savings as keep doubling number of element
  2) need natural app. vector length to match physical register length, or no help
How To Pick Number of Vector Registers?

• More Vector Registers:
  1) Reduces vector register “spills” (save/restore)
     – 20% reduction to 16 registers for su2cor and tomcatv
     – 40% reduction to 32 registers for tomcatv
     – others 10%-15%
  2) aggressive scheduling of vector instructions: better compiling to take advantage of ILP

• Fewer:
  Fewer bits in instruction format (usually 3 fields)
Vector Implementation

• Vector register file:
  – Each register is an array of elements
  – Size of each register determines maximum vector length (MVL) supported.
  – Vector length register (VLR) determines vector length for a particular operation
• Multiple parallel execution units = “lanes” (sometimes called “pipelines” or “pipes”)

Structure of a Vector Unit Containing Four Lanes

Lane 0
- FP add pipe 0
- Vector registers: elements 0, 4, 8, ...
- FP mul. pipe 0

Lane 1
- FP add pipe 1
- Vector registers: elements 1, 5, 9, ...
- FP mul. pipe 1

Lane 2
- FP add pipe 2
- Vector registers: elements 2, 6, 10, ...
- FP mul. pipe 2

Lane 3
- FP add pipe 3
- Vector registers: elements 3, 7, 11, ...
- FP mul. pipe 3

Vector load-store unit
Using multiple functional units to improve the performance of a single vector add instruction

(a) has a single add pipeline and can complete one addition per cycle. The machine shown in (b) has four add pipelines and can complete four additions per cycle.
Vector Memory operations

- Load/store operations move groups of data between registers and memory.
- Three types of addressing:
  - **Unit stride**
    - Fastest
  - **Non-unit (constant) stride**
  - **Indexed** (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize
# The VMIPS Vector FP Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV.D</td>
<td>V1, V2, V3</td>
<td>Add elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>ADDVS.D</td>
<td>V1, V2, F0</td>
<td>Add F0 to each element of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBV.D</td>
<td>V1, V2, V3</td>
<td>Subtract elements of V3 from V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBVS.D</td>
<td>V1, V2, F0</td>
<td>Subtract F0 from elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBSV.D</td>
<td>V1, F0, V2</td>
<td>Subtract elements of V2 from F0, then put each result in V1.</td>
</tr>
<tr>
<td>MULV.D</td>
<td>V1, V2, V3</td>
<td>Multiply elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>MULVS.D</td>
<td>V1, V2, F0</td>
<td>Multiply each element of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVV.D</td>
<td>V1, V2, V3</td>
<td>Divide elements of V2 by V3, then put each result in V1.</td>
</tr>
<tr>
<td>DIVVS.D</td>
<td>V1, V2, F0</td>
<td>Divide elements of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVSV.D</td>
<td>V1, F0, V2</td>
<td>Divide F0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Load vector register V1 from memory starting at address R1.</td>
</tr>
<tr>
<td>SV</td>
<td>R1, V1</td>
<td>Store vector register V1 into memory starting at address R1.</td>
</tr>
<tr>
<td>LVWS</td>
<td>V1, (R1, R2)</td>
<td>Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.</td>
</tr>
<tr>
<td>SVWS</td>
<td>(R1, R2), V1</td>
<td>Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.</td>
</tr>
<tr>
<td>LVI</td>
<td>V1, (R1+V2)</td>
<td>Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.</td>
</tr>
<tr>
<td>SVI</td>
<td>(R1+V2), V1</td>
<td>Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.</td>
</tr>
<tr>
<td>CVI</td>
<td>V1, R1</td>
<td>Create an index vector by storing the values 0, 1 × R1, 2 × R1, ..., 63 × R1 into V1.</td>
</tr>
<tr>
<td>S--V.D</td>
<td>V1, V2</td>
<td>Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.</td>
</tr>
<tr>
<td>POP</td>
<td>R1, VM</td>
<td>Count the 1s in the vector-mask register and store count in R1.</td>
</tr>
<tr>
<td>CVM</td>
<td>Set the vector-mask register to all 1s.</td>
<td></td>
</tr>
<tr>
<td>MTC1</td>
<td>VLR, R1</td>
<td>Move contents of R1 to the vector-length register.</td>
</tr>
<tr>
<td>MFC1</td>
<td>R1, VLR</td>
<td>Move the contents of the vector-length register to R1.</td>
</tr>
<tr>
<td>MVTM</td>
<td>VM, F0</td>
<td>Move contents of F0 to the vector-mask register.</td>
</tr>
<tr>
<td>MVFM</td>
<td>F0, VM</td>
<td>Move contents of vector-mask register to F0.</td>
</tr>
</tbody>
</table>
**DAXPY (Y = a * X + Y)**

**Assuming vectors X, Y are length 64**

**Scalar vs. Vector**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,a</td>
<td>load scalar a</td>
</tr>
<tr>
<td>LV V1,Rx</td>
<td>load vector X</td>
</tr>
<tr>
<td>MUL.VS.D V2,V1,F0</td>
<td>vector-scalar mult.</td>
</tr>
<tr>
<td>LV V3,Ry</td>
<td>load vector Y</td>
</tr>
<tr>
<td>ADDV.D V4,V2,V3</td>
<td>add</td>
</tr>
<tr>
<td>SV Ry,V4</td>
<td>store the result</td>
</tr>
</tbody>
</table>

L.D F0,a

DADDIU R4,Rx,#512 ;last address to load

**loop:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F2, 0(Rx)</td>
<td>load X(i)</td>
</tr>
<tr>
<td>MUL.D F2,F0,F2</td>
<td>a*X(i)</td>
</tr>
<tr>
<td>L.D F4, 0(Ry)</td>
<td>load Y(i)</td>
</tr>
<tr>
<td>ADD.D F4,F2,F4</td>
<td>a*X(i) + Y(i)</td>
</tr>
<tr>
<td>S.D F4,0(Ry)</td>
<td>store into Y(i)</td>
</tr>
<tr>
<td>DADDIU Rx,Rx,#8</td>
<td>increment index to X</td>
</tr>
<tr>
<td>DADDIU Ry,Ry,#8</td>
<td>increment index to Y</td>
</tr>
<tr>
<td>DSUBU R20,R4,Rx</td>
<td>compute bound</td>
</tr>
<tr>
<td>BNEZ R20,loop</td>
<td>check if done</td>
</tr>
</tbody>
</table>

Assuming vectors X, Y are length 64:

Scalar vs. Vector:

- 578 (2+9*64) vs. 321 (1+5*64) ops (1.8X)
- 578 (2+9*64) vs. 6 instructions (96X)
- 64 operation vectors +
- no loop overhead
- also 64X fewer pipeline hazards
Vector Execution Time

- Time = f(vector length, data dependencies, struct. Hazards, C)
- **Initiation rate**: rate that FU consumes vector elements. (= number of lanes; usually 1 or 2 on Cray T-90)
- **Convoy**: set of vector instructions that can begin execution in same clock (no struct. or data hazards)
- **Chime**: approx. time for a vector element operation (~ one clock cycle).
- **m convoys take m chimes**: if each vector length is n, then they take approx. m x n clock cycles (ignores overhead; good approximiation for long vectors)

1: LV \( \mathbf{V1}, \mathbf{Rx} \); load vector X  
2: MULV \( \mathbf{V2}, \mathbf{F0}, \mathbf{V1} \); vector-scalar mult.  
   LV \( \mathbf{V3}, \mathbf{Ry} \); load vector Y  
3: ADDV \( \mathbf{V4}, \mathbf{V2}, \mathbf{V3} \); add  
4: SV \( \mathbf{Ry}, \mathbf{V4} \); store the result

4 conveys, 1 lane, VL=64  
=> 4 x 64 - 256 clocks (or 4 clocks per result)
Vector FU Start-up Time

- **Start-up time**: pipeline latency time (depth of FU pipeline); another sources of overhead
- Operation Start-up penalty (from CRAY-1)
  - Vector load/store 12
  - Vector multiply 7
  - Vector add 6

Assume convoys don't overlap; vector length = n:

<table>
<thead>
<tr>
<th>Convoy</th>
<th>Start</th>
<th>1st result</th>
<th>last result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. LV</td>
<td>0</td>
<td>12</td>
<td>11+n (12+n-1)</td>
</tr>
<tr>
<td>2. MULV, LV</td>
<td>12+n</td>
<td>12+n+12</td>
<td>23+2n</td>
</tr>
<tr>
<td>3. ADDV</td>
<td>24+2n</td>
<td>24+2n+6</td>
<td>29+3n</td>
</tr>
<tr>
<td>4. SV</td>
<td>30+3n</td>
<td>30+3n+12</td>
<td>41+4n</td>
</tr>
</tbody>
</table>
Vector Loop Processing

• Use vectors for inner loop parallelism (LLP):
  – One dimension of array: A[0, 0], A[0, 1], A[0, 2], ...
  – think of machine as, say, 32 vector regs each with 64 elements
  – 1 instruction updates 64 elements of 1 vector register

• and for outer loop parallelism:
  – 1 element from each column: A[0,0], A[1,0], A[2,0], ...
  – think of machine as 64 “virtual processors” (VPs)
    each with 32 scalar registers (- multithreaded processor)
  – 1 instruction updates 1 scalar register in 64 VPs

• Hardware identical, just 2 compiler perspectives
Virtual Processor Vector Model

- Vector operations are SIMD (single instruction multiple data) operations.
- Each element is computed by a virtual processor (VP)
- Number of VPs given by vector length:
  - vector control register.
Vector Architectural State

General Purpose Registers
- \( vr_0 \)
- \( vr_1 \)
- \( vr_{31} \)

Flag Registers (32)
- \( vf_0 \)
- \( vf_1 \)
- \( vf_{31} \)

Virtual Processors ($vlr$)
- \( VP_0 \)
- \( VP_1 \)
- \( VP_{vlr-1} \)

Control Registers
- \( vcr_0 \)
- \( vcr_1 \)
- \( vcr_{31} \)

$vdw$ bits: 1 bit
32 bits
Vector Load/Store Units & Memories

- Start-up overheads usually longer for LSUs
- Memory system must sustain (# lanes x word) /clock cycle
- Many Vector Procs. use banks (vs. simple interleaving):
  1) support multiple loads/stores per cycle
     => multiple banks & address banks independently
  2) support non-sequential accesses (see soon)
- Note: No. memory banks > memory latency to avoid stalls
  - $m$ banks => $m$ words per memory lantecy $l$ clocks
  - if $m < l$, then gap in memory pipeline:
    clock: 0 ... \(l\) \(l+1\) \(l+2\) ... \(l+m-1\) \(l+m\) ... \(2l\)
    word: -- ... 0 \(1\) \(2\) ... \(m-1\) -- ... \(m\)
  - may have 1024 banks in SRAM
Vector Memory Requirements Example

- The Cray T90 has a CPU clock cycle of 2.167 ns and in its largest configuration (Cray T932) has 32 processors each capable of generating four loads and two stores per CPU clock cycle.
- The CPU clock cycle is 2.167 ns, while the cycle time of the SRAMs used in the memory system is 15 ns.
- Calculate the minimum number of memory banks required to allow all CPUs to run at full memory bandwidth.
- **Answer:**
  - The maximum number of memory references each cycle is 192 (32 CPUs times 6 references per CPU).
  - Each SRAM bank is busy for \( \frac{15}{2.167} = 6.92 \) clock cycles, which we round up to 7 CPU clock cycles. Therefore we require a minimum of \( 192 \times 7 = 1344 \) memory banks!
  - The Cray T932 actually has 1024 memory banks, and so the early models could not sustain full bandwidth to all CPUs simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.
Vector Memory Access Example

- Suppose we want to fetch a vector of 64 elements starting at byte address 136, and a memory access takes 6 clocks. How many memory banks must we have to support one fetch per clock cycle? With what addresses are the banks accessed?
- When will the various elements arrive at the CPU?
- Answer
- Six clocks per access require at least six banks, but because we want the number of banks to be a power of two, we choose to have eight banks as shown on next slide
## Vector Memory Access Example

<table>
<thead>
<tr>
<th>Cycle no.</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
<th>Bank 4</th>
<th>Bank 5</th>
<th>Bank 6</th>
<th>Bank 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>136</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>busy</td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>busy</td>
<td>busy</td>
<td>152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>168</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>176</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>192</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
<tr>
<td>8</td>
<td>busy</td>
<td>200</td>
<td></td>
<td></td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
<tr>
<td>9</td>
<td>busy</td>
<td>busy</td>
<td>208</td>
<td></td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
<tr>
<td>10</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>216</td>
<td></td>
<td>busy</td>
<td>busy</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>224</td>
<td></td>
<td>busy</td>
</tr>
<tr>
<td>12</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>240</td>
</tr>
<tr>
<td>14</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>248</td>
</tr>
<tr>
<td>15</td>
<td>256</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
<tr>
<td>16</td>
<td>busy</td>
<td>264</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
</tbody>
</table>
Vector Length

• What to do when vector length is not exactly 64?
• *vector-length register* (VLR) controls the length of any vector operation, including a vector load or store. (cannot be > the length of vector registers)

\[
\text{do 10 i = 1, n} \\
10 \ Y(i) = a \times X(i) + Y(i)
\]

• Don't know n until runtime!
  n > Max. Vector Length (MVL)?
Strip Mining

• Suppose Vector Length > Max. Vector Length (MVL)?

• **Strip mining**: generation of code such that each vector operation is done for a size ŝ to the MVL

• 1st loop do short piece (n mod MVL), reset VL = MVL

  low = 1
  VL = (n mod MVL) /*find the odd size piece*/
  do 1 j = 0,(n / MVL) /*outer loop*/
      do 10 i = low,low+VL-1 /*runs for length VL*/
          Y(i) = a*X(i) + Y(i) /*main operation*/
  10 continue
  low = low+VL /*start of next vector*/
  VL = MVL /*reset the length to max*/
  1 continue

Time for loop:

\[ T_n = \left( \frac{n}{MVL} \right) \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} \]
Strip Mining Example

- What is the execution time on VMIPS for the vector operation $A = B \times s$, where $s$ is a scalar and the length of the vectors $A$ and $B$ is 200 (MVL supported =64)?

**Answer**

- Assume the addresses of $A$ and $B$ are initially in $Ra$ and $Rb$, $s$ is in $Fs$, and recall that for MIPS (and VMIPS) $R0$ always holds 0.
- Since $(200 \mod 64) = 8$, the first iteration of the strip-mined loop will execute for a vector length of 8 elements, and the following iterations will execute for a vector length of 64 elements.
- The starting byte addresses of the next segment of each vector is eight times the vector length. Since the vector length is either 8 or 64, we increment the address registers by $8 \times 8 = 64$ after the first segment and $8 \times 64 = 512$ for later segments.
- The total number of bytes in the vector is $8 \times 200 = 1600$, and we test for completion by comparing the address of the next vector segment to the initial address plus 1600.
- Here is the actual code follows:
Strip Mining Example

DADDUI R2,R0,#1600 ;total # bytes in vector
DADDU R2,R2,Ra ;address of the end of A vector
DADDUI R1,R0,#8 ;loads length of 1st segment
MTC1 VLR,R1 ;load vector length in VLR
DADDUI R1,R0,#64 ;length in bytes of 1st segment
DADDUI R3,R0,#64 ;vector length of other segments

Loop:
LV V1,Rb ;load B
MULVS.D V2,V1,Fs ;vector * scalar
SV Ra,V2 ;store A
DADDU Ra,Ra,R1 ;address of next segment of A
DADDU Rb,Rb,R1 ;address of next segment of B
DADDUI R1,R0,#512 ;load byte offset next segment
MTC1 VLR,R3 ;set length to 64 elements
DSUBU R4,R2,Ra ;at the end of A?
BNEZ R4,Loop ;if not, go back
Strip Mining Example

The three vector instructions in the loop are dependent and must go into three convoys, hence $T_{chime} = 3$. Let’s use our basic formula:

$$T_n = \left[ \frac{n}{MVL} \right] \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

The value of $T_{start}$ is the sum of

- The vector load start-up of 12 clock cycles
- A 7-clock-cycle start-up for the multiply
- A 12-clock-cycle start-up for the store

Thus, the value of $T_{start}$ is given by

$$T_{start} = 12 + 7 + 12 = 31$$

So, the overall value becomes

$$T_{200} = 660 + 4 \times 31 = 784$$

The execution time per element with all start-up costs is then $784/200 = 3.9$, compared with a chime approximation of three. In Section G.4, we will be more
Strip Mining Example

The total execution time per element and the total overhead time per element versus the vector length for the strip mining example.

MVL supported = 64
Vector Stride

• Suppose adjacent elements not sequential in memory
  do 10 i = 1,100
      do 10 j = 1,100
          A(i,j) = 0.0
      do 10 k = 1,100
  10    A(i,j) = A(i,j)+B(i,k)*C(k,j)

• Either B or C accesses not adjacent (800 bytes between)

• **stride**: distance separating elements that are to be merged into a single vector (caches do **unit stride**)
  => **LVWS** (load vector with stride) instruction
  => **SVWS** (store vector with stride) instruction

• Strides => can cause bank conflicts and a stall will occur if:
Vector Stride Memory Access Example

• Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

Answer

• Since the number of banks is larger than the bank busy time, for a stride of 1, the load will take \(12 + 64 = 76\) clock cycles, or 1.2 clocks per element.

• The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks.

• Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time.

• The total time will be \(12 + 1 + 6 \times 63 = 391\) clock cycles, or 6.1 clocks per element.
## Compiler Vectorization on Cray XMP

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% FP</th>
<th>% FP in vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM</td>
<td>23%</td>
<td>68%</td>
</tr>
<tr>
<td>DYFESM</td>
<td>26%</td>
<td>95%</td>
</tr>
<tr>
<td>FLO52</td>
<td>41%</td>
<td>100%</td>
</tr>
<tr>
<td>MDG</td>
<td>28%</td>
<td>27%</td>
</tr>
<tr>
<td>MG3D</td>
<td>31%</td>
<td>86%</td>
</tr>
<tr>
<td>OCEAN</td>
<td>28%</td>
<td>58%</td>
</tr>
<tr>
<td>QCD</td>
<td>14%</td>
<td>1%</td>
</tr>
<tr>
<td>SPICE</td>
<td>16%</td>
<td>7%</td>
</tr>
<tr>
<td>TRACK</td>
<td>9%</td>
<td>23%</td>
</tr>
<tr>
<td>TRFD</td>
<td>22%</td>
<td>10%</td>
</tr>
</tbody>
</table>

(1% overall)
Vector Chaining

• Suppose:
  MULV.D \textbf{V1},V2,V3
  ADDV.D \textbf{V4},\textbf{V1},V5 ; separate convoys?

• \textit{chaining}: vector register (V1) is not treated as a single entity but as a group of individual registers, then \textbf{pipeline forwarding} can work on individual elements of a vector

• \textit{Flexible chaining}: allow vector to chain to any other active vector operation => more read/write ports

• As long as enough HW is available, increases convoy size

• The above sequence is treated as a single convoy and the total running time becomes:
  \text{Vector length} + \text{Start-up time}_{\text{ADDV}} + \text{Start-up time}_{\text{MULV}}
Vector Chaining Example

- Timings for a sequence of dependent vector operations
  
  MULV.D  **V1**, V2, V3
  ADDV.D  V4, **V1**, V5
  
  both unchained and chained.

Unchained:

- MULV: 7, 64, 6, 64
- ADDV: 64
- Total = 141

Chained:

- MULV: 7, 64
- ADDV: 6, 64
- Total = 77
Example Chained Execution of Vector Code

Scalar CPU

Vector Memory Pipeline (VMP)
Vector Multiply Pipeline (VP0)
Vector Adder Pipeline (VP1)

lhai.v
hmul.vv
sadd.vv
addu
lhai.v
hmul.vv
sadd.vv
addu

Operations
Instruction issue

8 lanes, vector length 32, chaining
Vector Conditional Execution

• Suppose:
  
  ```
  do 100 i = 1, 64
    if (A(i) .ne. 0) then
      A(i) = A(i) - B(i)
    endif
  100 continue
  ```

• *vector-mask control* takes a Boolean vector: when *vector-mask (VM) register* is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.

• Still requires clock even if result not stored; if still performs operation, what about divide by 0?
**Vector Conditional Execution Example**

```plaintext
do 100 i = 1, 64
  if (A(i).ne. 0) then
    A(i) = A(i) - B(i)
  endif
  100 continue

LV V1,Ra ; load vector A into V1
LV V2,Rb ; load vector B
L.D F0,#0 ; load FP zero into F0
SNEVS.D V1,F0 ; sets VM(i) to 1 if V1(i)!-F0
SUBV.D V1,V1,V2 ; subtract under vector mask
CVM ; set the vector mask to all 1s
SV Ra,V1 ; store the result in A
```
Vector operations: Gather, Scatter

- Suppose:
  
  ```
  do 100 i = 1, n
  100 A(K(i)) = A(K(i)) + C(M(i))
  ```

- **gather** (LVI) operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector => *a nonsparse vector in a vector register*

- After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a **scatter** store (SVI), using the same index vector

- Can't be done by compiler since can't know Ki elements distinct, no dependencies; by compiler directive

- Use CVI to create index 0, 1xm, 2xm, ..., 63xm
Gather, Scatter Example

\[
\text{do } \quad 100 \quad i = 1, n \\
100 \quad A(K(i)) = A(K(i)) + C(M(i))
\]

Assuming that Ra, Rc, Rk, and Rm contain the starting addresses of the vectors in the previous sequence, the inner loop of the sequence can be coded with vector instructions such as:

- **LV** \(V_k, R_k\) \; load K
- **LVI** \(V_a, (Ra+V_k)\) \; load \(A(K(I))\)
- **LV** \(V_m, R_m\) \; load M
- **LVI** \(V_c, (Rc+V_m)\) \; load \(C(M(I))\)
- **ADDV.D** \(V_a, V_a, V_c\) \; add them
- **SVI** \((Ra+V_k), V_a\) \; store \(A(K(I))\)
Vector Conditional Execution Using Gather, Scatter

- The indexed loads-stores and the create an index vector CVI instruction provide an alternative method to support conditional vector execution.

```plaintext
do 100 i = 1, 64
    if (A(i).ne. 0) then
        A(i) = A(i) - B(i)
    endif

100   continue
```

```
LV V1,Ra ; load vector A into V1
L.D F0,#0 ; load FP zero into F0
SNEVS.D V1,F0 ; sets the VM to 1 if V1(i)!=F0
CVI V2,#8 ; generates indices in V2
POP R1,VM ; find the number of 1's in VM
MTC1 VLR,R1 ; load vector-length register
CVM ; clears the mask
LVI V3,(Ra+V2) ; load the nonzero A elements
LVI V4,(Rb+V2) ; load corresponding B elements
SUBV.D V3, V3, V4 ; do the subtract
SVI (Ra+V2), V3 ; store A back
```
/* Multiply $a[m][k] * b[k][n]$ to get $c[m][n]$ */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<k; t++)
        {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
Straightforward Solution

• Must sum of all the elements of a vector besides grabbing one element at a time from a vector register and putting it in the scalar unit?
  • e.g., shift all elements left 32 elements or collapse into a compact vector all elements not masked
  • In T0, the vector extract instruction, vext.v. This shifts elements within a vector
  • Called a “reduction”
Vector Matrix Multiply Solution

- You don't need to do reductions for matrix multiply
- You can calculate multiple independent sums within one vector register
- You can vectorize the j loop to perform 32 dot-products at the same time
- Or you can think of each 32 Virtual Processor doing one of the dot products
- (Assume Maximum Vector Length is 32)
- Shown in C source code, but can imagine the assembly vector instructions from it
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j+=32)/* Step j 32 at a time. */
    {
        sum[0:31] = 0; /* Initialize a vector register to zeros. */
        for (t=1; t<k; t++)
        {
            a_scalar = a[i][t]; /* Get scalar from a matrix. */
            b_vector[0:31] = b[t][j:j+31]; /* Get vector from b matrix. */
            prod[0:31] = b_vector[0:31]*a_scalar; /* Do a vector–scalar multiply. */
            /* Vector–vector add into results. */
            sum[0:31] += prod[0:31];
        }
        /* Unit-stride store of vector of results. */
        c[i][j:j+31] = sum[0:31];
    }
}  

Optimized Vector Solution
Common Vector Metrics

• **$R_\infty$:** MFLOPS rate on an infinite-length vector
  – vector “speed of light” or peak vector performance.
  – Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
  – ($R_n$ is the MFLOPS rate for a vector of length n)

• **$N_{1/2}$**: The vector length needed to reach one-half of $R_\infty$
  – a good measure of the impact of start-up

• **$N_V$:** The vector length needed to make vector mode faster than scalar mode
  – measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit
The Peak Performance $R_\infty$ of VMIPS on DAXPY

$$T_{\text{start}} = 12 + 7 + 12 + 6 + 12 = 49$$

Using $MVL = 64$, $T_{\text{loop}} = 15$, $T_{\text{start}} = 49$, and $T_{\text{chime}} = 3$ in the performance equation, and assuming that $n$ is not an exact multiple of 64, the time for an $n$-element operation is

$$T_n = \left\lfloor \frac{n}{64} \right\rfloor \times (15 + 49) + 3n \leq (n + 64) + 3n = 4n + 64$$

The sustained rate is actually over 4 clock cycles per iteration, rather than the theoretical rate of 3 chimes, which ignores overhead. The major part of the difference is the cost of the start-up overhead for each block of 64 elements (49 cycles versus 15 for the loop overhead).

We can now compute $R_\infty$ for a 500 MHz clock as

$$R_\infty = \lim_{n \to \infty} \left( \frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

The numerator is independent of $n$, hence

$$R_\infty = \frac{\text{Operations per iteration} \times \text{Clock rate}}{\lim_{n \to \infty} \left( \text{Clock cycles per iteration} \right)}$$

$$\lim_{n \to \infty} \left( \text{Clock cycles per iteration} \right) = \lim_{n \to \infty} \left( \frac{T_n}{n} \right) = \lim_{n \to \infty} \left( \frac{4n + 64}{n} \right) = 4$$

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{4} = 250 \text{ MFLOPS}$$
Sustained Performance of VMIPS on the Linpack Benchmark

The Linpack benchmark is a Gaussian elimination on a $100 \times 100$ matrix. Thus, the vector element lengths range from 99 down to 1. A vector of length $k$ is used $k$ times. Thus, the average vector length is given by

$$\sum_{i=1}^{99} \frac{i^2}{99} = 66.3$$

$$\sum_{i=1}^{99} i$$

Now we can obtain an accurate estimate of the performance of DAXPY using a vector length of 66.

$$T_{66} = 2 \times (15 + 49) + 66 \times 3 = 128 + 198 = 326$$

$$R_{66} = \frac{2 \times 66 \times 500}{326} \text{ MFLOPS} = 202 \text{ MFLOPS}$$
VMIPS DAXPY $N_{1/2}$

**Example**  What is $N_{1/2}$ for just the inner loop of DAXPY for VMIPS with a 500 MHz clock?

**Answer**  Using $R_\infty$ as the peak rate, we want to know the vector length that will achieve about 125 MFLOPS. We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

$$MFLOPS = \frac{\text{FLOPS executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{Second}} \times 10^{-6}$$

$$125 = \frac{2 \times N_{1/2}}{T_n} \times 500$$

Simplifying this and then assuming $N_{1/2} \leq 64$, so that $T_{n \leq 64} = 1 \times 64 + 3 \times n$, yields

$$T_{N_{1/2}} = 8 \times N_{1/2}$$

$$1 \times 64 + 3 \times N_{1/2} = 8 \times N_{1/2}$$

$$5 \times N_{1/2} = 64$$

$$N_{1/2} = 12.8$$

So $N_{1/2} = 13$; that is, a vector of length 13 gives approximately one-half the peak performance for the DAXPY loop on VMIPS.
VMIPS DAXPY $N_v$

Example  What is the vector length, $N_v$, such that the vector operation runs faster than the scalar?

Answer  Again, we know that $N_v < 64$. The time to do one iteration in scalar mode can be estimated as $10 + 12 + 12 + 7 + 6 + 12 = 59$ clocks, where 10 is the estimate of the loop overhead, known to be somewhat less than the strip-mining loop overhead. In the last problem, we showed that this vector loop runs in vector mode in time $T_{n \leq 64} = 64 + 3 \times n$ clock cycles. Therefore,

$$64 + 3N_v = 59N_v$$

$$N_v = \left\lceil \frac{64}{56} \right\rceil$$

$$N_v = 2$$

For the DAXPY loop, vector mode is faster than scalar as long as the vector has at least two elements. This number is surprisingly small,
DAXPY Performance on an Enhanced VMIPS

DAXPY, like many vector problems, is memory limited. Consequently, performance could be improved by adding more memory access pipelines. This is the major architectural difference between the Cray X-MP (and later processors) and the Cray-1. The Cray X-MP has three memory pipelines, compared with the Cray-1’s single memory pipeline, and the X-MP has more flexible chaining. How does this affect performance?

Example

What would be the value of $T_{66}$ for DAXPY on VMIPS if we added two more memory pipelines?

Answer

With three memory pipelines all the instructions fit in one convoy and take one chime. The start-up overheads are the same, so

$$T_{66} = \left[ \frac{66}{64} \right] \times (T_{\text{loop}} + T_{\text{start}}) + 66 \times T_{\text{chime}}$$

$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

With three memory pipelines, we have reduced the clock cycle count for sustained performance from 326 to 194, a factor of 1.7. Note the effect of Amdahl’s Law: We improved the theoretical peak rate as measured by the number of chimes by a factor of 3, but only achieved an overall improvement of a factor of 1.7 in sustained performance.
Vector Applications

*Not Limited to scientific computing*

- Multimedia Processing (compress., graphics, audio synth, image proc.)
- Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)
- Lossy Compression (JPEG, MPEG video and audio)
- Lossless Compression (Zero removal, RLE, Differencing, LZW)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Speech and handwriting recognition
- Operating systems/Networking (*memcpy*, *memset*, parity, checksum)
- Databases (hash/join, data mining, image/video serving)
- Language run-time support (stdlib, garbage collection)
Vector for Multimedia?

• Intel MMX: 57 new 80x86 instructions (1st since 386)
  – similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC

• 3 data types: 8 8-bit, 4 16-bit, 2 32-bit in 64bits
  – reuse 8 FP registers (FP and MMX cannot mix)

• Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  – use in drivers or added to library routines; no compiler
MMX Instructions

• Move 32b, 64b
• Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
  – opt. signed/unsigned saturate (set to max) if overflow
• Shifts (sll, srl, sra), And, And Not, Or, Xor
  in parallel: 8 8b, 4 16b, 2 32b
• Multiply, Multiply-Add in parallel: 4 16b
• Compare = , > in parallel: 8 8b, 4 16b, 2 32b
  – sets field to 0s (false) or 1s (true); removes branches
• Pack/Unpack
  – Convert 32b<--> 16b, 16b <--> 8b
  – Pack saturates (set to max) if number is too large
Vectors and Variable Data Width

• Programmer thinks in terms of vectors of data of some width (8, 16, 32, or 64 bits)
• Good for multimedia; More elegant than MMX-style extensions
• Don’t have to worry about how data stored in hardware
  – No need for explicit pack/unpack operations
• Just think of more virtual processors operating on narrow data
• Expand Maximum Vector Length with decreasing data width:
  64 x 64 bit, 128 x 32 bit, 256 x 16 bit, 512 x 8 bit
## Mediaprocesing: Vectorizable? Vector Lengths?

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Vector length</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Matrix transpose/multiply</td>
<td># vertices at once</td>
</tr>
<tr>
<td>• DCT (video, communication)</td>
<td>image width</td>
</tr>
<tr>
<td>• FFT (audio)</td>
<td>256-1024</td>
</tr>
<tr>
<td>• Motion estimation (video)</td>
<td>image width, iw/16</td>
</tr>
<tr>
<td>• Gamma correction (video)</td>
<td>image width</td>
</tr>
<tr>
<td>• Haar transform (media mining)</td>
<td>image width</td>
</tr>
<tr>
<td>• Median filter (image processing)</td>
<td>image width</td>
</tr>
<tr>
<td>• Separable convolution (img. proc.)</td>
<td>image width</td>
</tr>
</tbody>
</table>

Vector Pitfalls

- Pitfall: Concentrating on peak performance and ignoring start-up overhead: $N_V$ (length faster than scalar) > 100!
- Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)
  - failure of Cray competitor from his former company
- Pitfall: Good processor vector performance without providing good memory bandwidth
  - MMX?
Vector Processing Advantages

• Easy to get **high performance**; N operations:
  – are independent
  – use same functional unit
  – access disjoint registers
  – access registers in same order as previous instructions
  – access contiguous memory words or known pattern
  – can exploit large memory bandwidth
  – hide memory latency (and any other latency)
• **Scalable** (get higher performance as more HW resources available)
• **Compact:** Describe N operations with 1 short instruction (v. VLIW)
• **Predictable** (real-time) performance vs. statistical performance (cache)
• **Multimedia** ready: choose N * 64b, 2N * 32b, 4N * 16b, 8N * 8b
• Mature, developed **compiler technology**
• **Vector Disadvantage:** Out of Fashion
Intelligent RAM (IRAM)

Microprocessor & DRAM on a single chip:
- on-chip memory latency 5-10X, bandwidth 50-100X
- improve energy efficiency 2X-4X (no off-chip bus)
- serial I/O 5-10X v. buses
- smaller board area/volume
- adjustable memory size/width
Potential IRAM Latency: 5 - 10X

• No parallel DRAMs, memory controller, bus to turn around, SIMM module, pins…
• New focus: Latency oriented DRAM?
  – Dominant delay = RC of the word lines
  – keep wire length short & block sizes small?
• 10-30 ns for 64b-256b IRAM “RAS/CAS”?
• AlphaSta. 600: 180 ns=128b, 270 ns= 512b
  Next generation (21264): 180 ns for 512b?
Potential IRAM Bandwidth: 100X

- 1024 1Mbit modules (1Gb), each 256b wide
  - 20% @ 20 ns RAS/CAS = 320 GBytes/sec
- If cross bar switch delivers 1/3 to 2/3 of BW of 20% of modules
  ⇒ 100 - 200 GBytes/sec
- FYI: AlphaServer 8400 = 1.2 GBytes/sec
  - 75 MHz, 256-bit memory bus, 4 banks
Characterizing IRAM
Cost/Performance

- Cost - embedded processor + memory
- Small memory on-chip (25 - 100 MB)
- High vector performance (2 - 16 GFLOPS)
- High multimedia performance (4 - 64 GOPS)
- Low latency main memory (15 - 30ns)
- High BW main memory (50 - 200 GB/sec)
- High BW I/O (0.5 - 2 GB/sec via N serial lines)
  - Integrated CPU/cache/memory with high memory BW ideal for fast serial I/O
Vector IRAM Architecture

Maximum Vector Length (mvl) = # elts per register

- Maximum vector length is given by a read-only register mvl
  - E.g., in VIRAM-1 implementation, each register holds 32 64-bit values
- Vector Length is given by the register vl
  - This is the # of “active” elements or “virtual processors”

To handle variable-width data (8,16,32,64-bit):
- Width of each VP given by the register vpw
  - vpw is one of {8b,16b,32b,64b} (no 8b in VIRAM-1)
  - mvl depends on implementation and vpw: 32 64-bit, 64 32-bit, 128 16-bit,…
Vector IRAM Organization

Scalar Processor -> Queue

Net Int | L1 I cache | L1 D cache

Memory Interface Unit

Vector Registers

Load/Store

+ 

× 

÷
### V-IRAM1 Instruction Set

**Scalar**
- Standard scalar instruction set (e.g., ARM, MIPS)
  - Arithmetic operations: $+$, $-$, $\times$, $\div$, $\&$, $\mid$, $\text{shl}$, $\text{shr}$
  - Data types: $s.\text{int}$, $u.\text{int}$, $s.\text{fp}$, $d.\text{fp}$
  - Sizes: 8, 16, 32, 64

**Vector ALU**
- Operations: $s.\text{int}$, $u.\text{int}$, $s.\text{fp}$, $d.\text{fp}$
- Sizes: 8, 16, 32, 64
- Saturate and Overflow
  - Saturate: $\text{vv}$, $\text{vs}$, $\text{sv}$
  - Overflow: masked/unmasked

**Vector Memory**
- Load, Store
  - Data types: $s.\text{int}$, $u.\text{int}$
  - Sizes: 8, 16, 32, 64
  - Constants: indexed, unindexed
- Sizes: 32 x 32 x 64b (or 32 x 64 x 32b or 32 x 128 x 16b)
  + 32 x 128 x 1b flag

**Vector Registers**
- Sizes: 32 x 32 x 64b (or 32 x 64 x 32b or 32 x 128 x 16b)
  + 32 x 128 x 1b flag

**Plus:** flag, convert, DSP, and transfer operations
Goal for Vector IRAM Generations

- **V-IRAM-1 (-2000)**
  - 256 Mbit generation (0.20)
  - Die size = 1.5X 256 Mb die
  - 1.5 - 2.0 v logic, 2-10 watts
  - 100 - 500 MHz
  - 4 64-bit pipes/lanes
  - 1-4 GFLOPS/6-16G (16b)
  - 30 - 50 GB/sec Mem. BW
  - 32 MB capacity + DRAM bus
  - Several fast serial I/O

- **V-IRAM-2 (-2003)**
  - 1 Gbit generation (0.13)
  - Die size = 1.5X 1 Gb die
  - 1.0 - 1.5 v logic, 2-10 watts
  - 200 - 1000 MHz
  - 8 64-bit pipes/lanes
  - 2-16 GFLOPS/24-64G
  - 100 - 200 GB/sec Mem. BW
  - 128 MB cap. + DRAM bus
  - Many fast serial I/O
VIRAM-1 Microarchitecture

- 2 arithmetic units
  - both execute integer operations
  - one executes FP operations
  - 4 64-bit datapaths (lanes) per unit
- 2 flag processing units
  - for conditional execution and speculation support
- 1 load-store unit
  - optimized for strides 1, 2, 3, and 4
  - 4 addresses/cycle for indexed and strided operations
  - decoupled indexed and strided stores

- Memory system
  - 8 DRAM banks
  - 256-bit synchronous interface
  - 1 sub-bank per bank
  - 16 Mbytes total capacity

- Peak performance
  - 3.2 GOPS\textsubscript{64}, 12.8 GOPS\textsubscript{16} (w. madd)
  - 1.6 GOPS\textsubscript{64}, 6.4 GOPS\textsubscript{16} (wo. madd)
  - 0.8 GFLOPS\textsubscript{64}, 1.6 GFLOPS\textsubscript{32}
  - 6.4 Gbyte/s memory bandwidth consumed by VU
Tentative VIRAM-1 Floorplan

- 0.18 µm DRAM
  32 MB in 16 banks x 256b, 128 subbanks
- 0.25 µm, 5 Metal Logic
- - 200 MHz MIPS, 16K I$, 16K D$
- - 4 200 MHz FP/int. vector units
- die: - 16x16 mm
- xtors: - 270M
- power: -2 Watts

Memory (128 Mbits / 16 MBytes)

4 Vector Pipes/Lanes

CPU +$
V-IRAM-2: 0.13 µm, Fast Logic, 1GHz
16 GFLOPS(64b)/64 GOPS(16b)/128MB

2-way Superscalar Processor

Vector Instruction Queue

Vector Registers

Load/Store

Memory Crossbar Switch

8K I cache

8K D cache

I/O

Instruction Queue

Queue

Superscalar Processor

8 x 64

or

16 x 32

or

32 x 16

8 x 64
0.13 µm, 1 Gbit DRAM

>1B Xtors: 98% Memory, Xbar, Vector ⇒ regular design

Spare Pipe & Memory ⇒ 90% die repairable

Short signal distance ⇒ speed scales <0.1 µm
Retarget of Cray compiler

Steps in compiler development
- Build MIPS backend (done)
- Build VIRAM backend for vectorized loops (done)
- Instruction scheduling for VIRAM-1 (done)
- Insertion of memory barriers (using Cray strategy, improving)
- Additional optimizations (ongoing)
- Feedback results to Cray, new version from Cray (ongoing)