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
Vector Programming Model

**Scalar Registers**
- r15
- r0

**Vector Registers**
- v15
- v0

### Vector Load and Store Instructions
- LV v1, r1, r2
- Base, r1
- Stride, r2
- Memory

### Vector Arithmetic Instructions
- ADDV v3, v1, v2
- v1
- v2
- v3

Vector Length Register: VLR

---

Vector Code Example

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

### # 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)
    DADDUI R1, 8
    DADDUI R2, 8
    DADDUI 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
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!)

\[ V_3 <- V_1 \times V_2 \]
Vector Instruction Execution

ADDV C,A,B

Execution using one pipelined functional unit


C[0]

Execution using four pipelined functional units

9/20/2007 11

Vector Memory System

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

- Bank busy time: Cycles between accesses to same bank

9/20/2007 12
Vector Unit Structure

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

2. **Functional Unit**

3. **Memory Subsystem**

---

T0 Vector Microprocessor (UCB/ICSI, 1995)

- **Vector register elements**
  - striped over lanes

---

9/20/2007
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

CS252 Administrivia

• Project proposal due one week from today, send via email (some problems with bspace server)
• Title, team members’ names, one page PDF writeup
• Send matchmaking email to class if you don’t have partner
• Krste office hours 1-3pm, Monday 645 Soda
In the news…

• Sep 18, 2007: Intel announces next generation Nehalem microarchitecture
  – Up to 8 cores in one socket (two quad-core die)
  – Each core runs two threads => 16 hardware threads in one socket

• Also, announces successful fabrication in 32nm technology
  – Moore’s Law to continue for another decade???
    » 45->32->22->16->11???

• AMD announces 3-core Phenom chip

Vector Chaining

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

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 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

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

```
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?
- VMMAs make it difficult to overlap execution of multiple vector operations, why?
- 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 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

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, load, add, store
- Iter. 2: 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];

loop:
    ANDI R1, N, 63    # N mod 64
    MTCL VLR, R1      # Do remainder
    LV V1, RA
    DSLL R2, R1, 3    # Multiply by 8
    DADDU RA, RA, R2  # Bump pointer
    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:
    for (i=0; i<N; i++)
        A[i] = B[i] + C[D[i]]

Indexed load instruction (Gather)
    LV vD, rD        # Load indices in D vector
    LVI vC, rC, vD   # Load indirect from rC base
    LV vB, rB        # Load B vector
    ADDV.D vA,vB,vC  # Do add
    SV vA, rA        # Store result
Vector Scatter/Gather

Scatter example:

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

Is following a correct translation?

```assembly
LV vB, rB # Load indices in B vector
LVI va, ra, vB # Gather initial A values
ADDV va, va, 1 # Increment
SVI va, ra, vB # Scatter incremented values
```

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:

```assembly
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

\[
\begin{align*}
M[1] &= 1 & C[1] & \downarrow \quad \text{Write Enable} \\
M[0] &= 0 & C[0] & \downarrow \quad \text{Write data port}
\end{align*}
\]

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

\[
\begin{align*}
M[1] &= 1 & C[1] & \uparrow \quad \text{Write data port}
\end{align*}
\]

9/20/2007

---

**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

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

**Compress**  **Expand**

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

9/20/2007
Vector Reductions

Problem: Loop-carried dependence on reduction variables

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

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

\[
\# \text{Rearrange as:} \\
\text{sum}[0:VL-1] = 0 \quad \# \text{Vector of VL partial sums} \\
\text{for (i=0; i<N; i+=VL)} \quad \# \text{Stripmine VL-sized chunks} \\
\quad \text{sum}[0:VL-1] += A[i:i+VL-1]; \quad \# \text{Vector sum} \\
\# \text{Now have VL partial sums in one vector register}
\]

\[
\text{do} \{ \\
\quad \text{VL} = \text{VL}/2; \quad \# \text{Halve vector length} \\
\quad \text{sum}[0:VL-1] += \text{sum}[VL:2*VL-1] \quad \# \text{Halve no. of partials} \\
\} \text{ while (VL>1)}
\]


- CMOS Technology
  - 1.1GHz CPU, 2.2GHz vector unit, on single chip
- Scalar unit
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache
- Vector unit
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
  - 8 lanes (16 FLOPS/cycle, 35.2 GFLOPS peak)
  - 1 load or store unit (8x8 byte accesses/cycle)
  - 70.4 GB/s memory bandwidth per processor
- SMP structure
  - 8 CPUs connected to memory through crossbar
  - 256 GB capacity/8-way node
  - 563 GB/s shared memory bandwidth (4096 interleaved banks)

(See also Cray X1E in Appendix F)
Multimedia Extensions

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (Altivec, SSE2/3)
- 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

Next Time

- Look at modern memory system design
- Discussion of VLIW versus Vector, pick a side and argue for that style of architecture