Recap: VLIW

- In a classic VLIW, compiler is responsible for avoiding all hazards -> simple hardware, complex compiler. Later VLIWs added more dynamic hardware interlocks.
- Use loop unrolling and software pipelining for loops, trace scheduling for more irregular code.
- Static scheduling difficult in presence of unpredictable branches and variable latency memory.
- VLIWs somewhat successful in embedded computing, no clear success in general-purpose computing despite several attempts.
- Static scheduling compiler techniques also useful for superscalar processors.
Supercomputers

Definition of a supercomputer:
• Fastest machine in world at given task
• A device to turn a compute-bound problem into an I/O bound problem
• Any machine costing $30M+
• Any machine designed by Seymour Cray

CDC6600 (Cray, 1964) regarded as first supercomputer

Supercomputer Applications

Typical application areas
• Military research (nuclear weapons, cryptography)
• Scientific research
• Weather forecasting
• Oil exploration
• Industrial design (car crash simulation)
• Bioinformatics
• Cryptography

All involve huge computations on large data sets

*In 70s-80s, Supercomputer = Vector Machine*
Vector Supercomputers

Epitomized by Cray-1, 1976:

- Scalar Unit
  - Load/Store Architecture
- Vector Extension
  - Vector Registers
  - Vector Instructions
- Implementation
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

Cray-1 (1976)

Single Port Memory

- 16 banks of 64-bit words
- 8-bit SECDED
- 80MW/sec data load/store
- 320MW/sec instruction buffer refill

64 Element Vector Registers

- \((A_d) + j k m\)

64 T Regs

64 B Regs

V. Mask
V. Length
FP Add
FP Mul
FP Recip
Int Add
Int Logic
Int Shift
Pop Cnt

addr Add
addr Mul

memory bank cycle 50 ns  processor cycle 12.5 ns (80MHz)
Vector Programming Model

Scalar Registers

Vector Registers

Vector Length Register

Vector Arithmetic Instructions

ADDV v3, v1, v2

Vector Load and Store Instructions

LV v1, r1, r2

Vector Code Example

# Scalar Code
LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop

# Vector Code
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3

# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
Vector Instruction Set Advantages

-Compact
  - one short instruction encodes N operations
-Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory
    (unit-stride load/store)
  - access memory in a known pattern
    (strided load/store)
-Scalable
  - can run same code on more parallel pipelines (lanes)

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

$V_3 \leftarrow V_1 \times V_2$

Six stage multiply pipeline
Vector Instruction Execution

ADDV C,A,B

Execution using one pipelined functional unit

Execution using four pipelined functional units

Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- Bank busy time: Cycles between accesses to same bank
Vector Unit Structure

- Functional Unit
- Vector Registers
- Elements 0, 4, 8, ...
- Elements 1, 5, 9, ...
- Elements 2, 6, 10, ...
- Elements 3, 7, 11, ...

Memory Subsystem

T0 Vector Microprocessor (UCB/ICSI, 1995)

- Vector register elements striped over lanes
- Lane
**Vector Instruction Parallelism**

Can overlap execution of multiple vector instructions
- example machine has 32 elements per vector register and 8 lanes

Complete 24 operations/cycle while issuing 1 short instruction/cycle

**CS152 Administrivia**

- Wednesday section, 9:30-10:30AM, PS4 review
- Thursday quiz, covers L12-L15, PS4, Lab 4
Vector Chaining

- Vector version of register bypassing
  - introduced with Cray-1

\[
\begin{align*}
&\text{LV } v_1 \\
&\text{MULV } v_3, v_1, v_2 \\
&\text{ADDV } v_5, v_3, v_4
\end{align*}
\]

Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction

- With chaining, can start dependent instruction as soon as first result appears
**Vector Startup**

Two components of vector startup penalty

- functional unit latency (time through pipeline)
- dead time or recovery time (time before another vector instruction can start down pipeline)

---

**Dead Time and Short Vectors**

- **Cray C90, Two lanes**
  - 4 cycle dead time
  - Maximum efficiency 94% with 128 element vectors

- **T0, Eight lanes**
  - No dead time
  - 100% efficiency with 8 element vectors
Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 (’73) and TI ASC (’71), were memory-memory machines
- Cray-1 (’76) was first vector register machine

Example Source Code

```c
for (i=0; i<N; i++)
{
    C[i] = A[i] + B[i];
    D[i] = A[i] - B[i];
}
```

Vector Memory-Memory Code

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV C, A, B</td>
</tr>
<tr>
<td>SUBV D, A, B</td>
</tr>
</tbody>
</table>

Vector Register Code

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV V1, A</td>
</tr>
<tr>
<td>LV V2, B</td>
</tr>
<tr>
<td>ADDV V3, V1, V2</td>
</tr>
<tr>
<td>SV V3, C</td>
</tr>
<tr>
<td>SUBV V4, V1, V2</td>
</tr>
<tr>
<td>SV V4, D</td>
</tr>
</tbody>
</table>

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMA make difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMA incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2 elements

⇒ Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures

(we ignore vector memory-memory from now on)
Automatic Code Vectorization

Scalar Sequential Code

Iter. 1
load
load
add
store

Iter. 2
load
load
add
store

Vectorized Code

load
load
add
store

load
load
add
store

Vectorization is a massive compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length
Solution: Break loops into pieces that fit in registers, “Stripmining”

for (i=0; i<N; i++)
C[i] = A[i] + B[i];

ANDI R1, N, 63  # N mod 64
MTCL VLR, R1    # Do remainder

for (i=0; i<N; i++)

loop:

C[i] = A[i] + B[i];

DSLL R2, R1, 3  # Multiply by 8
DADDU RA, RA, R2 # Bump pointer
LV V1, RA
LV V2, RB
DADDU RB, RB, R2
ADDV.D V3, V1, V2
SV V3, RC
DADDU RC, RC, R2
DSUBU N, N, R1  # Subtract elements
LI R1, 64
MTCL VLR, R1    # Reset full length
BGTZ N, loop    # Any more to do?
Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

\[
\text{for } (i=0; i<N; i++) \\
A[i] = B[i] + C[D[i]]
\]

Indexed load instruction (\textit{Gather})

\[
\begin{align*}
&\text{LV } vD, rD \quad \# \text{ Load indices in } D \text{ vector} \\
&\text{LVI } vC, rC, vD \quad \# \text{ Load indirect from } rC \text{ base} \\
&\text{LV } vB, rB \quad \# \text{ Load } B \text{ vector} \\
&\text{ADDV.D } vA, vB, vC \quad \# \text{ Do add} \\
&\text{SV } vA, rA \quad \# \text{ Store result}
\end{align*}
\]

Vector Scatter/Gather

Scatter example:

\[
\text{for } (i=0; i<N; i++) \\
A[B[i]]++;
\]

Is following a correct translation?

\[
\begin{align*}
&\text{LV } vB, rB \quad \# \text{ Load indices in } B \text{ vector} \\
&\text{LVI } vA, rA, vB \quad \# \text{ Gather initial } A \text{ values} \\
&\text{ADDV } vA, vA, 1 \quad \# \text{ Increment} \\
&\text{SVI } vA, rA, vB \quad \# \text{ Scatter incremented values}
\end{align*}
\]
Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```c
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers
- vector version of predicate registers, 1 bit per element

... and *maskable* vector instructions
- vector operation becomes NOP at elements where mask bit is clear

Code example:

```c
CVM             # Turn on all elements
LV vA, rA      # Load entire A vector
SGTVS.D vA, F0 # Set bits in mask register where A>0
LV vA, rB      # Load B vector into A under mask
SV vA, rA      # Store A back to memory under mask
```

---

Masked Vector Instructions

Simple Implementation – execute all N operations, turn off result
writeback according to mask

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M[0]</td>
<td>A[0]</td>
<td>B[0]</td>
</tr>
</tbody>
</table>

Write Enable

Density-Time Implementation – scan mask vector and only execute
elements with non-zero masks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M[0]</td>
<td>A[0]</td>
<td>B[0]</td>
</tr>
</tbody>
</table>

Write data port

---

4/8/2008 27

CS152-Spring’08
Compress/Expand Operations

- Compress packs non-masked elements from one vector register contiguously at start of destination vector register
  - Population count of mask vector gives packed vector length
- Expand performs inverse operation

![Diagram showing Compress and Expand operations]

Used for density-time conditionals and also for general selection operations

Vector Reductions

Problem: Loop-carried dependence on reduction variables

\[
\text{sum} = 0; \\
\text{for } (i=0; i<N; i++) \\
\text{sum} += A[i]; \quad \# \text{Loop-carried dependence on sum}
\]

Solution: Re-associate operations if possible, use binary tree to perform reduction

\[
\text{# Rearrange as:} \\
\text{sum[0:VL-1] = 0} \quad \# \text{Vector of VL partial sums} \\
\text{for}(i=0; i<N; i+=\text{VL}) \quad \# \text{Stripmine VL-sized chunks} \\
\text{sum[0:VL-1] += A[i:i+VL-1];} \quad \# \text{Vector sum} \\
\text{# Now have VL partial sums in one vector register} \\
\text{do} \\
\quad \text{VL = VL/2;} \quad \# \text{Halve vector length} \\
\quad \text{sum[0:VL-1] += sum[VL:2*VL-1]} \quad \# \text{Halve no. of partials} \\
\text{while (VL>1)}
\]

- 65nm CMOS technology
- Vector unit (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
  - 1 load or store unit (8 x 8-byte accesses/cycle)
- Scalar unit (1.6 GHz)
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB l-cache and 64KB data cache

Memory system provides 256GB/s DRAM bandwidth per CPU
Up to 16 CPUs and up to 1TB DRAM form shared-memory node
  - total of 4TB/s bandwidth to shared DRAM memory
Up to 512 nodes connected via 128GB/s network links (message passing between nodes)

(Multimedia Extensions (aka SIMD extensions)

<table>
<thead>
<tr>
<th></th>
<th>64b</th>
<th>32b</th>
<th>32b</th>
</tr>
</thead>
<tbody>
<tr>
<td>16b</td>
<td>16b</td>
<td>16b</td>
<td>16b</td>
</tr>
<tr>
<td>8b</td>
<td>8b</td>
<td>8b</td>
<td>8b</td>
</tr>
</tbody>
</table>

- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - This concept first used on Lincoln Labs TX-2 computer in 1957, with 36b datapath split into 2x18b or 4x9b
  - Newer designs have 128-bit registers (PowerPC Altivec, Intel SSE2/3/4)
- Single instruction operates on all elements within register

(See also Cray X1E in Appendix F)
Multimedia Extensions versus Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary

- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure

- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252