

What is a vector instruction?

- What is a vector instruction?
 - Vector instructions handle many data values with a single instruction
 - Registers often contain a set of values
 - All values in set treated the “same” way
 - Vector instructions may operate on
 - Integers & floating point numbers
 - Bit vectors
- Where are they used?
 - Scientific computation (numerically intensive)
 - Graphics: bit-oriented vectors
 - MMX (x86)
 - AltiVec (PowerPC)



Problems with scalar processors

- How can scalar processors be sped up?
 - Use deeper pipelines
 - In longer pipelines, pipeline latencies become an issue
 - Reduce the instruction fetch/decode rate: for a given amount of data, fetch fewer instructions
 - Make instructions more complex?
 - Make instructions operate on more values?
- Speed up scalar processors with vectors
 - One instruction operates on many values
 - Rather than fetching 64 or more instructions to perform 64 FP adds, the CPU fetches only one
 - Good for small instruction caches!



Using vectors to avoid scalar bottlenecks

- Independent computations
 - Computation on each vector element is independent of all other vector elements
 - Deep pipelines can be used without creating data hazards
 - ⇒ Compiler must generate vector instructions
- Lots of work per instruction
 - Each instruction can cause many operation
 - ⇒ Fewer instructions need to be fetched and decoded
- Reduce control hazards and loop overhead
 - A vector instruction can function as a loop in a scalar architecture
 - No branches!
 - Lower loop overhead
 - No control hazards

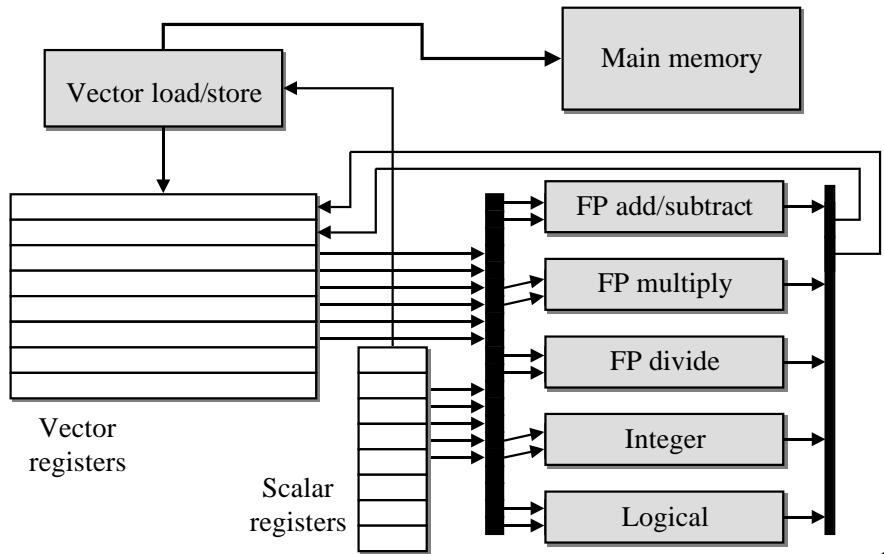


Using vectors to avoid scalar bottlenecks

- Optimize memory access
 - The memory access pattern for an entire vector load/store is known at instruction issue
 - ⇒ CPU may be able to get all of the data by paying the latency for memory access only a few times
- Overlapping vector operations
 - Overlap vector operations if there are enough functional units
 - Keep CPU busy with useful work
 - Reduce execution time
 - Requires more hardware, but hardware provides performance improvements
- Memory-memory vs. register-memory vector architectures



Simple DLX vector architecture (DLXV)



16-Mar-00



UMBC

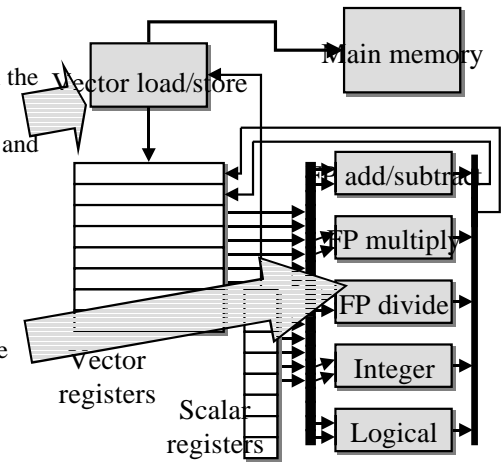
CMSC 611 (Advanced Computer Architecture), Spring 2000

Appendix B

5

DLXV architecture

- Vector load-store unit
 - Interfaces the vector unit with the memory
 - Moves data between memory and CPU
 - Fully pipelined: one word per clock cycle after startup
- Vector functional units
 - Fully pipelined: start a new operation on every clock cycle
 - Control unit detects hazards



16-Mar-00



UMBC

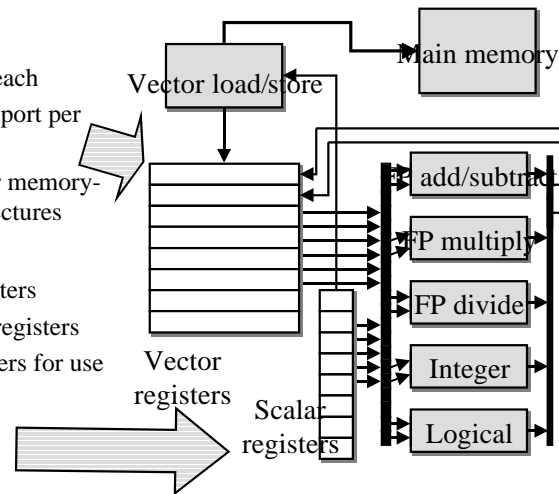
CMSC 611 (Advanced Computer Architecture), Spring 2000

Appendix B

6

DLXV architecture

- Vector registers
 - 8 registers, 64 values each
 - 2 read ports & 1 write port per register (is it enough?)
 - May not be needed for memory-memory vector architectures
- Scalar registers
 - Regular DLX FP registers
 - Regular DLX integer registers
 - Special-purpose registers for use by the vector unit
 - Vector length
 - Vector-mask



16-Mar-00



UMBC

CMSC 611 (Advanced Computer Architecture), Spring 2000

Appendix B

7

Sample vectorizable code

- Implement $\vec{Y} = a\vec{X} + \vec{Y}$
 - X and Y are vectors
 - A is a scalar
 - SAXPY/DAXPY loop (S or D indicates single or double precision)
 - Very common operation in scientific codes
- Code for DLX at right
 - Interlocks between the MULTD and ADDD and the memory operations
 - Possible problems with branches
 - Total instruction count ≈ 600

```
for (i = 0; i < 64; i++) {
    Y[i] = a * X[i] + Y[i];
}
```

```
LD    F0,a
ADDD  R4,Rx,#512
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 Y[i]
ADDD  Rx,Rx,#8 ; X index++
ADDD  Ry,Ry,#8 ; Y index++
SUB   R20,R4,Rx ; loop bound
BNEZ  R20,Loop ; loop if not done
```

16-Mar-00



UMBC

CMSC 611 (Advanced Computer Architecture), Spring 2000

Appendix B

8

Vectorized code for DAXPY

- Original code had
 - Lots of dependencies
 - Lots of loop overhead
- Vectorized code has
 - No looping!
 - Only a few vector instructions
 - Reduced instruction bandwidth (6 instructions)
 - Fewer interlocks: encountered only at startup
 - Simpler decoding: fewer dependencies to work out

```
LD    F0,a
ADDI  R4,Rx,#512
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 Y[i]
ADDI  Rx,Rx,#8 ; X index++
ADDI  Ry,Ry,#8 ; Y index++
SUB   R20,R4,Rx ; loop bound
BNEZ  R20,Loop ; loop if not done
```

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

Calculating vector execution time

- Terms (not “official”; made up by textbook authors)
 - Convoy: a group of vector instructions that could be issued in the “same” cycle because there are no dependencies between them
 - Chime: the time a maximum-length vector instruction takes to complete its execution
- Basic performance
 - Approximate execution time for a sequence of vector instructions is number of convoys * chime length
 - Only approximate because it ignores startup overheads
 - ⇒ Overheads are often short compared to instruction execution time

Vector startup latency

- Startup latency: delay before the first result comes out of the vector pipeline
 - Pipeline produces one result per cycle thereafter
 - Effect on performance reduced with longer vector lengths
 - Irrelevant if vectors infinitely long (but this isn't realistic)
- Startup latency == pipeline depth
- Sample startup latencies (for DLXV)
 - Load/store: 10 cycles
 - Multiply: 7 cycles
 - Add: 6 cycles
 - Memory often has a longer startup latency (why?)

Sample performance calculation

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

Idealized

Realistic

- Only MULTSV & second LV may be combined => 4 convoys
- Sequence requires 4 chimes
- Total time required => $64 * 4 = 256$
- Requires 4 cycles/element
- Compute actual start & finish times for each convoy

		Start	Finish
LD	F0,a	0	
LV	V1,Rx	0	9+n
MULTSV	V2,F0,V1	10+n	16+2n
LV	V3,Ry	10+n	20+2n
ADDV	V4,V2,V3	20+2n	26+3n
SV	Ry,V4	26+3n	36+4n

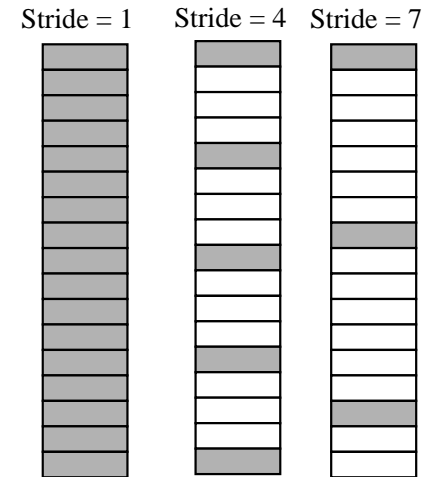
 - Total time is $37+4n$, or 293 cycles
 - Requires 4.58 cycles/element

Vector load-store unit issues

- Load-store units may not be able to complete one result per cycle (unlike most pipelined functional units)
- Long startup latencies
 - Relatively slow memory delays the first word of data
 - Data caches don't usually help vector processors (why?)
- Avoid memory conflicts
 - Vector processors often access several banks of memory at once
 - CPU gets more than one word per memory cycle
- Non-sequential memory accesses
 - Programs often need to load a vector from non-sequential elements
 - Done using *strided* memory access

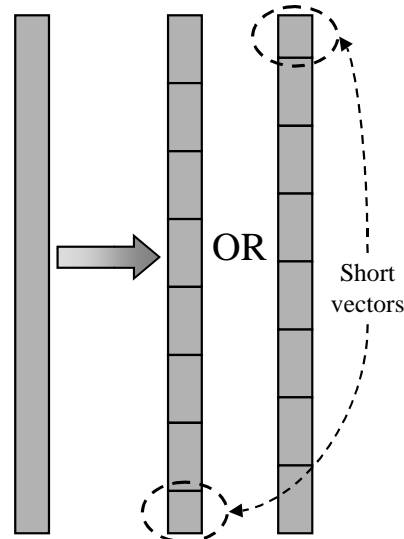
Vector stride

- Data in memory may not be sequential
 - Distance between adjacent elements in the vector is called the stride
- Strided access needed only for load and store
 - ⇒ Vectors held in a register are accessed normally
- Stride & vector address obtained from general purpose registers
- Stride and number of memory banks should be relatively prime
 - ⇒ Spreads accesses evenly for better performance



Handling odd-length vectors & strip mining

- Must handle vectors of less than the maximal length
 - Done by setting the vector length register (VLR)
- Strip mining example (500 elements)
 - Create 7 operations that run on full-length (64 element) vectors
 - Create 1 operation that runs on 52 elements
 - Shorter vector can be the first or last handled
 - Making it first simplifies the end-of-loop checks



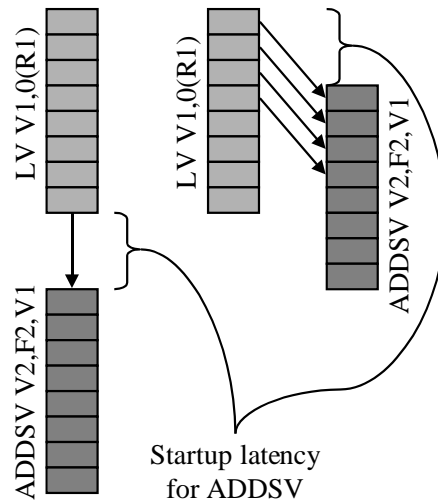
Performance for odd-size vectors

- Estimate loop performance using same method as before
 - Include time per element
 - Include vector instruction startup time
 - Include loop overhead
 - Doesn't include loop startup (paid once per execution, not once per loop)
- Compute T_{start} by adding up all of the vector startup latencies (excluding those that overlap in convoys)
- Compute T_{chime} by counting the convoys

$$T_n = \left\lceil \frac{n}{\text{max vector length}} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

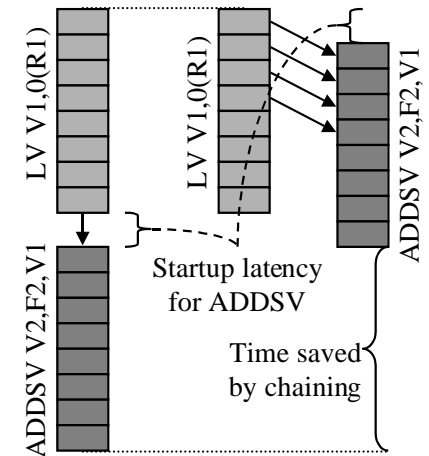
Vector chaining

- Chaining \approx forwarding for vector operations
 - Entire result from a vector operation may take 64+ cycles to produce
 - First element is ready after the startup latency
 - Could be fed into a second operation that uses the vector register as a source,
 - Time to do two chained operations is $(\text{length} + \text{startup}_{\text{op1}} + \text{startup}_{\text{op2}})$
- Chaining reduces overall time by overlapping vector instructions



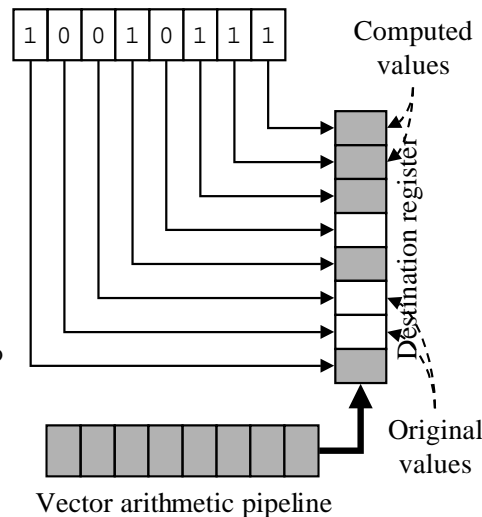
Vector chaining: example

- Example
 - LV V1, 0(Rx)
 - ADDSV V2, F0, V1
 - Without chaining, requires $10 + 64 + 6 + 64 = 146$ cycles
 - With chaining, the ADDSV could start after the load produced its first element
 - Reduces total time to $10 + 6 + 64 = 82$ cycles
 - Total time reduced to 56.2% of the original time
- Long chains (multiple instructions chained together) can drastically cut execution time



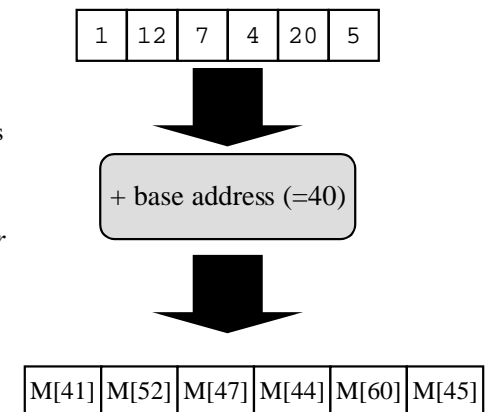
Vector mask control

- Vector mask register: one bit for each element in a vector register
 - Bit set \Rightarrow perform operation
 - Bit clear \Rightarrow do no operation
 - Functional unit busy for a cycle regardless of bit set/clear
 - Vector mask register set by
 - Copying a value from a scalar register
 - Doing a vector-based operation to set each bit individually
 - Example: SNESV
- \Rightarrow Allows the handling of conditions inside loops



Handling sparse matrices

- Sparse matrix: much of matrix is zero
 - Zero elements aren't stored
 - Non-zero elements represented as index-value pairs
 - Example: $A[5] = 4 \Rightarrow 5,4$
- Sparse matrices use *scatter-gather*
 - Memory indices stored in a vector
 - Load uses vector values as pointers or offsets
- Scatter-gather also useful when indices determined on-the-fly



Measures of vector processor performance

- R_∞ : MFLOPS rate on an infinitely long vector
 - Ignores startup latency
 - Useful for determining the throughput on really long vectors
- $N_{1/2}$: vector length necessary to achieve half R_∞
 - Shows how quickly vector performance fails if non-maximal vectors are used
 - Affected greatly by startup latency
- N_v : vector length necessary so that using vector operations to do a computation will be faster than using scalar operations
 - Shows how long vectors have to be to be useful
 - Fast scalar operations and startup latency affect this number



Vector performance: example

- Assume
 - One memory pipeline, 500 MHz
 - $T_{base} = 0$ and $T_{loop} = 15$
- Compute
 - $T_{start} = 10$ (load) + 7 (multiply) + 10 + 6 (add) + 12 (store) = 45
 - Need 3 chimes

Time to compute n elements

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_n = 0 + \left\lceil \frac{n}{64} \right\rceil \times (15 + 45) + n \times 3$$

DAXPY with chaining

LV	V1, Rx	; chained w/MULTSV
MULTSV	V2, F0, V1	
LV	V3, Ry	; chained w/ADDV
ADDV	V4, V2, V3	
SV	Ry, V4	; store the result

$$\frac{\text{cycles}}{\text{element}} = \lim_{n \rightarrow \infty} \left(\frac{3n + \frac{n \times (15 + 45)}{64}}{n} \right) = 3 + \frac{60}{64} = 3.94$$

$$R_\infty = 2 \times 500 / 3.94 = 253.8 \text{ MFLOPS}$$

$$\text{For } N_{1/2}: \frac{253.8}{2} = \frac{2 \times 500}{\text{cycles/element}}, n \leq 64$$

$$7.88n = 0 + 1 \times (15 + 45) + n \times 3 \Rightarrow n = 12.3$$



Vector processor gotchas

- Remember the startup latency!
 - High startup latency will drastically lower performance because vectors can be short
- Improve scalar & vector performance together
 - Otherwise, the machine will perform poorly overall
 - Most *instructions* are scalar, even though most *operations* are vector
- Get a good memory system!
 - True for any processor running heavy scientific codes!
 - This is the main reason that desktop workstations can't match big iron for heavy-duty scientific use
 - ⇒ Can't move data in and out of the CPU fast enough

