CS252
Graduate Computer Architecture
Lecture 12

Multithreading / Vector Processing
February 29th, 2012

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252

Review: Discussion of SPARCLE paper

- Example of close coupling between processor and memory controller (CMMU)
  - All of features mentioned in this paper implemented by combination of processor and memory controller
  - Some functions implemented as special "coprocessor" instructions
  - Others implemented as "Tagged" loads/stores/swaps
- Course Grained Multithreading
  - Using SPARC register windows
  - Automatic synchronous trap on cache miss
  - Fast handling of all other traps/interrupts (great for message interface!)
  - Multithreading half in hardware/half software (hence 14 cycles)
- Fine Grained Synchronization
  - Full-Empty bit/32 bit word (effectively 33 bits)
    - » Groups of 4 words/cache line ⇒ F/E bits put into memory TAG
  - Fast TRAP on bad condition
  - Multiple instructions. Examples:
    - » LDT (load/trap if empty)
    - » LDET (load/set empty/trap if empty)
    - » STF (Store/set full)
    - » STFT (store/set full/trap if full)

Review: Discussion of Papers: Sparcle (Con't)

- Message Interface
  - Closely couple with processor
    - » Interface at speed of first-level cache
  - Atomic message launch:
    - » Describe message (including DMA ops) with simple stio insts
    - » Atomic launch instruction (ipilaunch)
  - Message Reception
    - » Possible interrupt on message receive: use fast context switch
    - » Examine message with simple ldio instructions
    - » Discard in pieces, possibly with DMA
    - » Free message (ipicst, i.e "coherent storeback")
- We will talk about message interface in greater detail

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications
  - e.g., Database or Scientific codes
  - Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: instruction stream with own PC and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Thread Level Parallelism (TLP):
  - Exploit the parallelism inherent between threads to improve performance
- Data Level Parallelism (DLP):
  - Perform identical operations on data, and lots of data

One approach to exploiting threads: Multithreading (TLP within processor)

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks
- When switch?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
  - Usually done in a round-robin fashion, skipping any stalled threads
  - CPU must be able to switch threads every clock
- Advantage:
  - can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage:
  - slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's Niagra (recent), several research multiprocessors, Tera

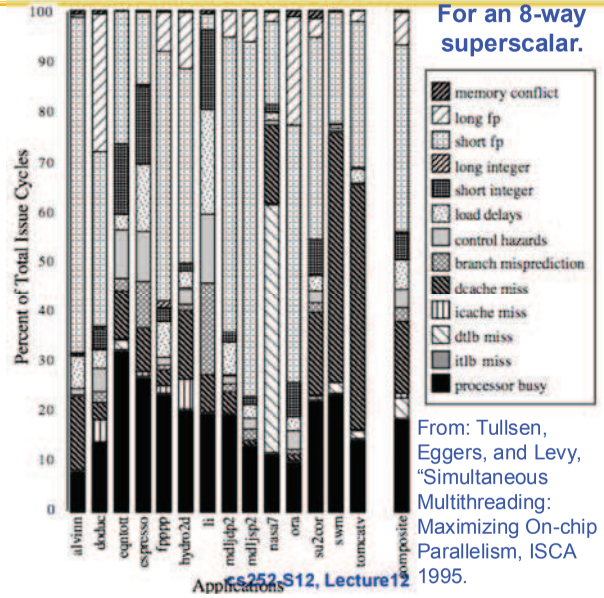
Course-Grained Multithreading

- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time
- Used in IBM AS/400, Sparcle (for Alewife)

Simultaneous Multithreading (SMT): Do both ILP and TLP

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

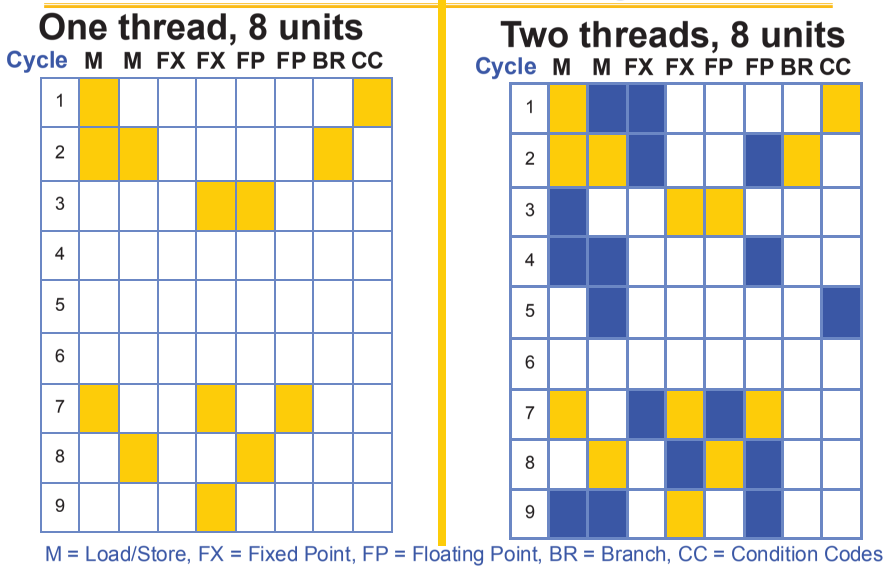
Justification: For most apps, most execution units lie idle



For an 8-way superscalar.

From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.

Simultaneous Multi-threading ...



M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Simultaneous Multithreading Details

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

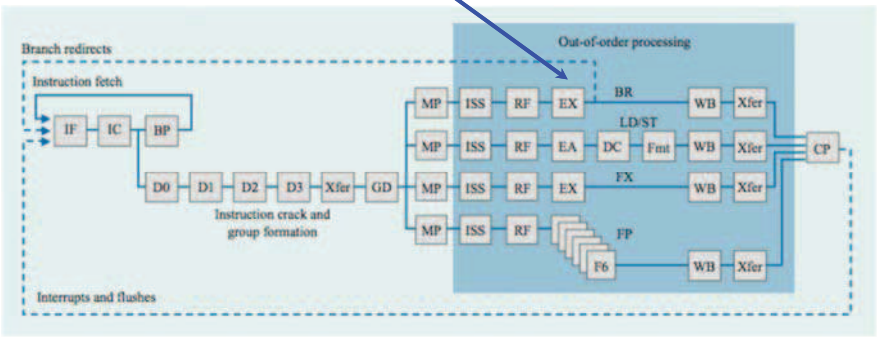
Source: Micrprocessor Report, December 6, 1999 "Compaq Chooses SMT for Alpha"

Design Challenges in SMT

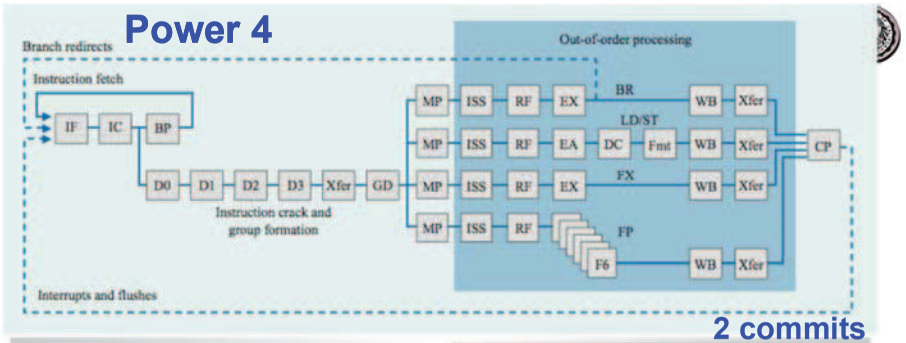
- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Clock cycle time, especially in:
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

Power 4

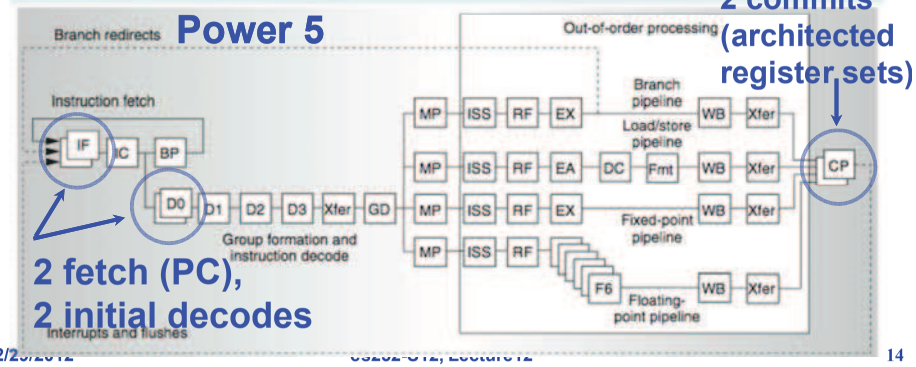
Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.



Power 4



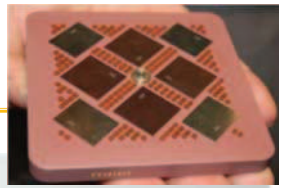
Power 5

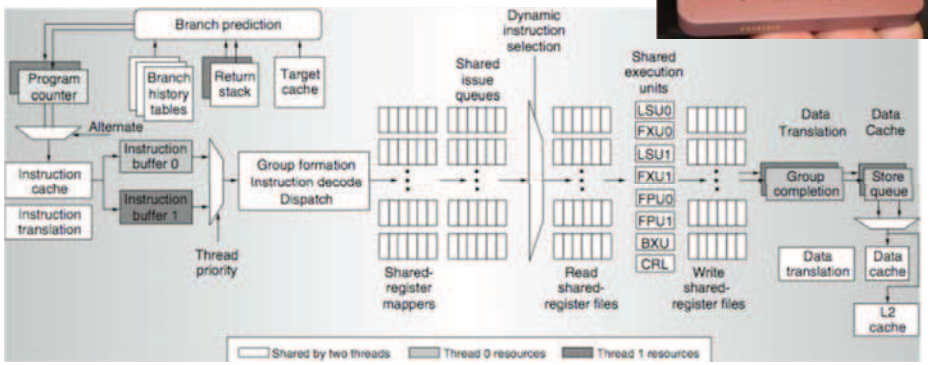


2 commits (architected register sets)

2 fetch (PC), 2 initial decodes

Power 5 data flow ...



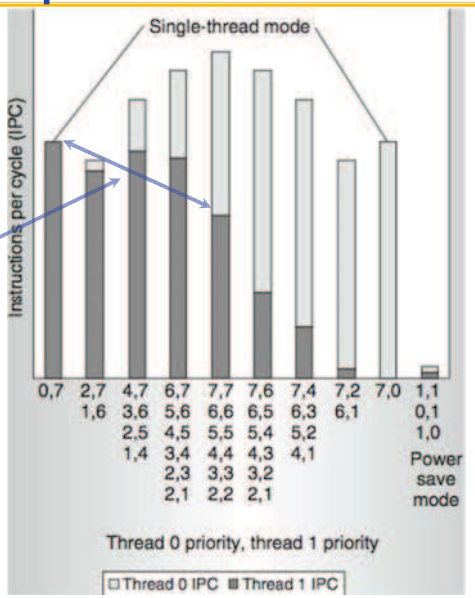


Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

Power 5 thread performance ...

Relative priority of each thread controllable in hardware.

For balanced operation, both threads run slower than if they "owned" the machine.



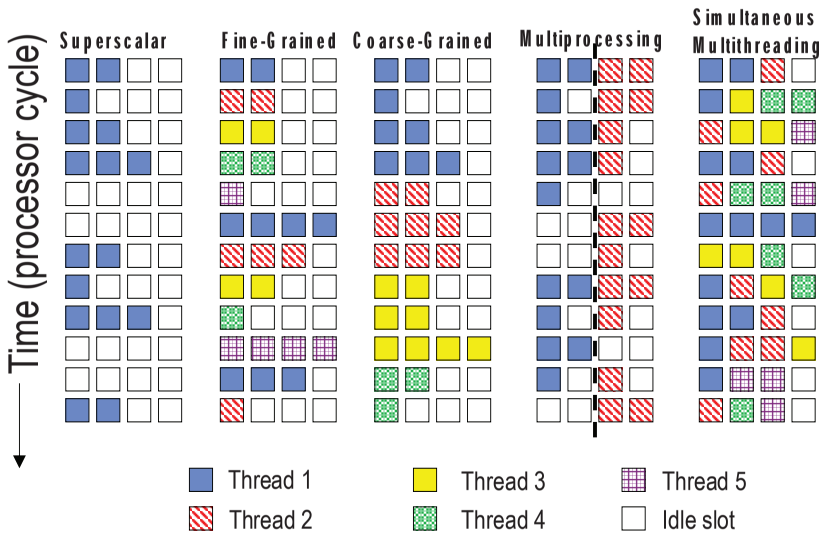
Changes in Power 5 to support SMT

- Increased associativity of L1 instruction cache and the instruction address translation buffers
- Added per thread load and store queues
- Increased size of the L2 (1.92 vs. 1.44 MB) and L3 caches
- Added separate instruction prefetch and buffering per thread
- Increased the number of virtual registers from 152 to 240
- Increased the size of several issue queues
- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support

Initial Performance of SMT

- Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
  - Pentium 4 is dual threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (26^2 runs) speed-ups from 0.90 to 1.58; average was 1.20
- Power 5, 8 processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate
- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains

Multithreaded Categories



Administrivia

- Midterm I: Wednesday 3/21
  Location: 405 Soda Hall
  TIME: 5:00-8:00
  - Can have 1 sheet of 8½x11 *handwritten* notes – both sides
  - No microfiche of the book!
- Meet at LaVal's afterwards for Pizza and Beverages
  - Great way for me to get to know you better
  - I'll Buy!
- CS252 First Project proposal due by Friday 3/2
  - Need two people/project (although can justify three for right project)
  - Complete Research project in 9 weeks
    - » Typically investigate hypothesis by building an artifact and measuring it against a "base case"
    - » Generate conference-length paper/give oral presentation
    - » Often, can lead to an actual publication.

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

Vector Supercomputers

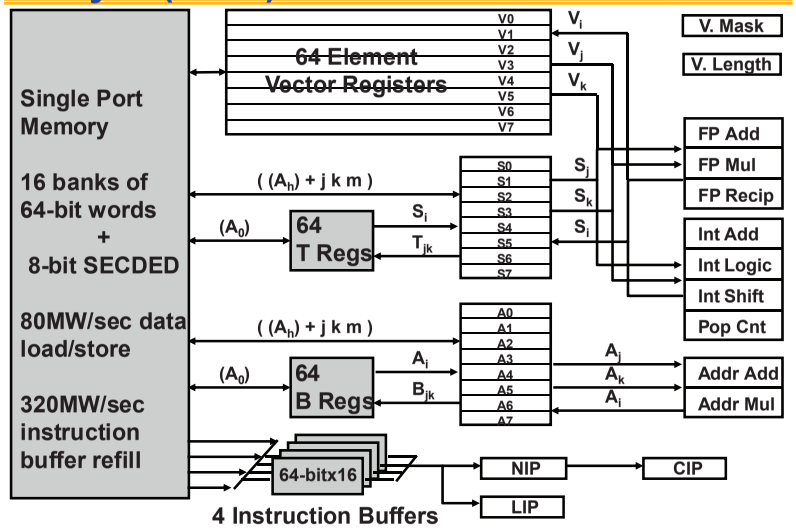
Epitomized by Cray-1, 1976:

Scalar Unit + Vector Extensions

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

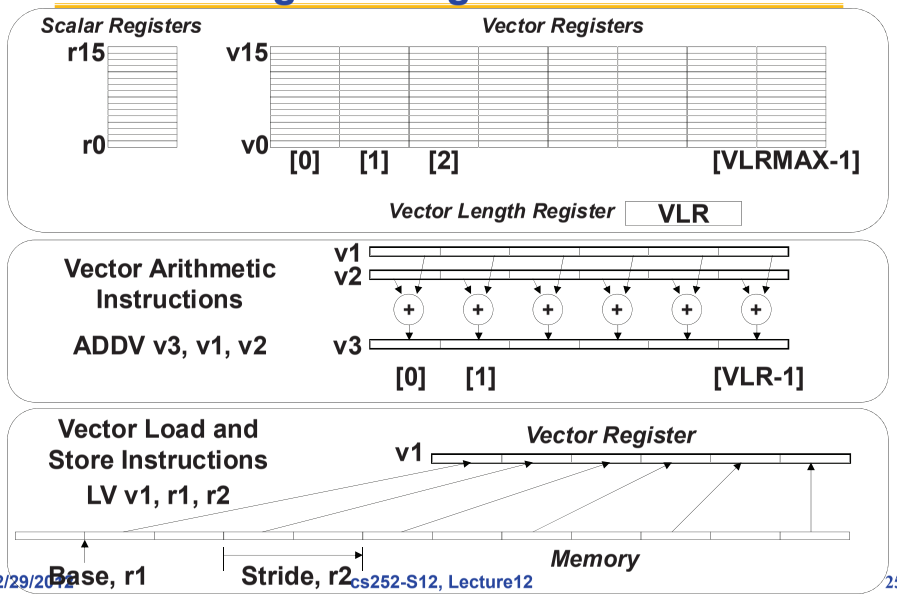
Cray-1 (1976)

Cray-1 (1976)



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

Vector Programming Model



Vector Code Example

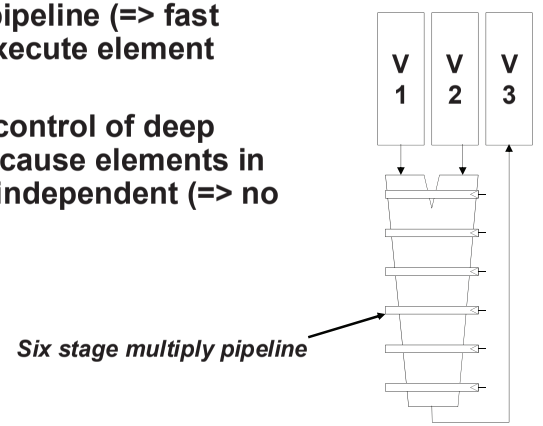
# C code	# Scalar Code	# Vector Code
for (i=0; i<64; i++)	LI R4, 64	LI VLR, 64
C[i] = A[i] + B[i];	loop:	LV V1, R1
	L.D F0, 0(R1)	LV V2, R2
	L.D F2, 0(R2)	ADDV.D V3, V1, V2
	ADD.D F4, F2, F0	SV V3, R3
	S.D F4, 0(R3)	
	DADDIU R1, 8	
	DADDIU R2, 8	
	DADDIU R3, 8	
	DSUBIU R4, 1	
	BNEZ R4, loop	

Vector Instruction Set Advantages

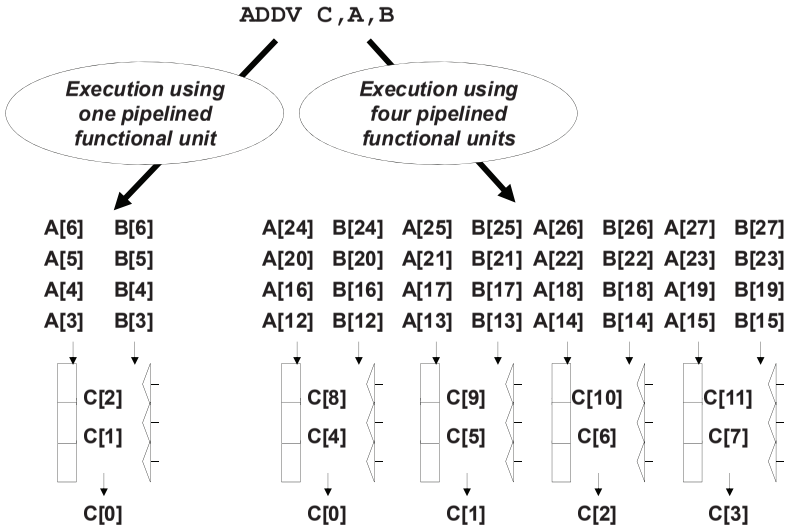
- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in the same pattern as previous instructions
  - access a contiguous block of memory (unit-stride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

Vector Arithmetic Execution

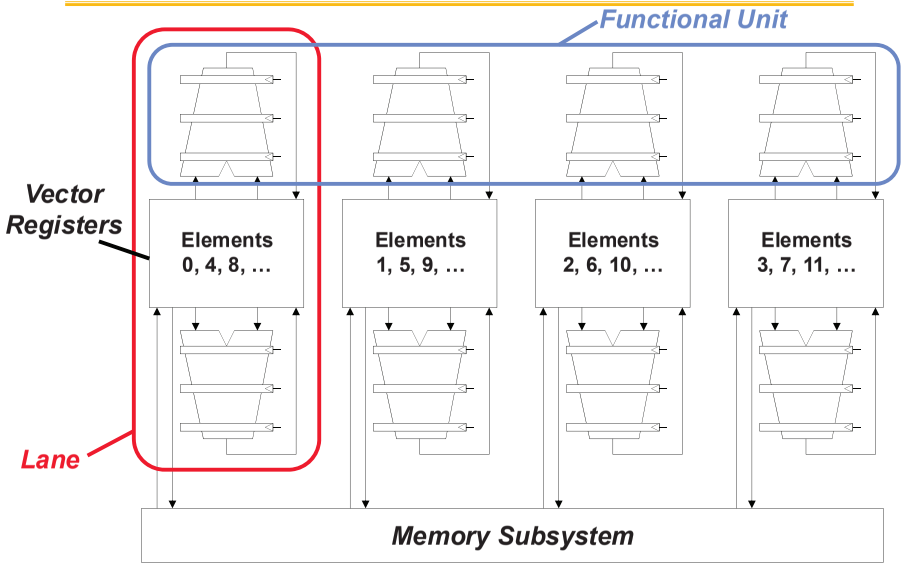
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



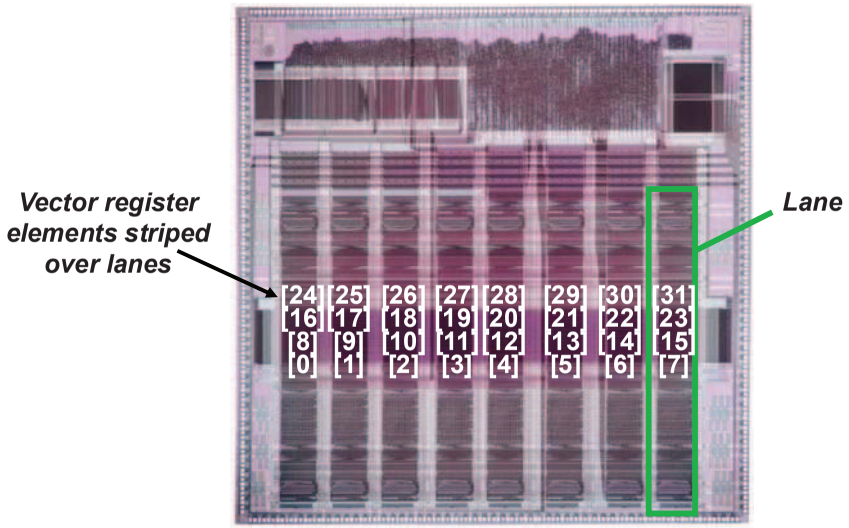
Vector Instruction Execution



Vector Unit Structure



T0 Vector Microprocessor (1995)



Vector Memory-Memory vs. 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

```
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

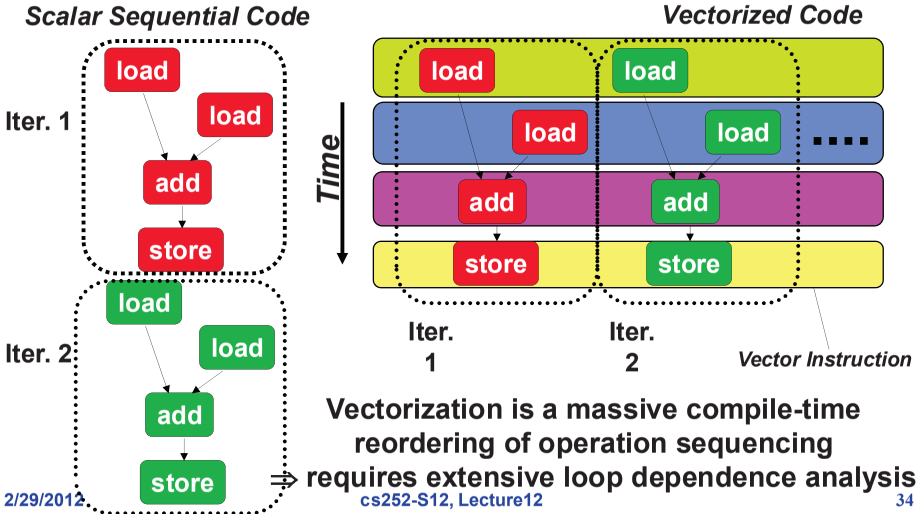
- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency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



Vectorization is a massive compile-time reordering of operation sequencing

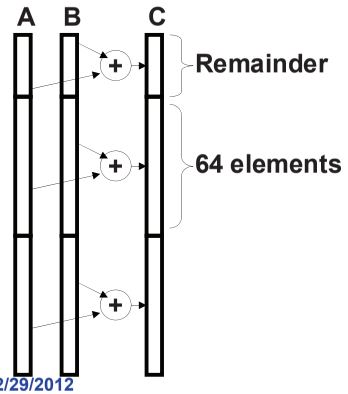
⇒ requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit into vector registers, *"Stripmining"*

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

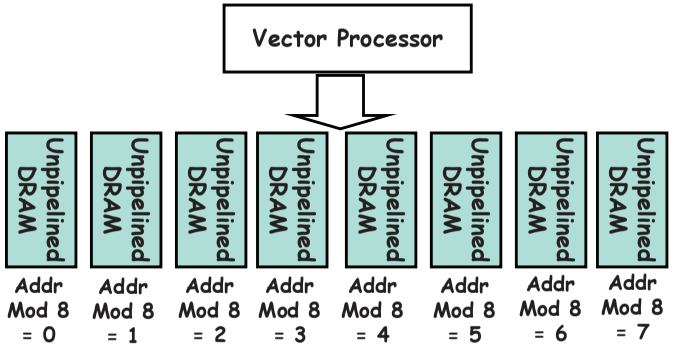



```
   ANDI R1, N, 63     # N mod 64
 MTC1 VLR, R1        # Do remainder
                loop:
              LV V1, RA
 DSLL R2, R1, 3     # Multiply by 8
 DADDU RA, RA, R2   # Bump pointer
              LV V2, RB
          DADDU RB, RB, R2
         ADDV.D V3, V1, V2
              SV V3, RC
          DADDU RC, RC, R2
 DSUBU N, N, R1 # Subtract elements
              LI R1, 64
 MTC1 VLR, R1    # Reset full length
 BGTZ N, loop    # Any more to do?
```


Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - » Contiguous block of information in memory
    - » Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - » Harder to optimize memory system for all possible strides
    - » Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - » Vector equivalent of register indirect
    - » Good for sparse arrays of data
    - » Increases number of programs that vectorize

Interleaved Memory Layout



- Great for unit stride:
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- What about non-unit stride?
  - Above good for strides that are relatively prime to 8
  - Bad for: 2, 4

How to get full bandwidth for Unit Stride?

- Memory system must sustain (# lanes x word) /clock
- 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* … 2 *l*

    word: -- … 0 1 2 … *m*-1 -- … *m*
  - may have 1024 banks in SRAM
- If desired throughput greater than one word per cycle
  - Either more banks (start multiple requests simultaneously)
  - Or wider DRAMS. Only good for unit stride or large data types
- More banks/weird numbers of banks good to support more strides at full bandwidth
  - can read paper on how to do prime number of banks efficiently

Avoiding Bank Conflicts

- Lots of banks

```
int x[256][512];
    for (j = 0; j < 512; j = j+1)
        for (i = 0; i < 256; i = i+1)
            x[i][j] = 2 * x[i][j];
```

- Even with 128 banks, since 512 is multiple of 128, conflict on word accesses
- SW: loop interchange or declaring array not power of 2 ("array padding")
- HW: Prime number of banks
  - bank number = address mod number of banks
  - address within bank = address / number of words in bank
  - modulo & divide per memory access with prime no. banks?
  - address within bank = address mod number words in bank
  - bank number? easy if 2^N words per bank

Finding Bank Number and Address within a bank

- Problem: Determine the number of banks, N_b and the number of words in each bank, N_w, such that:
  - given address x, it is easy to find the bank where x will be found, B(x), and the address of x within the bank, A(x).
  - for any address x, B(x) and A(x) are unique
  - the number of bank conflicts is minimized
- Solution: Use the Chinese remainder theorem to determine B(x) and A(x):

  $B(x) = x \text{ MOD } N_b$

  $A(x) = x \text{ MOD } N_w$ where N_b and N_w are co-prime (no factors)
  - Chinese Remainder Theorem shows that B(x) and A(x) unique.
- Condition allows N_w to be power of two (typical) if N_b is prime of form 2^m-1.
- Simple (fast) circuit to compute (x mod N_b) when $N_b = 2^m$-1:
  - Since $2^k = 2^{k-m}(2^m-1) + 2^{k-m} \Rightarrow 2^k \text{ MOD } N_b = 2^{k-m} \text{ MOD } N_b = \ldots = 2^j$ with $j < m$
  - And, remember that: (A+B) MOD C = [(A MOD C)+(B MOD C)] MOD C
  - for every power of 2, compute single bit MOD (in advance)
  - B(x) = sum of these values MOD N_b (low complexity circuit, adder with ~ m bits)

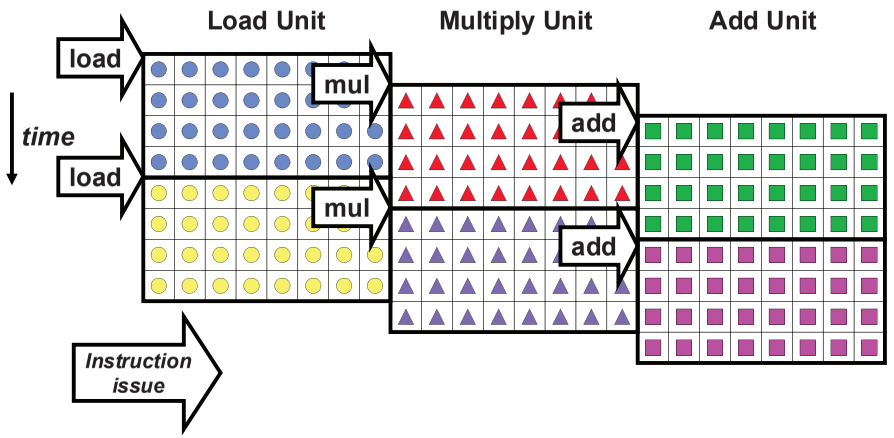
Administrivia

- Exam: Wednesday 3/30
  Location: 320 Soda
  TIME: 2:30-5:30
  - This info is on the Lecture page (has been)
  - Get on 8 ½ by 11 sheet of notes (both sides)
  - Meet at LaVal's afterwards for Pizza and Beverages
- CS252 First Project proposal due by Friday 3/4
  - Need two people/project (although can justify three for right project)
  - Complete Research project in 9 weeks
    - » Typically investigate hypothesis by building an artifact and measuring it against a "base case"
    - » Generate conference-length paper/give oral presentation
    - » Often, can lead to an actual publication.

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

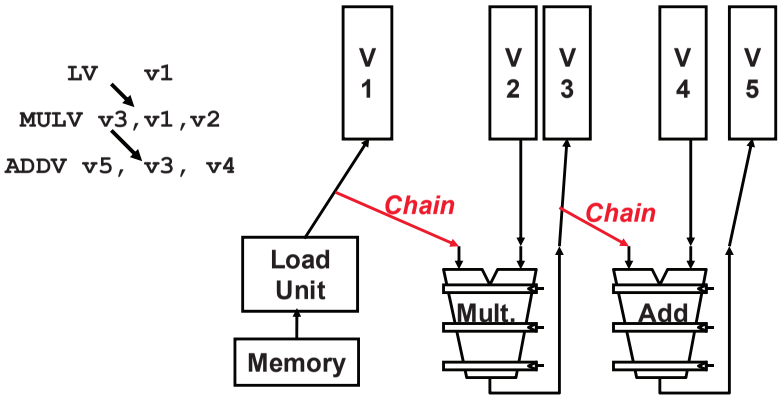
- example machine has 32 elements per vector register and 8 lanes



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

Vector Chaining

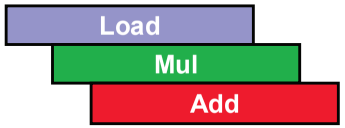
- Vector version of register bypassing
  - introduced with Cray-1



Vector Chaining Advantage

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

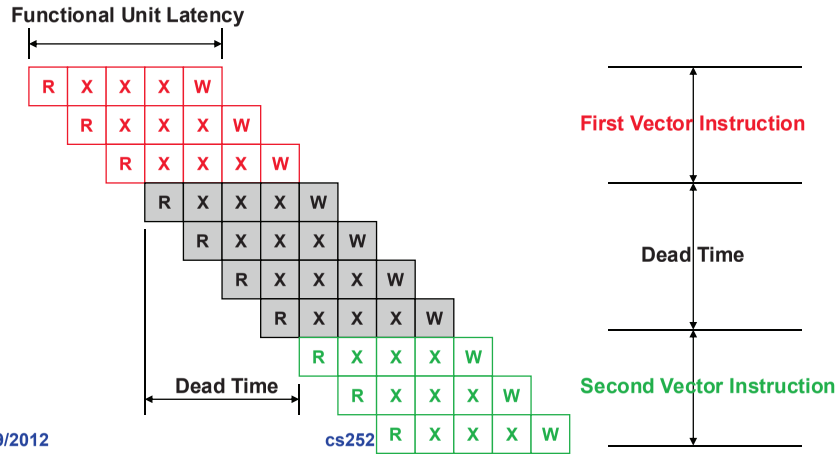
With chaining, can start dependent instruction as soon as first result appears



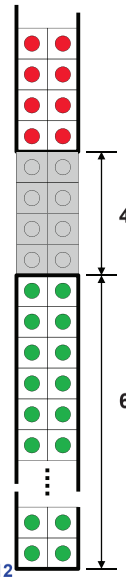
Vector Startup

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

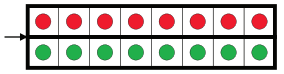


Cray C90, Two lanes

4 cycle dead time

Maximum efficiency 94% with 128 element vectors

No dead time



T0, Eight lanes

No dead time

100% efficiency with 8 element vectors

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 Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
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

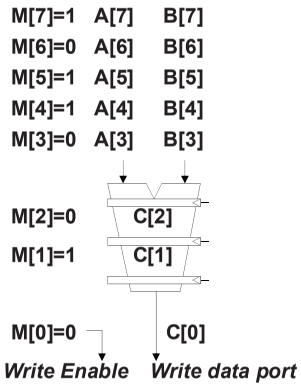
Code example:

```
        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

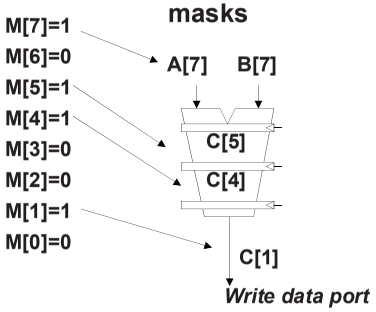
Simple Implementation

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



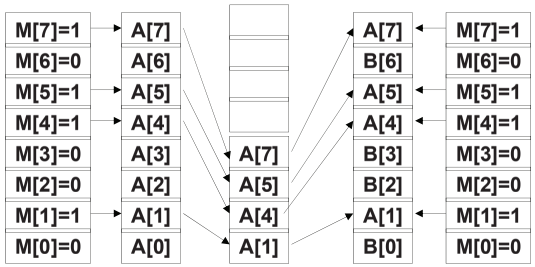
Density-Time Implementation

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



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



Compress *Expand*

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

Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
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

```
# Rearrange as:
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)
```


Novel Matrix Multiply Solution

- Consider the following:

```
/* 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;
    }
}
```

- Do you need to do a bunch of reductions? NO!
  - Calculate multiple independent sums within one vector register
  - You can vectorize the j loop to perform 32 dot-products at the same time (Assume Max Vector Length is 32)
- Show it in C source code, but can imagine the assembly vector instructions from it

Optimized Vector Example

```
/* 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; /* Init vector reg to zeros. */
    for (t=1; t<k; t++) {
    a_scalar = a[i][t]; /* Get scalar */
      b_vector[0:31] = b[t][j:j+31]; /* Get vector */

      /* Do a vector-scalar multiply. */
      prod[0:31] = b_vector[0:31]*a_scalar;

      /* 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];
  }
}
```


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)
- 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

"Vector" for Multimedia?

- Intel MMX: 57 additional 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)
- short vector: load, add, store 8 8-bit operands

- 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

Multithreading and Vector Summary

- Explicitly parallel (Data level parallelism or Thread level parallelism) is next step to performance
- Coarse grain vs. Fine grained multihreading
  - Only on big stall vs. every clock cycle
- Simultaneous Multithreading if fine grained multithreading based on OOO superscalar microarchitecture
  - Instead of replicating registers, reuse rename registers
- Vector is alternative model for exploiting ILP
  - If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
  - Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- Fundamental design issue is memory bandwidth
  - With virtual address translation and caching