Last Time in Lecture 8

Overcoming the worst hazards in OoO superscalars:

• Branch prediction
  • Bimodel
  • Local/Branch History Table
  • Global/gselect, gshare
  • Tournament
  • Branch address cache (predict multiple branches per cycle)
  • Trace cache
  • Return Address Predictors

• Today - Load/Store Queues, Vector Supercomputers
Load-Store Queue Design

- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by **allowing loads to be speculatively issued**
Speculative Store Buffer

- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into “store address” and “store data” micro-operations
- “Store address” execution writes tag
- “Store data” execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit
Load bypass from speculative store buffer

- If data in both store buffer and cache, which should we use? Speculative store buffer
- If same address in store buffer twice, which should we use? Youngest store older than load
Memory Dependencies

\[ \text{sd } x1, (x2) \]
\[ \text{ld } x3, (x4) \]

- When can we execute the load?
In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

- Need a structure to handle memory ordering...
Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and \( x4 \neq x2 \)
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don’t execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)
Address Speculation

$\text{sd } x1, \ (x2)\n\text{ld } x3, \ (x4)$

- Guess that $x4 \neq x2$
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find $x4 = x2$, squash load and all following instructions

=> Large penalty for inaccurate address speculation
Memory Dependence Prediction (Alpha 21264)

\[ \text{sd \ x1, (x2)} \]
\[ \text{ld \ x3, (x4)} \]

- Guess that \( x4 \neq x2 \) and execute load before store.

- If later find \( x4 = x2 \), squash load and all following instructions, but mark load instruction as store-wait.

- Subsequent executions of the same load instruction will wait for all previous stores to complete.

- Periodically clear store-wait bits.
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 set

- Supercomputers: CDC6600, CDC7600, Cray-1, ...

- 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 Research, 1976]
Cray-1 Internals displayed at EPFL

Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr
Vector Programming Model

Scalar Registers

Vector Registers

Vector Length Register

Vector Arithmetic Instructions

vadd v3, v1, v2

Vector Load and Store Instructions

vld v1, x1, x2

Base, x1

Stride, x2

Memory

© Krste Asanovic, 2015
## Vector Code Example

<table>
<thead>
<tr>
<th>C code</th>
<th>Scalar Code</th>
<th>Vector Code</th>
</tr>
</thead>
</table>
| for (i=0; i<64; i++)
  C[i] = A[i] + B[i]; | # Scalar Code
  loop:
    fld f1, 0(x1)
    fld f2, 0(x2)
    fadd.d f3,f1,f2
    fsd f3, 0(x3)
    addi x1, 8
    addi x2, 8
    addi x3, 8
    subi x4, 1
    bnez x4, loop | # Vector Code
  li x4, 64
  setvlr x4
  vfld v1, x1
  vfld v2, x2
  vfadd.d v3,v1,v2
  vfsd v3, x3 |
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
  - 16 banks of 64-bit words
  - 8-bit SECDED
  - 80MW/sec data load/store

- 320MW/sec instruction buffer refill

- 4 Instruction Buffers

- 64-bitx16

- 64 T Regs
  - T_{jk}
  - S_i

- 64 B Regs
  - B_{jk}
  - A_i

- 64 Element Vector
  - V0
  - V1
  - V2
  - V3
  - V4
  - V5
  - V6
  - V7
  - V_i
  - V_j
  - V_k

- 4 Instruction Buffers
  - NIP
  - LIP
  - CIP

memory bank cycle 50 ns  processor cycle 12.5 ns (80MHz)
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!)

\[
v_3 < - v_1 \times v_2
\]

Six-stage multiply pipeline
Vector Instruction Execution

vfadd.d vc, va, vb

Execution using one pipelined functional unit


C[0]
C[1]
C[2]

Execution using four pipelined functional units


A[22] B[22]

A[27] B[27]

C[0]
C[1]
C[2]
C[3]
Interleaved Vector Memory System

- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency
Vector Unit Structure

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

**Lane**
T0 Vector Microprocessor (UCB/ICSI, 1995)

Vector register elements striped over lanes
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
Vector Chaining

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

```
vld v1
vfmul v3, v1, v2
Vfadd v5, v3, v4
```
Vector Chaining Advantage

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

<table>
<thead>
<tr>
<th>Load</th>
<th>Mul</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time→</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• With chaining, can start dependent instruction as soon as first result appears

<table>
<thead>
<tr>
<th>Load</th>
<th>Mul</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time→</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

Functional Unit Latency

First Vector Instruction

Dead Time

Second Vector Instruction
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

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```
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
- VMMAs make it difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs 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 2-4 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**

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

**Scalar Sequential Code**

- Iter. 1:
  - Load
  - Add
  - Store

- Iter. 2:
  - Load
  - Add
  - Store

**Vectorized Code**

- Iter. 1:
  - Load
  - Add
  - Store

- Iter. 2:
  - 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”

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

```
+    +
A --- B --- C

64 elements

Remainder

+    +
A --- B

Remainder

A[0:N-64]
```

```plaintext
and i x1, xN, 63  # N mod 64
setvlr x1         # Do remainder

loop:
vld v1, xA
sll x2, x1, 3    # Multiply by 8
add xA, x2       # Bump pointer
vld v2, xB
add xB, x2
vfadd.d v3, v1, v2
vsd v3, xC
add xC, x2
sub xN, x1       # Subtract elements
li x1, 64
setvlr x1        # Reset full length
bgtz xN, loop    # Any more to do?
```
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 bubble (“NOP”) at elements where mask bit is clear

Code example:

```assembly
  cvm                 # Turn on all elements
  vld vA, xA         # Load entire A vector
  vfsgts.d vA, f0    # Set bits in mask register where A>0
  vld vA, xB         # Load B vector into A under mask
  vsd vA, xA         # Store A back to memory under mask
```
Masked Vector Instructions

Simple Implementation

- execute all N operations, turn off result writeback according to mask

\[
\begin{align*}
M[0] &= 0 & A[0] & B[0]
\end{align*}
\]

Write Enable → \(C[0]\)

Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks

\[
\begin{align*}
M[7] &= 1 \\
M[6] &= 0 \\
M[5] &= 1 \\
M[4] &= 1 \\
M[3] &= 0 \\
M[2] &= 0 \\
M[1] &= 1 \\
M[0] &= 0
\end{align*}
\]

Write data port

M[7]=1
M[6]=0
M[5]=1
M[4]=1
M[3]=0
M[2]=0
M[1]=1
M[0]=0

Write Enable → \(C[0]\) → \(C[1]\) → \(C[2]\) → \(C[3]\) → \(C[4]\) → \(C[5]\) → \(A[7]\) → \(B[7]\)

Write data port
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

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

**Problem:** Loop-carried dependence on reduction variables

```c
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  // Loop-carried dependence on sum
```

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

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

Want to vectorize loops with indirect accesses:
\[
\text{for } (i=0; i<N; i++) \\
\quad A[i] = B[i] + C[D[i]]
\]

Indexed load instruction (\textit{Gather})
\begin{align*}
\text{vld } &vD, xD &\# \text{ Load indices in } D \text{ vector} \\
\text{vdli } &vC, xC, vD &\# \text{ Load indirect from } rC \text{ base} \\
\text{vld } &vB, xB &\# \text{ Load } B \text{ vector} \\
\text{vfadd.d } &vA,vB,vC &\# \text{ Do add} \\
\text{vsd } &vA, xA &\# \text{ Store result}
\end{align*}
Vector Scatter/Gather

Histogram example:

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

Is following a correct translation?

```
vld vB, xB  # Load indices in B vector
vldi vA, xA, vB  # Gather initial A values
vadd vA, vA, 1  # Increment
vsdi vA, xA, vB  # Scatter incremented values
```
Vector Memory Models

- Most vector machines have a very relaxed memory model, e.g.
  \[\text{vsd v1, x1} \quad \# \text{ Store vector to x1}\]
  \[\text{vld v2, x1} \quad \# \text{ Load vector from x1}\]
  - No guarantee that elements of v2 will have value of elements of v1 even when store and load execute by same processor!

- Requires explicit memory barrier or fence
  \[\text{vsd v1, x1} \quad \# \text{ Store vector to x1}\]
  \[\text{fence.vs.vl} \quad \# \text{ Enforce ordering s->l}\]
  \[\text{vld v2, x1} \quad \# \text{ Load vector from x1}\]

Vector machines support highly parallel memory systems (multiple lanes and multiple load and store units) with long latency (100+ clock cycles)
  - hardware coherence checks would be prohibitively expensive
  - vectorizing compiler can eliminate most dependencies

- 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 I-cache and 64KB data cache

[ ©NEC ]

- 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)

[ New announcement SX-ACE, 4x16-lane vector CPUs on one chip]
Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)