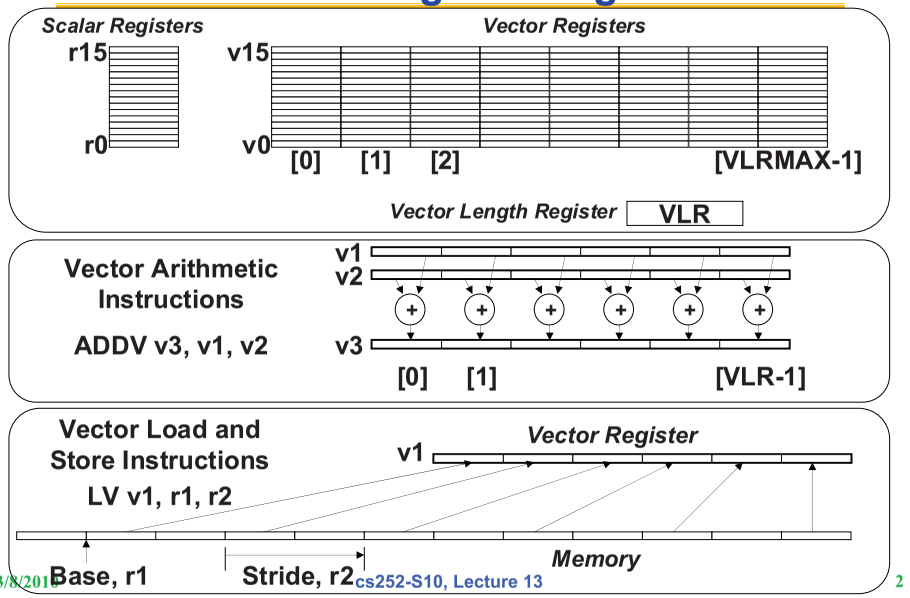
# CS252
# Graduate Computer Architecture
# Lecture 13

## Vector Processing (Con't)
## Intro to Multiprocessing
## March 8th, 2010

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

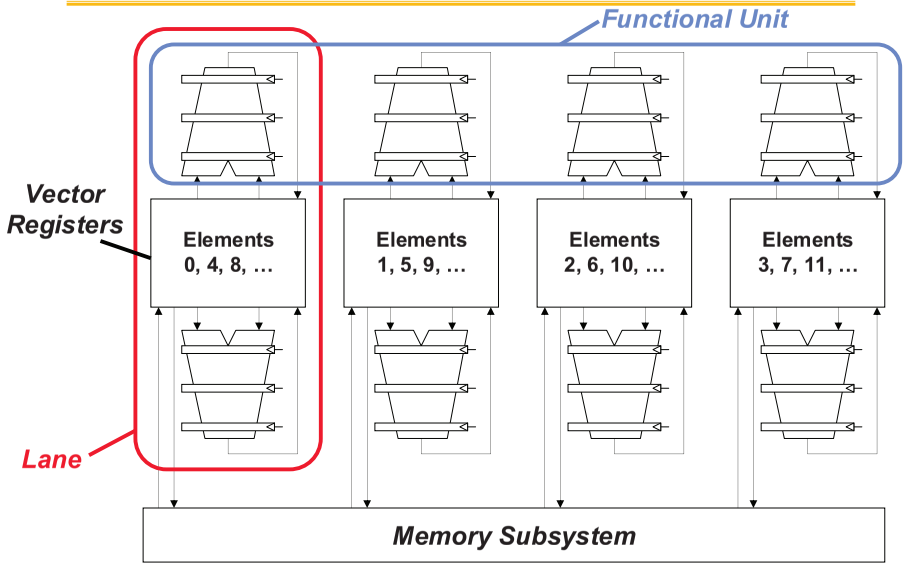
http://www.eecs.berkeley.edu/~kubitron/cs252

## Recall: Vector Programming Model






## Recall: Vector Unit Structure






## 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

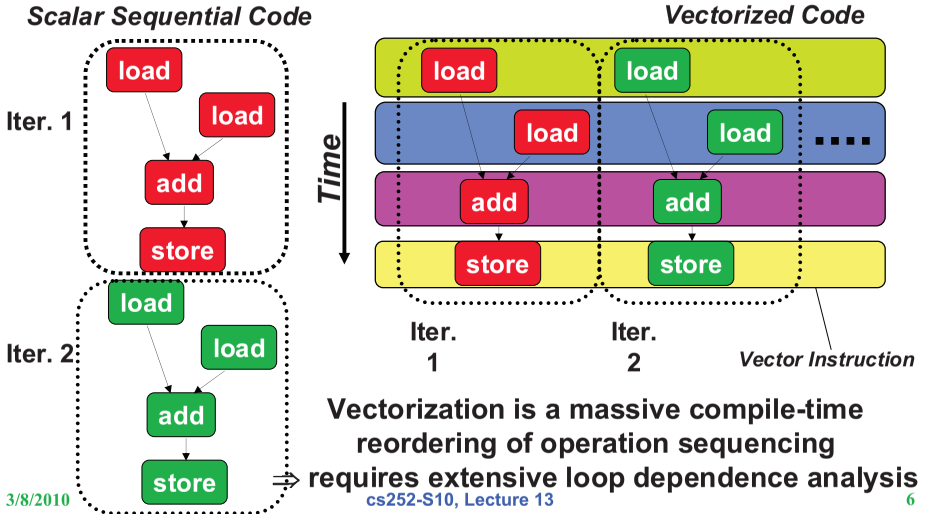
⇒*Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures*

*(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

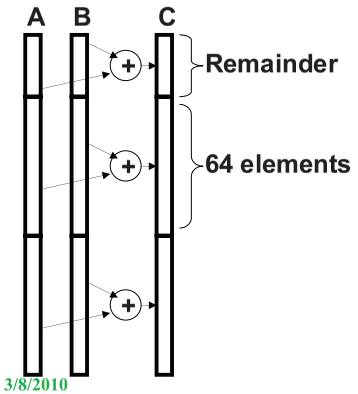


# Vector Stripmining

**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];
```



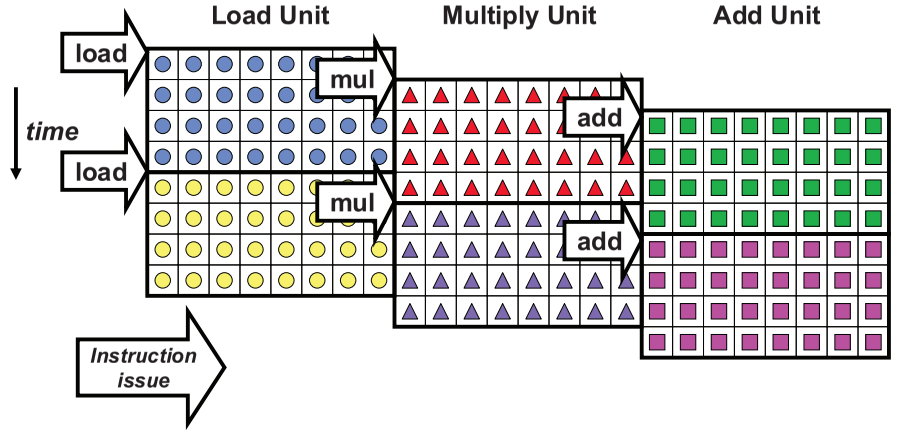

```
        ANDI R1, N, 63     # N mod 64
        MTC1 VLR, R1       # Do remainder
loop:
        LV V1, RA
        DSLL R2, R1, 3     # Multiply by 8
        DADDU RA, RA, R2   # Bump pointer
        LV V2, RB
        DADDU RB, RB, R2
        ADDV.D V3, V1, V2
        SV V3, RC
        DADDU RC, RC,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



# Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

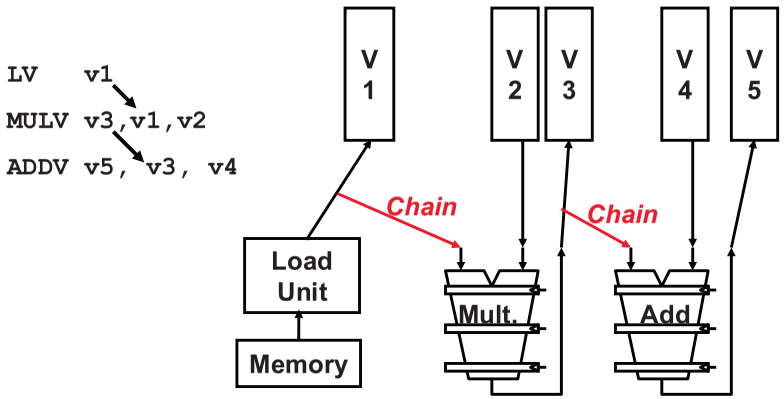
- example machine has 32 elements per vector register and 8 lanes




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

# Vector Chaining

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

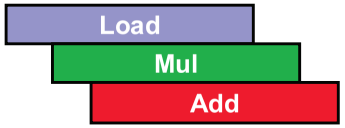





# Vector Chaining Advantage

- 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

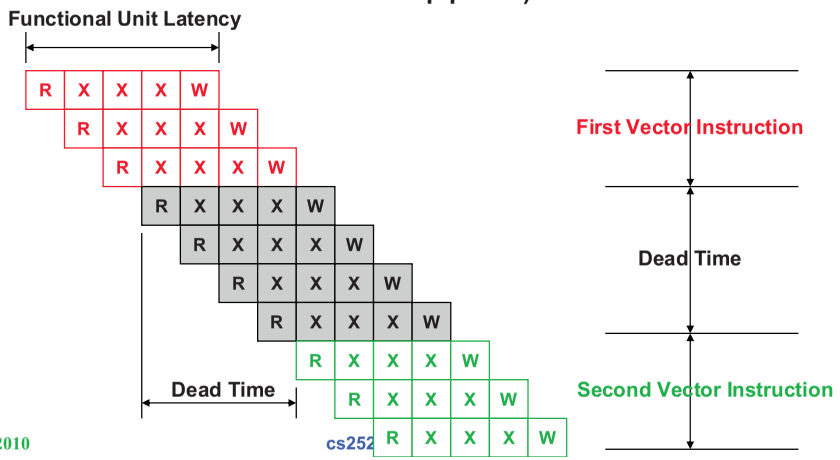





# Vector Startup

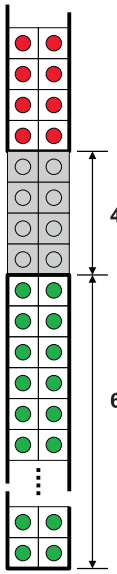
Two components of vector startup penalty

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

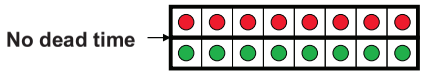





# Dead Time and Short Vectors




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




*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:**

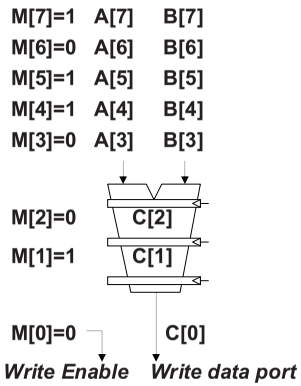
```
CVM             # Turn on all elements
LV vA, rA       # Load entire A vector
SGTVS.D vA, F0  # Set bits in mask register where A>0
LV vA, rB       # Load B vector into A under mask
SV vA, rA       # Store A back to memory under mask
```



## Masked Vector Instructions

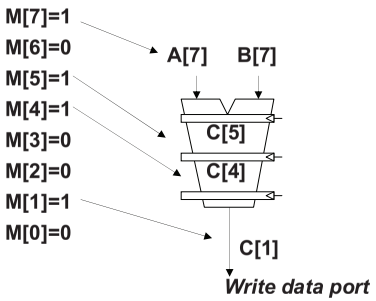
**Simple Implementation**

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




**Density-Time Implementation**

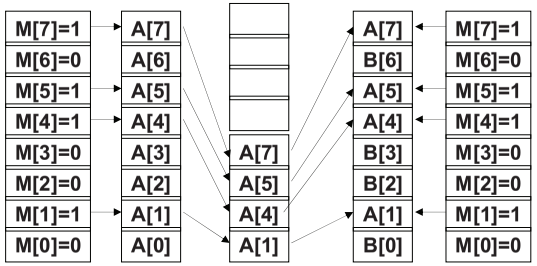
- scan mask vector and only execute elements with non-zero masks






## 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**

## Administrivia

- Exam: one week from Wednesday (3/17)
  Location: 310 Soda
  TIME: 6:00-9:00
  - 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
- I have your proposals.
  We need to meet to discuss them
  - Time this week? Wednesday after class



## 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];
  }
}
```

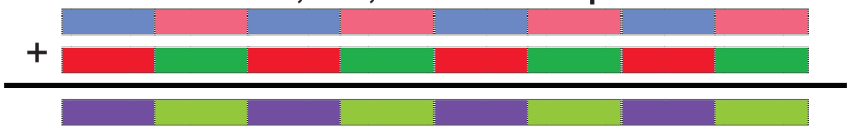
## Multimedia Extensions

- 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



## VLIW: Very Large Instruction Word (revisited)

- Each "instruction" has explicit coding for multiple operations
  - In IA-64, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches



## Recall: Loop Unrolling in VLIW

| *Memory reference 1* | *Memory reference 2* | *FP operation 1* | *FP op. 2* | *Int. op/ branch* | *Clock* |
|---|---|---|---|---|---|
| L.D F0,0(R1) | L.D F6,-8(R1) | | | | 1 |
| L.D F10,-16(R1) | L.D F14,-24(R1) | | | | 2 |
| L.D F18,-32(R1) | L.D F22,-40(R1) | ADD.D F4,F0,F2 | ADD.D F8,F6,F2 | | 3 |
| L.D F26,-48(R1) | | ADD.D F12,F10,F2 | ADD.D F16,F14,F2 | | 4 |
| | | ADD.D F20,F18,F2 | ADD.D F24,F22,F2 | | 5 |
| S.D 0(R1),F4 | S.D -8(R1),F8 | ADD.D F28,F26,F2 | | | 6 |
| S.D -16(R1),F12 | S.D -24(R1),F16 | | | | 7 |
| S.D -32(R1),F20 | S.D -40(R1),F24 | | | DSUBUI R1,R1,#48 | 8 |
| S.D -0(R1),F28 | | | | BNEZ R1,LOOP | 9 |

Unrolled 7 times to avoid delays

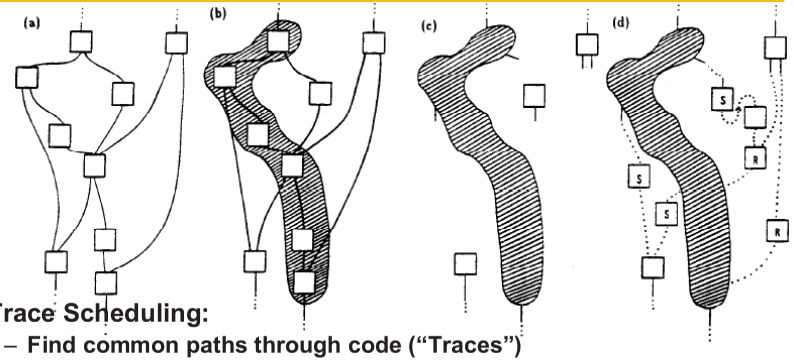
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

## Paper Discussion: VLIW and the ELI-512

Joshua Fisher




- Trace Scheduling:
  - Find common paths through code ("Traces")
  - Compact them
  - Build fixup code for trace exits (the "split" boxes)
  - Must not overwrite live variables⇒ use extra variables to store results
- N+1 way jumps
  - Used to handle exit conditions from traces
- Software prediction of memory bank usage
  - Use it to avoid bank conflicts/deal with limited routing



## Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict
- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code



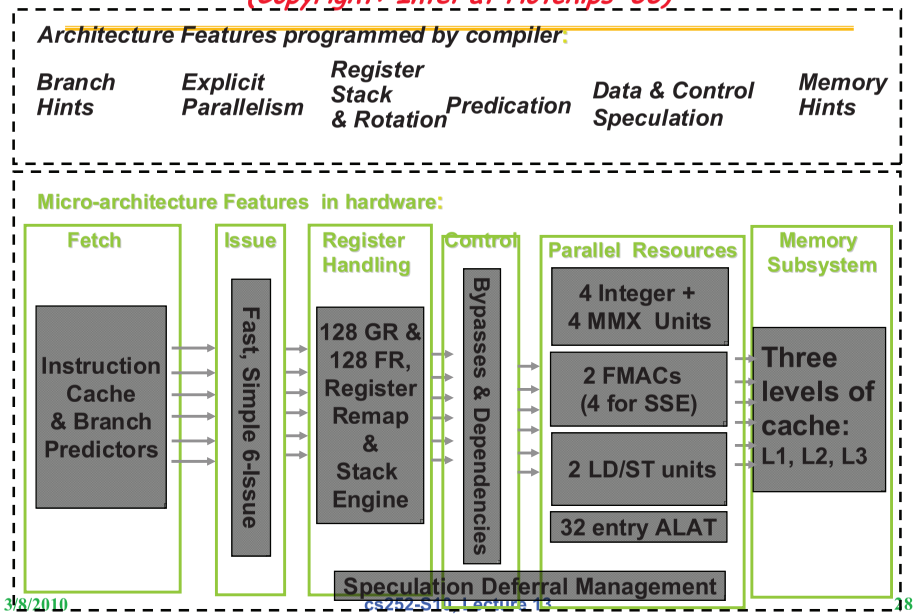
## Intel/HP IA-64 "Explicitly Parallel Instruction Computer (EPIC)"

- IA-64: instruction set architecture
  - 128 64-bit integer regs + 128 82-bit floating point regs
    - » Not separate register files per functional unit as in old VLIW
  - Hardware checks dependencies (interlocks ⇒ binary compatibility over time)
- 3 Instructions in 128 bit "bundles"; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr
- Predicated execution (select 1 out of 64 1-bit flags) ⇒ 40% fewer mispredictions?
- Speculation Support:
  - deferred exception handling with "poison bits"
  - Speculative movement of loads above stores + check to see if incorect
- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3



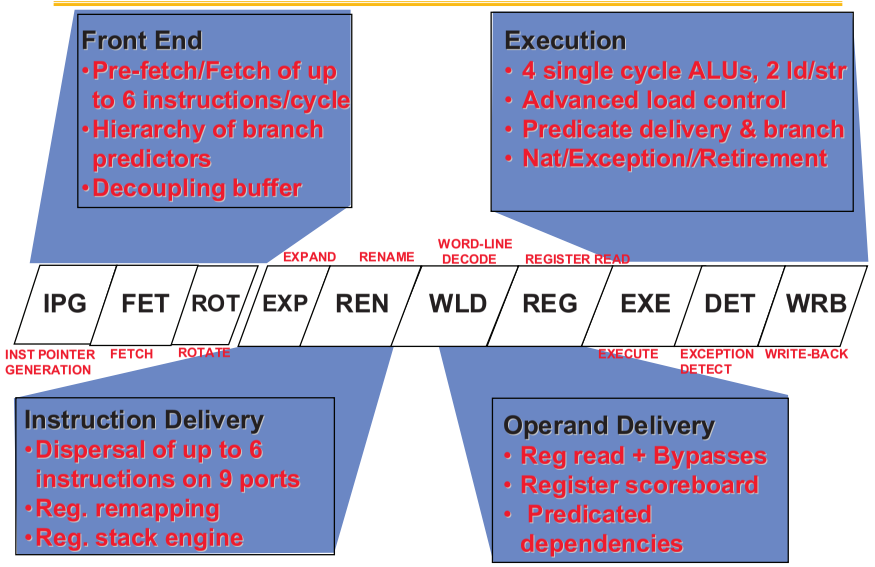
## Itanium™ EPIC Design Maximizes SW-HW Synergy

*(Copyright: Intel at Hotchips '00)*



# 10 Stage In-Order Core Pipeline

*(Copyright: Intel at Hotchips '00)*






# What is *Parallel* Architecture?

- A parallel computer is a collection of processing elements that cooperate to solve large problems
  - Most important new element: It is all about communication!
- What does programmer (or OS or Compiler) think about?
  - Models of computation:
    - » PRAM? BSP? Sequential Consistency?
  - Resource Allocation:
    - » how powerful are the elements?
    - » how much memory?
- What mechanisms must be in hardware vs software
  - What does a single processor look like?
    - » High performance general purpose processor
    - » SIMD processor/Vector Processor
  - Data access, Communication and Synchronization
    - » how do the elements cooperate and communicate?
    - » how are data transmitted between processors?
    - » what are the abstractions and primitives for cooperation?



# Flynn's Classification (1966)

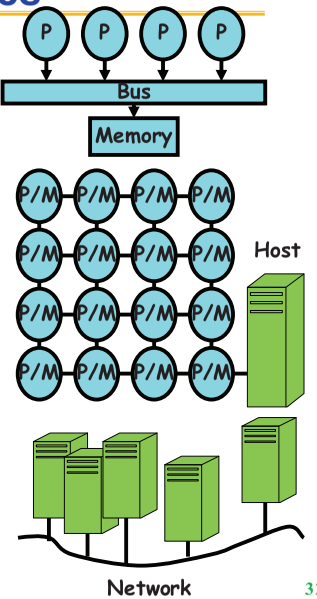
Broad classification of parallel computing systems

- SISD: Single Instruction, Single Data
  - conventional uniprocessor
- SIMD: Single Instruction, Multiple Data
  - one instruction stream, multiple data paths
  - distributed memory SIMD (MPP, DAP, CM-1&2, Maspar)
  - shared memory SIMD (STARAN, vector computers)
- MIMD: Multiple Instruction, Multiple Data
  - message passing machines (Transputers, nCube, CM-5)
  - non-cache-coherent shared memory machines (BBN Butterfly, T3D)
  - cache-coherent shared memory machines (Sequent, Sun Starfire, SGI Origin)
- MISD: Multiple Instruction, Single Data
  - Not a practical configuration

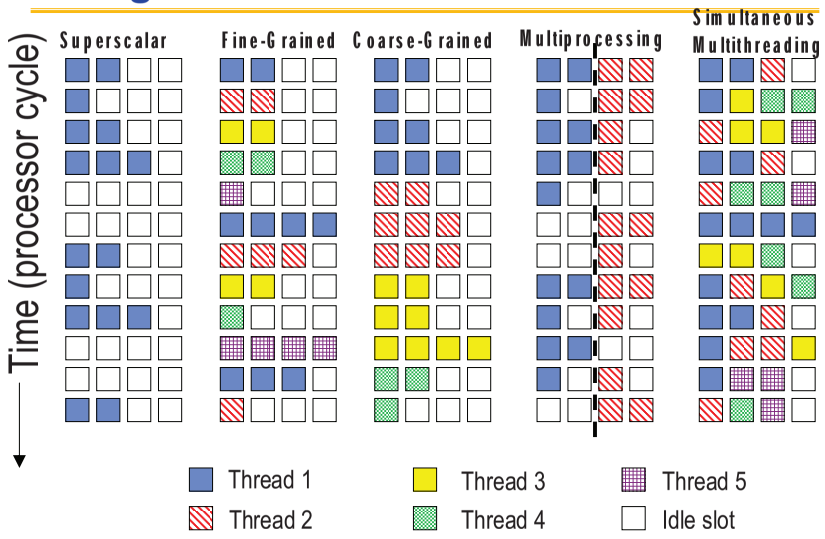


# Examples of MIMD Machines

- Symmetric Multiprocessor
  - Multiple processors in box with shared memory communication
  - Current MultiCore chips like this
  - Every processor runs copy of OS
- Non-uniform shared-memory with separate I/O through host
  - Multiple processors
    - » Each with local memory
    - » general scalable network
  - Extremely light "OS" on node provides simple services
    - » Scheduling/synchronization
  - Network-accessible host for I/O
- Cluster
  - Many independent machine connected with general network
  - Communication through messages



## Categories of Thread Execution






## Parallel Programming Models

- **Programming model** is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - What orderings exist between operations?
  - How do different threads of control synchronize?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?
- Cost
  - How do we account for the cost of each of the above?



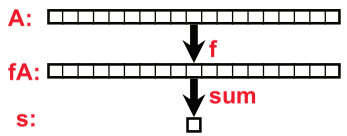
## Simple Programming Example

- Consider applying a function f to the elements of an array A and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

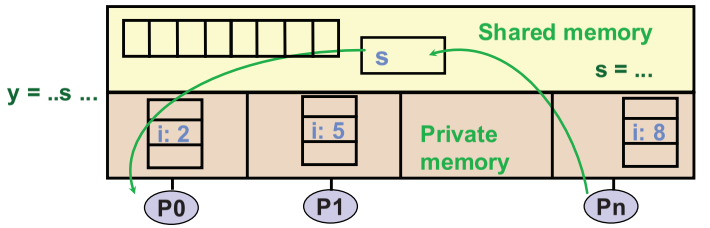
- Questions:
  - Where does A live? All in single memory? Partitioned?
  - What work will be done by each processors?
  - They need to coordinate to get a single result, how?

A = array of all data
fA = f(A)
s = sum(fA)






## Programming Model 1: Shared Memory

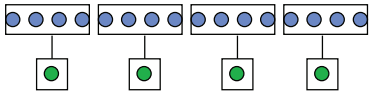



- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

## Simple Programming Example: SM

- **Shared memory strategy:**
  - small number p << n=size(A) processors
  - attached to single memory
- **Parallel Decomposition:**
  - Each evaluation and each partial sum is a task.
- **Assign n/p numbers to each of p procs**
  - Each computes independent "private" results and partial sum.
  - Collect the p partial sums and compute a global sum.

$$\sum_{i=0}^{n-1} f(A[i])$$



**Two Classes of Data:**

- **Logically Shared**
  - The original n numbers, the global sum.
- **Logically Private**
  - The individual function evaluations.
  - What about the individual partial sums?



## Shared Memory "Code" for sum

```
static int s = 0;

Thread 1                          Thread 2
for i = 0, n/2-1                  for i = n/2, n-1
    s = s + f(A[i])                   s = s + f(A[i])
```

- Problem is a race condition on variable s in the program
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously



## A Closer Look

A | 3 | 5 | f = square

```
static int s = 0;
```

| Thread 1 | | Thread 2 | |
|---|---|---|---|
| …. | | … | |
| compute f([A[i]) and put in reg0 | 9 | compute f([A[i]) and put in reg0 | 25 |
| reg1 = s | 0 | reg1 = s | 0 |
| reg1 = reg1 + reg0 | 9 | reg1 = reg1 + reg0 | 25 |
| s = reg1 | 9 | s = reg1 | 25 |
| … | | … | |

- Assume A = [3,5], f is the square function, and s=0 initially
- For this program to work, s should be 34 at the end
  - but it may be 34,9, or 25
- The *atomic* operations are reads and writes
  - Never see ½ of one number, but += operation is not atomic
  - All computations happen in (private) registers



## Improved Code for Sum

```
static int s = 0;
static lock lk;

Thread 1                          Thread 2
local_s1= 0                       local_s2 = 0
for i = 0, n/2-1                  for i = n/2, n-1
   local_s1 = local_s1 + f(A[i])     local_s2= local_s2 + f(A[i])
lock(lk);                         lock(lk);
s = s + local_s1                  s = s +local_s2
unlock(lk);                       unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
- The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

# What about Synchronization?

- All shared-memory programs need synchronization
- Barrier – global (/coordinated) synchronization
  - simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

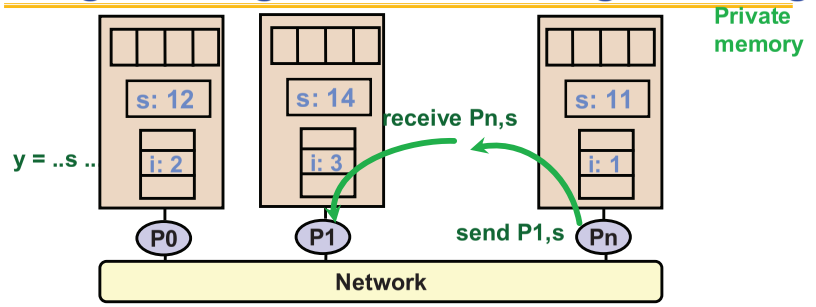
- Mutexes – mutual exclusion locks
  - threads are mostly independent and must access common data

```
lock *l = alloc_and_init();    /* shared */
lock(l);
 access data
unlock(l);
```

- Need atomic operations bigger than loads/stores
  - Actually – Dijkstra's algorithm can get by with only loads/stores, but this is quite complex (and doesn't work under all circumstances)
  - Example: atomic swap, test-and-test-and-set
- Another Option: Transactional memory
  - Hardware equivalent of optimistic concurrency
  - Some think that this is the answer to all parallel programming



# Programming Model 2: Message Passing




- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW



# Compute A[1]+A[2] on each processor

° First possible solution – what could go wrong?

| Processor 1 | Processor 2 |
|---|---|
| xlocal = A[1] | xlocal = A[2] |
| send xlocal, proc2 | send xlocal, proc1 |
| receive xremote, proc2 | receive xremote, proc1 |
| s = xlocal + xremote | s = xlocal + xremote |

° If send/receive acts like the telephone system? The post office?

° Second possible solution

| Processor 1 | Processor 2 |
|---|---|
| xlocal = A[1] | xloadl = A[2] |
| send xlocal, proc2 | receive xremote, proc1 |
| receive xremote, proc2 | send xlocal, proc1 |
| s = xlocal + xremote | s = xlocal + xremote |

° What if there are more than 2 processors?



# MPI – the de facto standard

- MPI has become the de facto standard for parallel computing using message passing
- Example:

```
for(i=1;i<numprocs;i++) {
  sprintf(buff, "Hello %d! ", i);
  MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,
           MPI_COMM_WORLD);
}
for(i=1;i<numprocs;i++) {
  MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
           MPI_COMM_WORLD, &stat);
  printf("%d: %s\n", myid, buff);
}
```

- Pros and Cons of standards
  - MPI created finally a standard for applications development in the HPC community → portability
  - The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation

## Which is better? SM or MP?

- Which is better, Shared Memory or Message Passing?
  - Depends on the program!
  - Both are "communication Turing complete"
    - » i.e. can build Shared Memory with Message Passing and vice-versa
- Advantages of Shared Memory:
  - Implicit communication (loads/stores)
  - Low overhead when cached
- Disadvantages of Shared Memory:
  - Complex to build in way that scales well
  - Requires synchronization operations
  - Hard to control data placement within caching system
- Advantages of Message Passing
  - Explicit Communication (sending/receiving of messages)
  - Easier to control data placement (no automatic caching)
- Disadvantages of Message Passing
  - Message passing overhead can be quite high
  - More complex to program
  - Introduces question of reception technique (interrupts/polling)

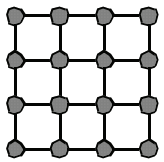


## What characterizes a network?

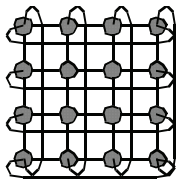
- Topology (what)
  - physical interconnection structure of the network graph
  - direct: node connected to every switch
  - indirect: nodes connected to specific subset of switches
- Routing Algorithm (which)
  - restricts the set of paths that msgs may follow
  - many algorithms with different properties
    - » gridlock avoidance?
- Switching Strategy (how)
  - how data in a msg traverses a route
  - circuit switching vs. packet switching
- Flow Control Mechanism (when)
  - when a msg or portions of it traverse a route
  - what happens when traffic is encountered?



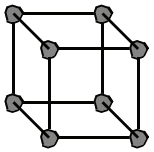
## Example: Multidimensional Meshes and Tori



2D Grid



2D Torus

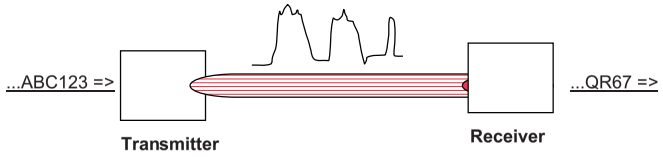


3D Cube

- *n*-dimensional array
  - $N = k_{d-1} X ... X k_0$ nodes
  - described by *n*-vector of coordinates $(i_{n-1}, ..., i_0)$
- *n*-dimensional *k*-ary mesh: $N = k^n$
  - $k = \sqrt[n]{N}$
  - described by *n*-vector of radix k coordinate
- *n*-dimensional *k*-ary torus (or *k*-ary *n*-cube)?



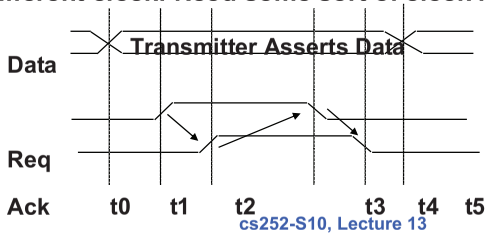
## Links and Channels




- *transmitter* converts stream of digital symbols into signal that is driven down the link
- *receiver* converts it back
  - tran/rcv share *physical protocol*
- trans + link + rcv form *Channel* for digital info flow between switches
- *link-level protocol* segments stream of symbols into larger units: packets or messages (framing)
- *node-level protocol* embeds commands for dest communication assist within packet

## Clock Synchronization?

- Receiver must be synchronized to transmitter
  - To know when to latch data
- Fully Synchronous
  - Same clock and phase: Isochronous
  - Same clock, different phase: Mesochronous
    - » High-speed serial links work this way
    - » Use of encoding (8B/10B) to ensure sufficient high-frequency component for clock recovery
- Fully Asynchronous
  - No clock: Request/Ack signals
  - Different clock: Need some sort of clock recovery?



## Conclusion

- 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
- VLIW: Explicitly Parallel
  - Trace Scheduling: Select primary "trace" to compress + fixup code
- Itanium/EPIC/VLIW is not a breakthrough in ILP
  - If anything, it is as complex or more so than a dynamic processor
- Multiprocessing
  - Multiple processors connect together
  - It is all about communication!
- Programming Models:
  - Shared Memory
  - Message Passing
- Networking and Communication Interfaces
  - Fundamental aspect of multiprocessing