TEXAS TECH UNIVERSITY
Department *of* Computer Science

CS5375 Computer Systems Organization and Architecture

Lecture 15

Instructor: Yong Chen, Ph.D.
Department of Computer Science
Texas Tech University
Yong.Chen@ttu.edu, 806-834-0284

Announcements

- Programming project #1 due date extended till 10/29, Saturday, and no further extension will be provided
- Midterm exam will be graded by 10/29, Saturday
- Guest lectures by Mr. Ghazanfar Ali and Mr. Mert Side for GPU (Graphics Processing Units) architecture and programming

Outline

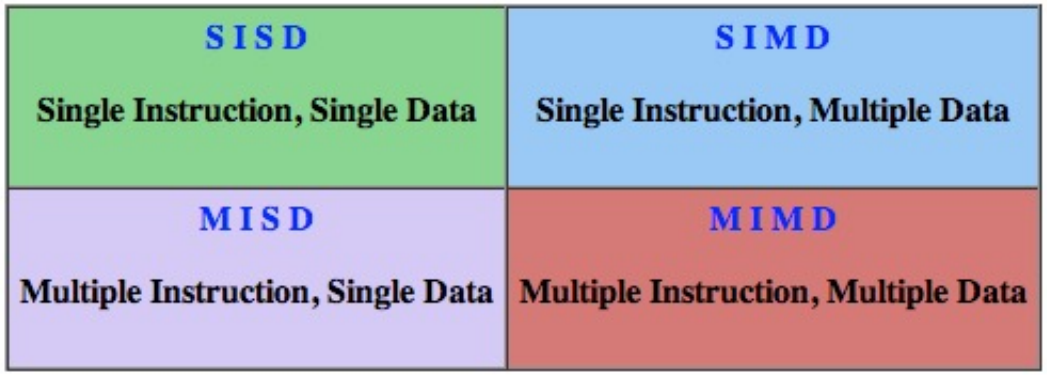
- Data-Level Parallelism in Vector, SIMD, and GPU architectures
- Vector Architecture

Introduction

- Data-Level Parallelism (DLP): identical operations operate on different data items concurrently to solve a problem

```
for (i=0; i<1000; i++)
    a[i]=b[i]+c[i];
```

- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix/vector-oriented scientific computing
  - Media-oriented image and sound processors



Courtesy: Blaise Barney, LLNL

Introduction (cont.)

- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for accelerators, personal mobile devices, etc.

- SIMD allows programmer to continue to think sequentially
  - V.S. task/thread-level parallelism, where often time programmers need to explicitly manage parallelism/concurrency
  - E.g., pthreads programming model, MPI (message-passing interface) programming model

SIMD Parallelism

- Three variations of SIMD:
- Vector architectures
- SIMD extensions
  - E.g. MMX (multimedia extensions), SSE (Streaming SIMD Extensions), and AVX (Advanced Vector Extensions) for x86 architectures
- Graphics Processor Units (GPUs)

Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory

Vector Architecture Example: RV64V

- RV64V: RISC-V vector instruction set extension (still under development)
  - Loosely based on Cray-1
  - Vector registers
    - Each vector register holds a single vector, and 32 64-bit vector registers
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 31 general-purpose registers
    - 32 floating-point registers

Vector Architecture Example: RV64V (cont.)

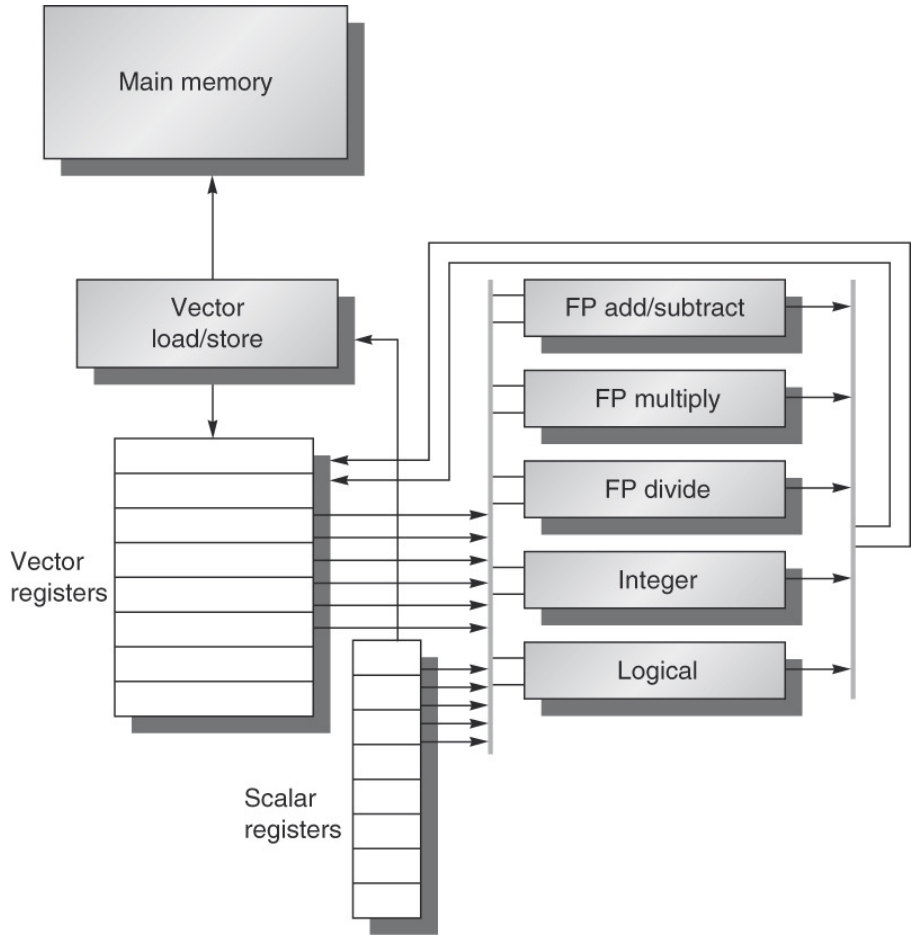


Figure 4.1 The basic structure of a vector architecture, RV64V, which includes a RISC-V scalar architecture.

RV64V Vector Instructions

- Addition/subtraction:

- `vadd  v3, v1, v2` # Vector-vector add, v3 = v1 + v2

- `vadd  v3, r1, v2` # Vector-scalar add, v3 = r1 + v2 (r1 is a scalar value)

- `vsub  v3, v1, v2` # Vector-vector subtract, v3 = v1 - v2

RV64V Vector Instructions (cont.)

- Load/store vectors:
- `vld v1, r1`: vector load from memory starting at address r1
- `vst r1, v1`: store vector into memory starting at address r1

RV64V Vector Instructions (cont.)

- Other sample instructions
- `vmul` # Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
- `vdiv` # Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
- `vsll` # Left Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
- `vsrl` # Right Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
- `vxor` # Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
- `vor` # Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
- `vand` # Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
- `vpeq` # Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
- `vpne` # Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
- `vplt` # Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1- bit element of p[rd]; otherwise, put 0
- …

How Vector Processors Work: An Example

- Example: SAXPY/DAXPY
  - "single-precision/double-precision *a* * X plus Y", i.e. to calculate Y = *a* * X + Y, where X and Y are vectors and *a* is a scalar

- Assume that X and Y have 32 elements and the starting addresses of X and Y are in x5 and x6, respectively

- RISC-V code without vectors

```
        fld f0,a # Load scalar a
        addi x28,x5,#256 # Last address to load
Loop:   fld f1,0(x5) # Load X[i]
        fmul.d f1,f1,f0 # a   X[i]
        fld f2,0(x6) # Load Y[i]
        fadd.d f2,f2,f1 # a   X[i] + Y[i]
        fsd f2,0(x6) # Store into Y[i]
        addi x5,x5,#8 # Increment index to X
        addi x6,x6,#8 # Increment index to Y
        bne x28,x5,Loop # Check if done
```

How Vector Processors Work: An Example (cont.)

- RV64V code for DAXPY:

```
vsetdcfg 4*FP64 # Enable 4 DP FP vregs
fld f0,a # Load scalar a
vld v0,x5 # Load vector X
vmul v1,v0,f0 # Vector-scalar mult
vld v2,x6 # Load vector Y
vadd v3,v1,v2 # Vector-vector add
vst v3,x6 # Store the sum
vdisable # Disable vector regs
```

- Execution of 8 instructions v.s. execution of 258 instructions (without vectors)

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

- RV64V functional units consume one element per clock cycle
  - Execution time is approximately the vector length

Multiple Lanes

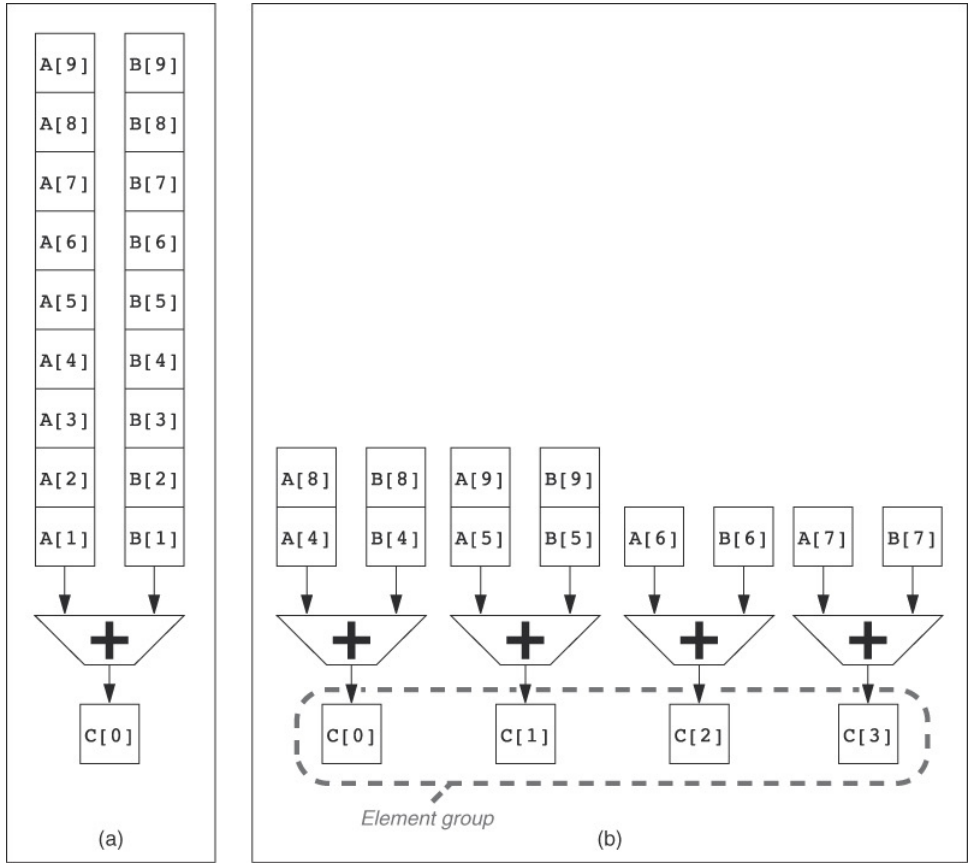


Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines and can complete four additions per cycle.

Multiple Lanes (cont.)

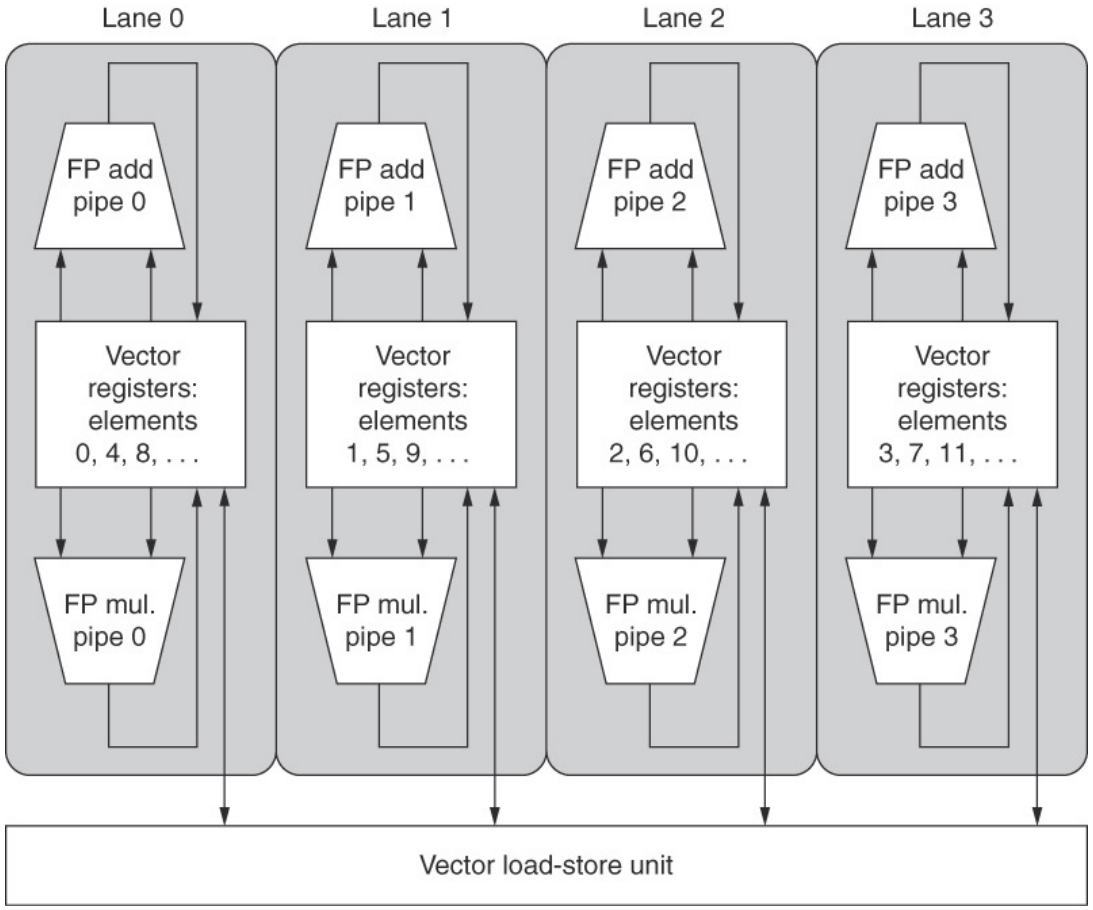
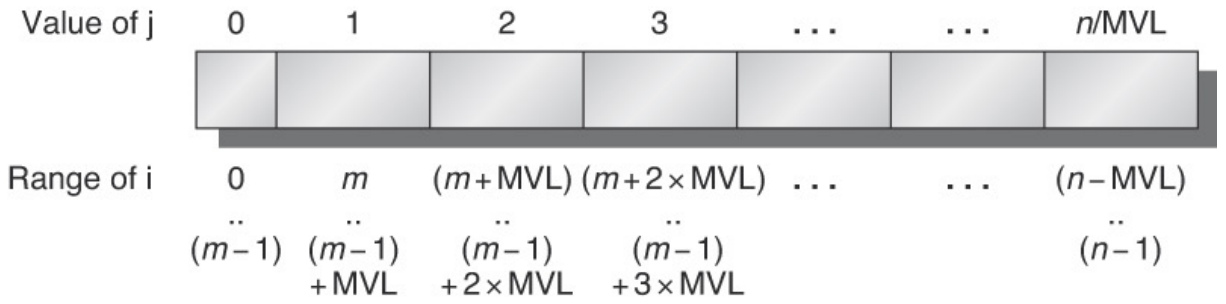


Figure 4.5 Structure of a vector unit containing four lanes.

Vector Length Registers: Handling Loops Not Equal to 32

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use strip mining for vectors over the maximum length:

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
   for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
     Y[i] = a * X[i] + Y[i] ; /*main operation*/
   low = low + VL; /*start of next vector*/
   VL = MVL; /*reset the length to maximum vector length*/
}
```



m = n % MVL

Predicate/Vector Mask Registers: Handling IF Statements in Vector Loops

- Consider:

```
for (i = 0; i < 64; i=i+1)
 if (X[i] != 0)
    X[i] = X[i] – Y[i];
```

Cannot normally be vectorized because of the conditional execution of the body

- Use predicate/vector mask register to “disable” elements:

```
vsetdcfg 2*FP64 # Enable 2 64b FP vector regs
vsetpcfgi 1 # Enable 1 predicate register
vld v0,x5 # Load vector X into v0
vld v1,x6 # Load vector Y into v1
fmv.d.x f0,x0 # Put (FP) zero into f0
vpne p0,v0,f0 # Set p0(i) to 1 if v0(i)!=f0
vsub v0,v0,v1 # Subtract under vector mask
vst v0,x5 # Store the result in X
vdisable # Disable vector registers
vpdisable # Disable predicate registers
```

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores

- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory

Scatter-Gather

- Handling sparse arrays or matrices (where many elements are zeros, and non-zero elements are sparse) in vector architectures
- Consider:

```
for (i = 0; i < n; i=i+1)
 A[K[i]] = A[K[i]] + C[M[i]];
```

- Use index vector:
  - vldx: load vector indexed, i.e. gather
  - vstx: store vector indexed, i.e. scatter

```
vsetdcfg   4*FP64     # 4 64b FP vector registers
vld        v0, x7     # Load K[]
vldx       v1, x5, v0# Load A[K[]]
vld        v2, x28    # Load M[]
vldx       v3, x6, v2# Load C[M[]]
vadd       v1, v1, v3# Add them
vstx       v1, x5, v0# Store A[K[]]
vdisable              # Disable vector registers
```

Readings

- Chapter 4, 4.1-4.2