EECS 252 Graduate Computer Architecture

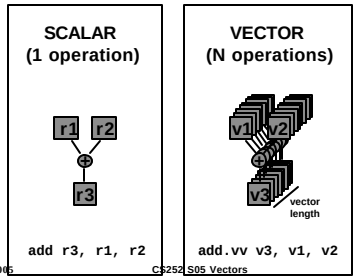
Lec 10 – Vector Processing

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~culler
http://www-inst.eecs.berkeley.edu/~cs252

Alternative Model:Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



What needs to be specified in a Vector Instruction Set Architecture?

- ISA in general
  - Operations, Data types, Format, Accessible Storage, Addressing Modes, Exceptional Conditions
- Vectors
  - Operations
  - Data types (Float, int, V op V, S op V)
  - Format
  - Source and Destination Operands
    - » Memory?, register?
  - Length
  - Successor (consecutive, stride, indexed, gather/scatter, ...)
  - Conditional operations
  - Exceptions

"DLXV" Vector Instructions

Instr.	Operands	Operation	Comment
• ADDV	V1,V2,V3	V1=V2+V3	vector + vector
• ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
• MULTV	V1,V2,V3	V1=V2xV3	vector x vector
• MULSV	V1,F0,V2	V1=F0xV2	scalar x vector
• LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
• LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
• LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	indir.("gather")
• CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
• MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
• MOV	VM,R1	Vec. Mask = R1	set vector mask

Properties of Vector Processors

- Each result independent of previous result
  => long pipeline, compiler ensures no dependencies
  => high clock rate
- Vector instructions access memory with known pattern
  => highly interleaved memory
  => amortize memory latency of over - 64 elements
  => no (data) caches required! (Do use instruction cache)
- Reduces branches and branch problems in pipelines
- Single vector instruction implies lots of work (- loop)
  => fewer instruction fetches

Operation & Instruction Count: RISC v. Vector Processor

(from F. Quintana, U. Barcelona.)

Spec92fp	Operations (Millions)			Instructions (M)		
Program	RISC	Vector	R / V	RISC	Vector	R / V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Vector reduces ops by 1.2X, instructions by 20X

Styles of Vector Architectures

- *memory-memory vector processors*: all vector operations are memory to memory
  - CDC Star100, Cyber203, Cyber205, 370 vector extensions
- *vector-register processors*: all vector operations between vector registers (except load and store)
  - Vector equivalent of load-store architectures
  - Introduced in the Cray-1
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC
  - We assume vector-register for rest of lectures

Components of Vector Processor

- *Vector Register*: fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-32 vector registers, each holding 64-128 64-bit elements
- *Vector Functional Units (FUs)*: fully pipelined, start new operation every clock
  - typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit
- *Vector Load-Store Units (LSUs)*: fully pipelined unit to load or store a vector; may have multiple LSUs
- *Scalar registers*: single element for FP scalar or address
- Cross-bar to connect FUs, LSUs, registers

Common Vector Metrics

- R_∞: MFLOPS rate on an infinite-length vector
  - vector "speed of light"
  - Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
  - (R_n is the MFLOPS rate for a vector of length n)
- $N_{1/2}$: The vector length needed to reach one-half of R_∞
  - a good measure of the impact of start-up
- N_V: The vector length needed to make vector mode faster than scalar mode
  - measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit

DAXPY (Y = a * X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
        LD    F0,a
        ADDI  R4,Rx,#512 ;last address to load
loop:   LD    F2,0(Rx)   ;load X(i)
        MULTD F2,F0,F2   ;a*X(i)
        LD    F4,0(Ry)   ;load Y(i)
        ADDD  F4,F2,F4   ;a*X(i) + Y(i)
        SD    F4,0(Ry)   ;store into Y(i)
        ADDI  Rx,Rx,#8   ;increment index to X
        ADDI  Ry,Ry,#8   ;increment index to Y
        SUB   R20,R4,Rx  ;compute bound
        BNZ   R20,loop   ;check if done
```

```
        LD    F0,a       ;load scalar a
        LV    V1,Rx      ;load vector X
        MULTS V2,F0,V1   ;vector-scalar mult.
        LV    V3,Ry      ;load vector Y
        ADDV  V4,V2,V3   ;add
        SV    Ry,V4      ;store the result
```

578 (2+9*64) vs. 321 (1+5*64) ops (1.8X)

578 (2+9*64) vs. 6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

Example Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	8	2 L, 1 S
Cray YMP	1988	166 MHz	8	64	8	2 L, 1 S
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

Vector Example with dependency

```
/* 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;
    }
}
```

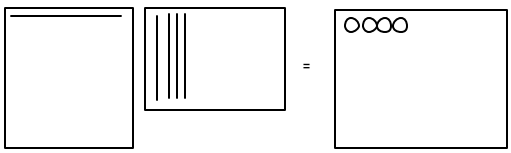
Straightforward Solution: Use scalar processor

- This type of operation is called a reduction
- Grab one element at a time from a vector register and send to the scalar unit?
  - Usually bad, since path between scalar processor and vector processor not usually optimized all that well
- Alternative: Special operation in vector processor
  - shift all elements left vector length elements or collapse into a compact vector all elements not masked
  - Supported directly by some vector processors
  - Usually not as efficient as normal vector operations
    - » (Number of cycles probably logarithmic in number of bits!)

Novel Matrix Multiply Solution

- You don't need to do reductions for matrix multiply
- You can calculate multiple independent sums within one vector register
- You can vectorize the j loop to perform 32 dot-products at the same time
- (Assume Maximul Vector Length is 32)
- Show it in C source code, but can imagine the assembly vector instructions from it

Matrix Multiply Dependences



- N^2 independent recurrences (inner products) of length N
- Do k = VL of these in parallel

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];
  }
}
```


Novel, Step #2

- What vector stride?
- What length?
- It's actually better to interchange the i and j loops, so that you only change vector length once during the whole matrix multiply
- To get the absolute fastest code you have to do a little register blocking of the innermost loop.

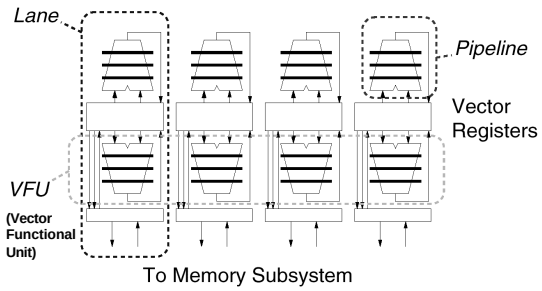
CS 252 Administrivia

- Exam:
- This info is on the Lecture page (has been)
- Meet at LaVal's afterwards for Pizza and Beverages

Vector Implementation

- Vector register file
  - Each register is an array of elements
  - Size of each register determines maximum vector length
  - Vector length register determines vector length for a particular operation
- Multiple parallel execution units = "lanes" (sometimes called "pipelines" or "pipes")

Vector Terminology: 4 lanes, 2 vector functional units



Vector Execution Time

- Time = f(vector length, data dependicies, struct. hazards)
- *Initiation rate*: rate that FU consumes vector elements (= number of lanes; usually 1 or 2 on Cray T-90)
- *Convoy*: set of vector instructions that can begin execution in same clock (no struct. or data hazards)
- *Chime*: approx. time for a vector operation
- *m* convoys take *m* chimes; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead; good approximization for long vectors)

```
1: LV    V1,Rx     ;load vector X
2: MULV  V2,F0,V1  ;vector-scalar mult.
   LV    V3,Ry     ;load vector Y
3: ADDV  V4,V2,V3  ;add
4: SV    Ry,V4     ;store the result
```

4 convoys, 1 lane, VL=64
=> 4 x 64 = 256 clocks
(or 4 clocks per result)

Hardware Vector Length

- What to do when software vector length doesn't exactly match hardware vector length?
- *vector-length register* (VLR) controls the length of any vector operation, including a vector load or store. (cannot be > the length of vector registers)

```
    do 10 i = 1, n
10  Y(i) = a * X(i) + Y(i)
```

- Don't know n until runtime! n > Max. Vector Length (MVL)?

Strip Mining

- Suppose Vector Length > Max. Vector Length (MVL)?
- *Strip mining*: generation of code such that each vector operation is done for a size Š to the MVL
- 1st loop do short piece (n mod MVL), rest VL = MVL

```
   low = 1
   VL = (n mod MVL)  /*find the odd size piece*/
   do 1 j = 0,(n / MVL)  /*outer loop*/
      do 10 i = low,low+VL-1  /*runs for length VL*/
            Y(i) = a*X(i) + Y(i)  /*main operation*/
10    continue
      low = low+VL  /*start of next vector*/
      VL = MVL  /*reset the length to max*/
1     continue
```


DLXV Start-up Time

- *Start-up time*: pipeline latency time (depth of FU pipeline); another sources of overhead

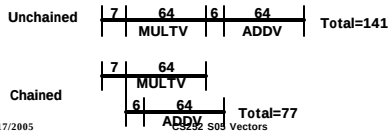
Operation	Start-up penalty (from CRAY-1)
Vector load/store	12
Vector multiply	7
Vector add	6

Assume convoys don't overlap; vector length = n:

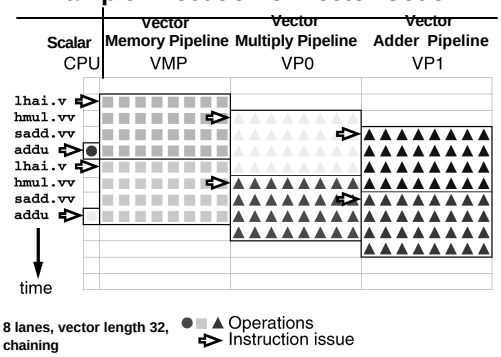
Convoy	*Start*	*1st result*	*last result*	
1. LV	0	12	11+n (=12+n-1)	
2. MULV, LV	12+n	12+n+7	18+2n	*Multiply startup*
	12+n+1	12+n+13	24+2n	*Load start-up*
3. ADDV	25+2n	25+2n+6	30+3n	*Wait convoy 2*
4. SV	31+3n	31+3n+12	42+4n	*Wait convoy 3*

Vector Opt #1: Chaining

- Suppose:
  MULV V1,V2,V3
  ADDV V4,V1,V5 ; separate convoy?
- *chaining*: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector
- *Flexible chaining*: allow vector to chain to any other active vector operation => more read/write ports
- As long as enough HW, increases convoy size



Example Execution of Vector Code



8 lanes, vector length 32, chaining

Vector Stride

- Suppose adjacent elements not sequential in memory

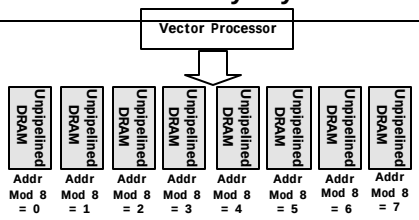
```
do 10 i = 1,100
     do 10 j = 1,100
          A(i,j) = 0.0
               do 10 k = 1,100
10             A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- Either B or C accesses not adjacent (800 bytes between)
- *stride*: distance separating elements that are to be merged into a single vector (caches do unit stride)
  => LVWS (load vector with stride) instruction
- Think of addresses per vector element

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
  - Better: prime number of banks...!

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/ 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

Vector Opt #2: Sparse Matrices

- Suppose:

```
          do    100 i = 1,n
100       A(K(i)) = A(K(i)) + C(M(i))
```

- *gather* (LVI) operation takes an *index vector* and fetches data from each address in the index vector
  - This produces a "dense" vector in the vector registers
- After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store (SVI), using the same index vector
- Can't be figured out by compiler since can't know elements distinct, no dependencies
- Use CVI to create index 0, 1xm, 2xm, ..., 63xm

Sparse Matrix Example

- Cache (1993) vs. Vector (1988)

	IBM RS6000	Cray YMP
Clock	72 MHz	167 MHz
Cache	256 KB	0.25 KB
Linpack	140 MFLOPS	160 (1.1)
Sparse Matrix (Cholesky Blocked)	17 MFLOPS	125 (7.3)

- Cache: 1 address per cache block (32B to 64B)
- Vector: 1 address per element (4B)

Vector Opt #3: Conditional Execution

- Suppose:

```
   do 100 i = 1, 64
      if (A(i) .ne. 0) then
         A(i) = A(i) – B(i)
      endif
100 continue
```

- *vector-mask control* takes a Boolean vector: when *vector-mask register* is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.
- Still requires clock even if result not stored; if still performs operation, what about divide by 0?

Parallelism and Power

- If code is vectorizable, then simple hardware, more energy efficient than Out-of-order machines.
- Can decrease power by lowering frequency so that voltage can be lowered, then duplicating hardware to make up for slower clock:

$$\text{Power} \propto CV^2 f$$

$$f = \frac{1}{n} f_0 \qquad Lanes = n \times Lanes_0 \qquad V = \delta V_0;\ \delta < 1$$

$$\text{Performance Constant} \quad \text{Power Change: } \delta^2 < 1$$

- Note that V_o can be made as small as permissible within process constraints by simply increasing "*n*"

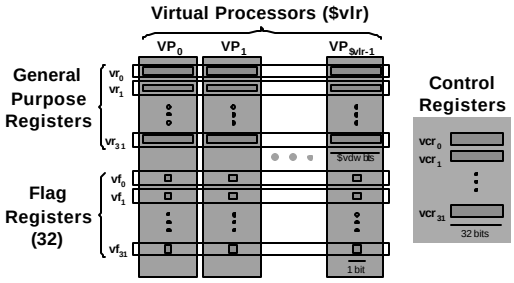
Vector Options

- Use vectors for inner loop parallelism (no surprise)
  - One dimension of array: A[0, 0], A[0, 1], A[0, 2], ...
  - think of machine as, say, 16 vector regs each with 32 elements
  - 1 instruction updates 32 elements of 1 vector register
- and for outer loop parallelism!
  - 1 element from each column: A[0,0], A[1,0], A[2,0], ...
  - think of machine as 32 "virtual processors" (VPs) each with 16 scalar registers! (- multithreaded processor)
  - 1 instruction updates 1 scalar register in 64 VPs
- Hardware identical, just 2 compiler perspectives

Virtual Processor Vector Model: Treat like SIMD multiprocessor

- Vector operations are SIMD (single instruction multiple data) operations
  - Each virtual processor has as many scalar "registers" as there are vector registers
  - There are as many virtual processors as current vector length.
  - Each element is computed by a virtual processor (VP)

Vector Architectural State



Designing a Vector Processor

- Changes to scalar
- How Pick Vector Length?
- How Pick Number of Vector Registers?
- Context switch overhead
- Exception handling
- Masking and Flag Instructions

Changes to scalar processor to run vector instructions

- Decode vector instructions
- Send scalar registers to vector unit (vector-scalar ops)
- Synchronization for results back from vector register, including exceptions
- Things that don't run in vector don't have high ILP, so can make scalar CPU simple

How Pick Vector Length?

- Longer good because:
  1) Hide vector startup
  2) lower instruction bandwidth
  3) tiled access to memory reduce scalar processor memory bandwidth needs
  4) if know max length of app. is < max vector length, no strip mining overhead
  5) Better spatial locality for memory access
- Longer not much help because:
  1) diminishing returns on overhead savings as keep doubling number of element
  2) need natural app. vector length to match physical register length, or no help (lots of short vectors in modern codes!)

How Pick Number of Vector Registers?

- More Vector Registers:
  1) Reduces vector register "spills" (save/restore)
     - » 20% reduction to 16 registers for su2cor and tomcatv
     - » 40% reduction to 32 registers for tomcatv
     - » others 10% -15%
  2) Aggressive scheduling of vector instructinons: better compiling to take advantage of ILP
- Fewer:
  1) Fewer bits in instruction format (usually 3 fields)
  2) Easier implementation

Context switch overhead: Huge amounts of state!

- Extra dirty bit per processor
  - If vector registers not written, don't need to save on context switch
- Extra valid bit per vector register, cleared on process start
  - Don't need to restore on context switch until needed

Exception handling: External Interrupts?

- If external exception, can just put pseudo-op into pipeline and wait for all vector ops to complete
  - Alternatively, can wait for scalar unit to complete and begin working on exception code assuming that vector unit will not cause exception and interrupt code does not use vector unit

Exception handling: Arithmetic Exceptions

- Arithmetic traps harder
- Precise interrupts => large performance loss!
- Alternative model: arithmetic exceptions set vector flag registers, 1 flag bit per element
- Software inserts trap barrier instructions from SW to check the flag bits as needed
- IEEE Floating Point requires 5 flag bits

Exception handling: Page Faults

- Page Faults must be precise
- Instruction Page Faults not a problem
  - Could just wait for active instructions to drain
  - Also, scalar core runs page-fault code anyway
- Data Page Faults harder
- Option 1: Save/restore internal vector unit state
  - Freeze pipeline, dump vector state
  - perform needed ops
  - Restore state and continue vector pipeline

Exception handling: Page Faults

- Option 2: expand memory pipeline to check addresses before send to memory + memory buffer between address check and registers
  - multiple queues to transfer from memory buffer to registers; check last address in queues before load 1st element from buffer.
  - Per Address Instruction Queue (PAIQ) which sends to TLB and memory while in parallel go to Address Check Instruction Queue (ACIQ)
  - When passes checks, instruction goes to Committed Instruction Queue (CIQ) to be there when data returns.
  - On page fault, only save intructions in PAIQ and ACIQ

Masking and Flag Instructions

- Flag have multiple uses (conditional, arithmetic exceptions)
- Alternative is conditional move/merge
- Clear that fully masked is much more effiecient that with conditional moves
  - Not perform extra instructions, avoid exceptions
- Downside is:

1) extra bits in instruction to specify the flag regsiter
2) extra interlock early in the pipeline for RAW hazards on Flag registers

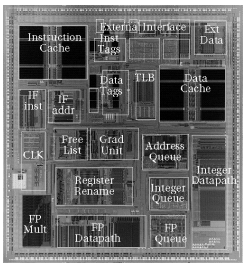
Flag Instruction Ops

- Do in scalar processor vs. in vector unit with vector ops?
- Disadvantages to using scalar processor to do flag calculations (as in Cray):

1) if MVL > word size => multiple instructions; also limits MVL in future
2) scalar exposes memory latency
3) vector produces flag bits 1/clock, but scalar consumes at 64 per clock, so cannot chain together

- Proposal: separate Vector Flag Functional Units and instructions in VU

MIPS R10000 vs. T0



*See http://www.icsi.berkeley.edu/real/spert/t0-intro.html

Vectors Are Inexpensive

Scalar

- N ops per cycle $\Rightarrow O(N^2)$ circuitry
- HP PA-8000
  - 4-way issue
  - reorder buffer: 850K transistors
    - incl. 6,720 5-bit register number comparators

Vector

- N ops per cycle $\Rightarrow O(N + \varepsilon N^2)$ circuitry
- T0 vector micro
  - 24 ops per cycle
  - 730K transistors total
    - only 23 5-bit register number comparators
  - No floating point

Vectors Lower Power

Single-issue Scalar	Vector
One instruction fetch, decode, dispatch per operation	One inst fetch, decode, dispatch per vector
Arbitrary register accesses, adds area and power	Structured register accesses
Loop unrolling and software pipelining for high performance increases instruction cache footprint	Smaller code for high performance, less power in instruction cache misses
All data passes through cache; waste power if no temporal locality	Bypass cache
One TLB lookup per load or store	One TLB lookup per group of loads or stores
Off-chip access in whole cache lines	Move only necessary data across chip boundary

Superscalar Energy Efficiency Even ~~Worse~~

Superscalar

- Control logic grows quad-ratically with issue width
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy

Vector

- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
  - Whether to use vector mask or compress/expand for conditionals

Vector Applications

Limited to scientific computing?

- **Multimedia Processing** (compress., graphics, audio synth, image proc.)
- **Standard benchmark kernels** (Matrix Multiply, FFT, Convolution, Sort)
- **Lossy Compression** (JPEG, MPEG video and audio)
- **Lossless Compression** (Zero removal, RLE, Differencing, LZW)
- **Cryptography** (RSA, DES/IDEA, SHA/MD5)
- **Speech and handwriting recognition**
- **Operating systems/Networking** (`memcpy`, `memset`, parity, checksum)
- **Databases** (hash/join, data mining, image/video serving)
- **Language run-time support** (stdlib, garbage collection)
- **even SPECint95**

Reality: Sony Playstation 2000

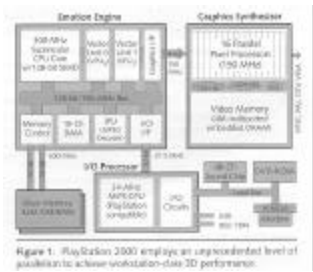
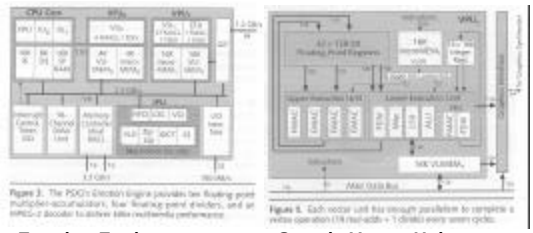


Figure 2. PlayStation 2000 screenshot. Source: Namco

- **(as reported in Microprocessor Report, Vol 13, No. 5)**
  - **Emotion Engine: 6.2 GFLOPS, 75 million polygons per second**
  - **Graphics Synthesizer: 2.4 Billion pixels per second**
  - **Claim: *Toy Story* realism brought to games!**

Playstation 2000 Continued



- Emotion Engine:
  - Superscalar MIPS core
  - Vector Coprocessor Pipelines
  - RAMBUS DRAM interface
- Sample Vector Unit
  - 2-wide VLIW
  - Includes Microcode Memory
  - High-level instructions like matrix-multiply

"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

New Architecture Directions?

- "…media processing will become the dominant force in computer arch. & microprocessor design."
- "... new media-rich applications... involve significant real-time processing of continuous media streams, and make heavy use of vectors of packed 8-, 16-, and 32-bit integer and Fl. Pt."
- Needs include high memory BW, high network BW, continuous media data types, real-time response, fine grain parallelism
  - "How Multimedia Workloads Will Change Processor Design", Diefendorff & Dubey, *IEEE Computer* (9/97)

Why do most vector extensions fail?

- Amdahl's law?
- Poor integration of vector data storage and operations with scalar
- CDC Star100 spent most of its time transposing matrices so consecutive stride
- Many attached array processors of the 70s and 80s (DSPs of the 90s and 00s)
- Single-Chip Fujitsu Vector unit was beautiful
  - Most designs tacked it on (Meiko CS-2)
    - » Operated on physical addresses, oblivious to cache
  - Need to be able to stream a lot of data through
  - Same data is sometimes accessed as scalar data
- Thinking Machine CM-5 vector memory controller

Summary

- 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
- Will multimedia popularity revive vector architectures?