Motivation
Vectors
SIMD (multimedia) instructions (brief recap)
GPUs (project presentations)
Recall SIMD from Chapter 5
Figure 4.2 The basic structure of a vector architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and memory accesses. The figure shows vector units for logical and integer operations so that VMIPS looks like a standard vector processor that usually includes these units; however, we will not be discussing these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.

We use VMIPS from an older edition. Book uses very similar RV64V, but it was still in transition at time of printing.
What are Vector Instructions?

A vector is a one-dimensional array of numbers

```
float A[64], B[64], C[64]
```

Original motivation: Many scientific programs operate on vectors of floating point data

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

Multimedia, graphics, neural networks, other emerging apps also operate on vectors of data

A vector instruction performs an operation on each vector element

```
ADDVV C, A, B
```
Why Vector Instructions?

Want deeper pipelines, BUT
  Interlock logic complexity grows
  Stalls due to data hazards increase
  Stalls due to control hazards increase
  Instruction issue bottleneck
  Stalls due to cache misses

Vector instructions allow deeper pipelines
  No *intra*-vector interlock logic
  No *intra*-vector data hazards
  “Inner” loop control hazards eliminated
  Need not issue multiple instrns per cycle (but many current proc do)
  Vectors have known memory access patterns
VMIPS Architecture

Strongly based on CRAY

Vector-Register architecture
  Load/store architecture
  All vector operations use registers (except load/store)
  Optimized for small vectors

Extend MIPS with vector instructions
  Scalar unit
    Eight vector registers (V0-V7): each is 64 elements, 64 bits wide

Five Vector Functional Units
  FP+, FP*, FP/, integer & logical
  Fully pipelined

Vector Load/Store Units
  Fully pipelined
Vector-Vector Instructions
Operate on two vectors
Produce a third vector

\[
\text{for } (i=0; \ i<64; \ i++) \\
\quad V1[i] = V2[i] + V3[i] \\
\]

\text{ADDVV.D V1, V2, V3}

Vector-Scalar Instructions
Operate on one vector, one scalar
Produce a third vector

\[
\text{for } (i=0; \ i<64; \ i++) \\
\quad V1[i] = F0 + V3[i] \\
\]

\text{ADDVS.D V1, V3, F0}
Vector Load/Store Instructions

Load/Store a vector from memory into a vector register

Operates on contiguous addresses

\[ \text{LV } V1, R1 \ ; \ V1[i] = M[R1 + i] \]
\[ \text{SV } R1, V1 \ ; \ M[R1 + i] = V1[i] \]

Load/Store Vector with Stride

Vectors not always contiguous in memory

Add *non-unit stride* on each access

\[ \text{LVWS } V1, (R1, R2) \ ; \ V1[i] = M[R1 + i \times R2] \]
\[ \text{SVWS } (R1, R2), V1 \ ; \ M[R1 + i \times R2] = V1[i] \]

Vector Load/Store Indexed

Indirect accesses through an index vector

\[ \text{LVI } V1, (R1+V2) \ ; \ V1[i] = M[R1 + V2[i]] \]
\[ \text{SVI } (R1+V2), V1 \ ; \ M[R1 + V2[i]] = V1[i] \]
Double-precision $A \times X$ Plus $Y$ (DAXPY):

for $i = 0$ to $63$

\[ Y[i] = a \times X[i] + Y[i] \]

6 instructions instead of 600!

Remember: MIPS means “Meaningless Indicator of Performance”
Not All Vectors are 64 Elements Long

Vector length register (VLR)
Controls length of vector operations
0 < VLR ≤ MVL = 64

for (i=0; i<100; i++)
    X[i] = a * X[i]

LD         F0, a
MTC1       VLR, 36       /* 100 - 64 */
LV          V1, Rx
MULVS      V2, V1, F0
SV          Rx, V2
ADD         Rx, Rx, 36
MTC1       VLR, 64
LV          V1, Rx
MULVS      V2, V1, F0
SV          Rx, V2

Strip Mining for i = 1, n
Strip Mining

General case: Parameter n

```c
for (i=0; i<n; i++)
    X[i] = a * X[i]
```

Strip-mined version (pseudocode)

```c
low = 0
VL = (n mod MVL) /* Odd sized piece */
for (j = 0; j < (n / MVL); j++) { /* Outer loop */
    for (i = low, i < low+VL; i++) /* Length */
        X[i] = a * X[i]
    low = low + VL /* Base of next chunk */
    VL = MVL /* Reset length to MAX */
}
```
Old Vector Machines Did Not Have Caches

Caches

- Vectorizable codes often have poor locality
  - Large vectors don't fit in cache
  - Large vectors flush other data from the cache
- Cannot exploit known access patterns
- Unpredictability hurts
  - Degrades cycle time

Vector Registers (like all registers)

- Very fast
  - Predictable
  - Short id
  - Multiple ports easier
More Options

Use vector mask register for vectorizing

```c
for (i=0; i<64; i++)
    if (A[i] != 0.0) then A[i] = A[i] + 5.0
```

Use chaining (vector register bypass) for RAWs

```assembly
MULTV V1, ,
ADDV , V1,
```

Use gather/scatter for sparse matrices

```c
for (i=0; i<64; i++)
    A[K[i]] = A[K[i]] + C[M[i]]
```

Use multiple lanes for parallelism: implementation

FINAL WARNING: Make scalar unit fast!

Amdahl's law

CRAY1 was the fastest scalar computer
Compiler Technology

Must detect vectorizable loops
Must detect dependences that prevent vectorization
  Data, anti, output dependences
  Only data (or true) dependences important, others can be eliminated with renaming
**SIMD (Multimedia) Instructions**

Multimedia data derived from sampling analog input

Correctness dictated by human perception

Smaller data types - 8-bit, 16-bit

Compare with 32, 64, 128 bit processor data paths

Significant levels of data parallelism

Large collection of small data elements

Identical processing of similar elements

- e.g. Image Addition

```
For I = 1 to 1024
For J = 1 to 1024
    dest[I, J] = src1[I, J] + src2[I, J]
```
Packed Data Types

16 bits

Operand 1
Operand 2
Result

48 bits are wasted!

Can we use them in any way?

64 bits

Operand 1
Operand 2
Result

16 bits

4 operations in 1 cycle
SPEEDUP: 4X??

64 bits
Other Extensions

Saturation arithmetic

Example: image addition

For I = 1 to 1024
For J = 1 to 1024
    dest[I,J] = src1[I,J]+src2[I,J]
    If (dest > 255)
        dest = 255;
    If (dest < 0)
        dest = 0;
Saturation ensures clamping of values
Sub-word Rearrangement

How do we go from unpacked data types to packed data types?

Provide ISA support for pack, unpack, expand, align, ...

Support for other types of sub-word rearrangement

Shift, rotate, permute, ...

E.g., for FFT butterfly algorithm

Many others

Conditional execution, memory instructions, special-purpose instructions, ...

Most processors today support such instructions

ML acceleration with quantization is recent example
### Example: Intel MMX ISA Extensions (~1996)

<table>
<thead>
<tr>
<th>Category</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>PADD[B,W,D], PADDS[B,W], PADDUS[B,W], PSUB[B,W,D], PSUBS[B,W,D], PSUBUS[B,W], PMULHW, PMULLW, PMADDWD</td>
</tr>
<tr>
<td>Comparison</td>
<td>PCMPEQ[B,W,D], PCMPGT[B,W,D]</td>
</tr>
<tr>
<td>Conversion</td>
<td>PACKUSWB, PACKSS[WB,DW], PUNPCKH[BW,WD,DQ], PUNPCKL[BW,WD,DQ]</td>
</tr>
<tr>
<td>Logical</td>
<td>PAND, PANDN, POR, PXOR</td>
</tr>
<tr>
<td>Shift</td>
<td>PSLL[W,D,Q], PSRL[W,D,Q], PSRA[W,D]</td>
</tr>
<tr>
<td>FP and MMX state mgt</td>
<td>EMMS</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>MOV[D,Q]</td>
</tr>
</tbody>
</table>

57 new instructions

Use FP registers, 32-bit data path, SIMD, saturation, ...
### Example: Intel SSE ISA Extensions (~1999)

<table>
<thead>
<tr>
<th>Category</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data movement</td>
<td>MOV, MOVUPS, MOVLPS, MOVLHPS, MOVHPS, MOVHLPS, MOVMSKPS, MOVSS</td>
</tr>
<tr>
<td>Shuffle</td>
<td>SHUFPS, UNPCKHPS, UNPCKLPS</td>
</tr>
<tr>
<td>State</td>
<td>FXSAVE, FXRSTOR, STMXCSR, LDMXCSR</td>
</tr>
<tr>
<td>MMX Tech Enhancements</td>
<td>PINSRW, PEXTRW, PMULXHU, PSHUFW, PMOVMSKRB, PSAD, PAVG, PMIN, PMAX</td>
</tr>
<tr>
<td>Streaming/prefetching</td>
<td>MASKMOVQ, MOVNTQ, MOVTPS, PREFETCH, SFENCE</td>
</tr>
<tr>
<td>Conversions</td>
<td>CVTSS2SI, CVTTSS2SI, CVTSI2SS, CVTPI2PS, CVTPS2PI, CVTTPS2PI</td>
</tr>
</tbody>
</table>

70 instructions

Separate register state, 128-bit data path
Later Versions

2001/04/07: SSE2/3/4: double precision floating point, instructions to accelerate specific functions

2010: Advanced vector extensions (AVX)

- 256 bits, three operands
- Relaxed alignment
- Fused multiply-add (FMA) \((A=A\times B+C)\)

AVX-512: 512 bits

AVX-VNNI (Vector Neural Network Instructions)