Data Parallel Architectures - SIMD

Motivation

Vectors

Multimedia SIMD

GPUs

Motivation

Recall SIMD from Chapter 5

Vector Processors

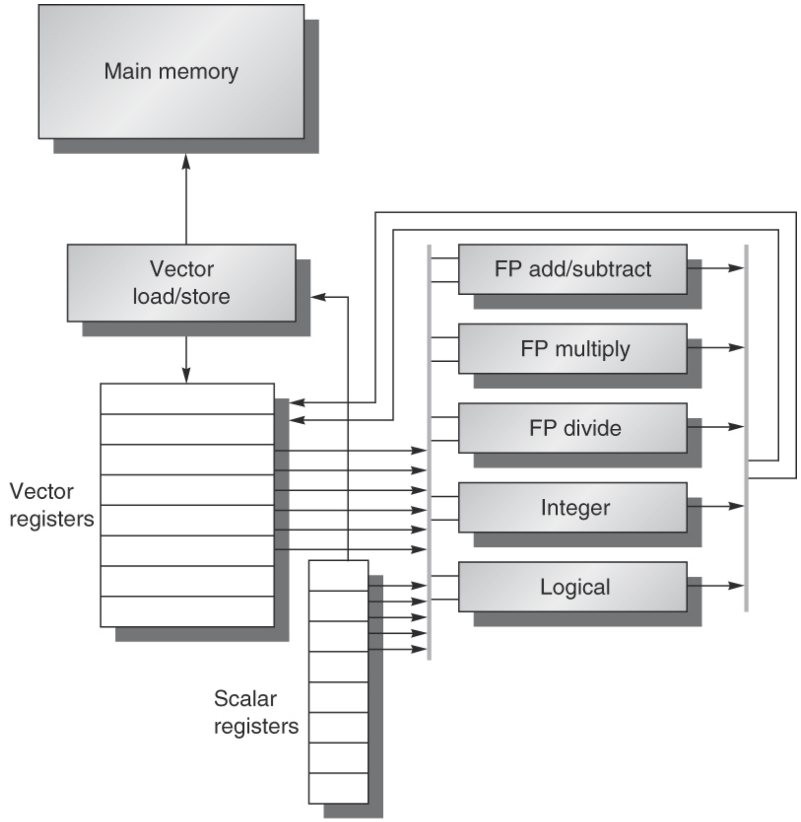

Figure 4.2 The basic structure of a vector architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and 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.

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, 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

*Why Vector Instructions?***

Want deeper pipelines, BUT

- Interlock logic complexity grows
- Stalls due to data hazards increase
- Stalls due to control hazards increase
- Instrn 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

Vector Architectures

Vector-Register Machines

- Load/store architecture
- All vector operations use registers (except load/store)
- Multiple ports are cheaper
- Optimized for small vectors

Memory-Memory Vector Machines

- All vectors reside in memory
- Long startup latency
- Multiple ports are expensive
- Optimized for long vectors

Often vectors are short

- Early machines were memory-memory (TI ASC, CDC STAR)
- Later machines use vector registers

VMIPS Architecture

- Strongly based on Cray
- 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

VMIPS Architecture, cont.

Vector-Vector Instructions

- Operate on two vectors
- Produce a third vector

```
for (i=0; i<64; i++)
      V1[i] = V2[i] + V3[i]
```

```
ADDVV.D V1, V2, V3
```

Vector-Scalar Instructions

- Operate on one vector, one scalar
- Produce a third vector

```
for (i=0; i<64; i++)
      V1[i] = F0 + V3[i]
```

```
ADDVS.D V1, V3, F0
```

VMIPS Architecture, cont.

Vector Load/Store Instructions

- Load/Store a vector from memory into a vector register
- Operates on contiguous addresses

```
LV V1, R1 ; V1[i] = M[R1 + i]
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

```
LVWS V1, (R1, R2) ; V1[i] = M[R1 + i*R2]
SVWS (R1, R2), V1 ; M[R1 + i*R2] = V1[i]
```

Vector Load/Store Indexed

- Indirect accesses through an index vector

```
LVI V1, (R1+V2) ; V1[i] = M[R1 + V2[i]]
SVI (R1+V2), V1 ; M[R1 + V2[i]] = V1[i]
```

VMIPS Architecture, cont.

Double-precision A*X Plus Y (DAXPY):

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

```
L.D       F0, a
LV        V1, Rx
MULVS.D   V2, V1, F0
LV        V3, Ry
ADDVV.D   V4, V2, V3
SV        Ry, V4
```

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
- $0 < VLR \leq MVL = 64$

```
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

```
    DO 10 I = 1, n
          X(i) = a * X(i)
10  CONTINUE
```

Strip-mined version (pseudocode)

```
    low = 1
    VL = (n mod MVL) /* Odd sized piece */
    DO 1 j = 0, (n / MVL) /* Outer loop */
       DO 10 i = low, low+VL1 /* Length */
          X(i) = a * X(i)
10     CONTINUE
       low = low + VL /* Base of next chunk */
       VL = MVL /* Reset length to MAX */
1   CONTINUE
```

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

```
    DO 10 i = 1, 64
       if A(i) ! 0.0 then A(i) = A(i)
 10 CONTINUE
```

Use chaining (vector register bypass) for RAWs

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

Use gather/scatter for sparse matrices

```
    DO 10 i = 1, 64
       A(K(i)) = A(K(i)) + C(M(i))
 10 CONTINUE
```

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