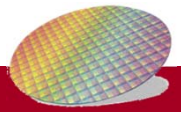
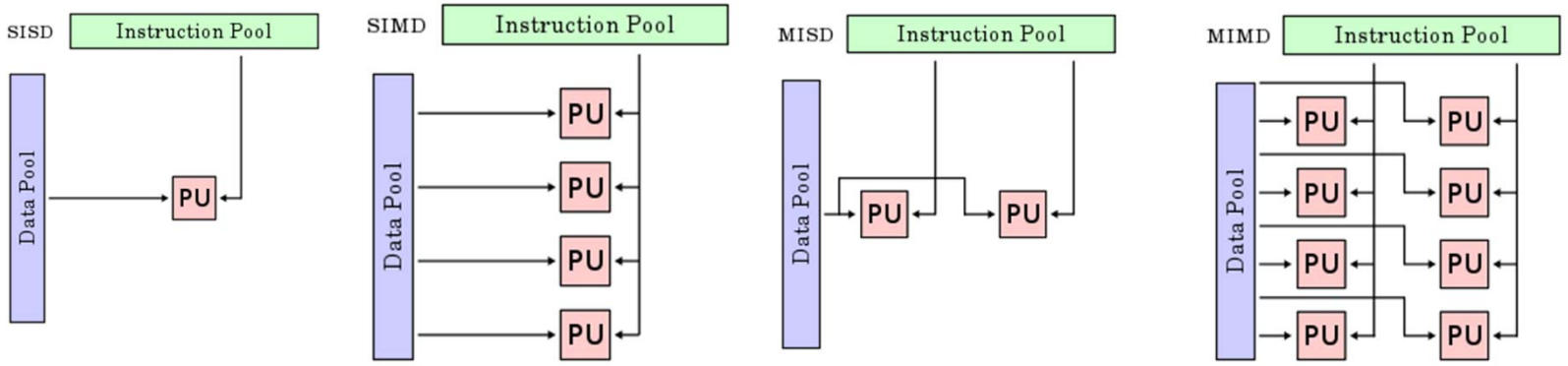
Data-Level Parallelism in Vector, SIMD, and GPU Architectures



Review: Flynn's Taxonomy

Proposed by Michael J. Flynn's (Stanford University Professor)

	Single Instruction Stream	Multiple Instruction Stream
Single Data Stream	SISD	MISD (No commercial implementation)
Multiple Data Stream	SIMD includes • Vector architectures • Multimedia extensions • Graphics processor units	MIMD • Tightly-coupled MIMD • Loosely-coupled MIMD



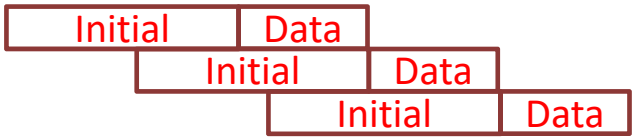
Source: Wiki

Introduction

- SIMD exploit significant data-level parallelism for
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
- SIMD can be more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially
  - Speedup can be achieved by parallel data operations
- Three architecture
  - Vector architectures: pipelined execution of may data operations
  - Multimedia SIMD instruction set extension: simultaneous parallel data operations
  - Graphics Processor Units (GPUs): similar to vector architecture, but include CPU, GPU, system memory, and graphic memory => heterogeneous computing

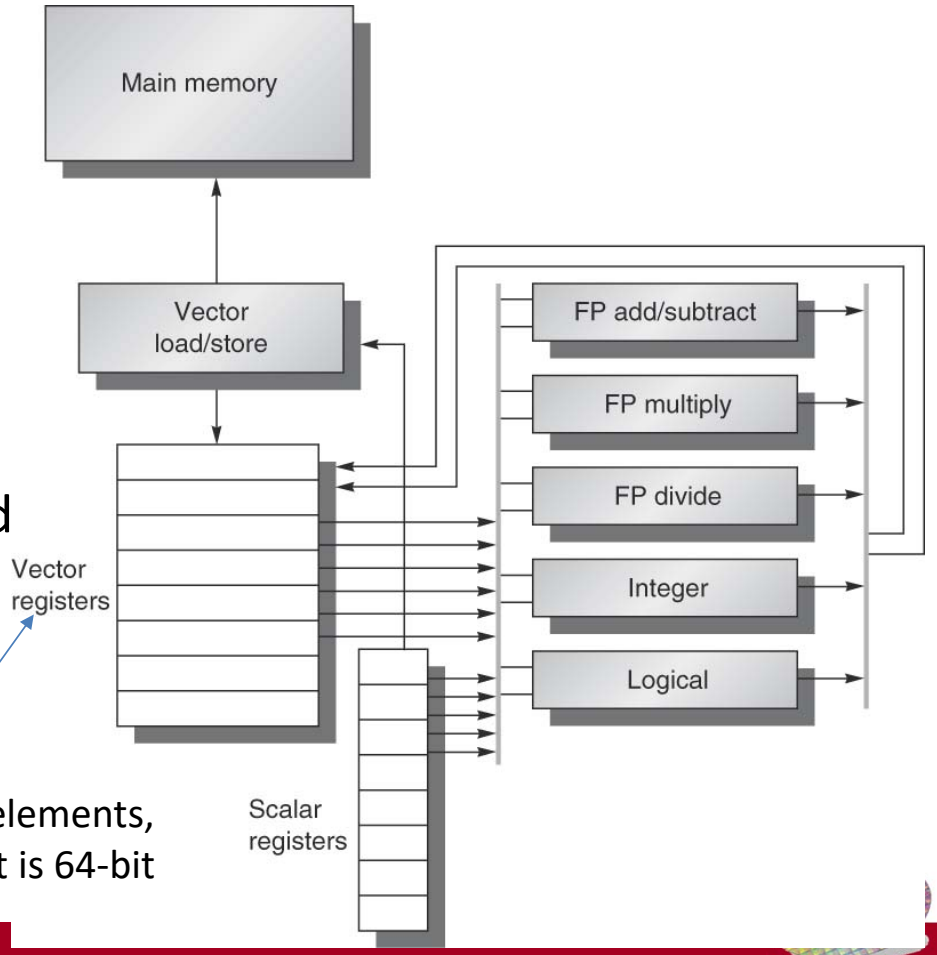
Vector Architectures

- Basic idea: pipelined execution of many data operations
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Write the results back into memory
- A single instruction on vector of data => results in dozens of register-register operations on independent data elements
- Registers are controlled by compiler
  - Used to hide memory latency because vector load and store are deeply pipelined
    - Pay long memory latency only once
  - Leverage memory bandwidth



Example: VMIPS

- Vector registers
  - Each register holds a 64-element, 64bits/element vector
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - Fully pipelined, start a new ops every cycle
  - Controller to detect data and control hazards (not shown)
- Vector load-store unit
  - Fully pipelined
  - One word per cycle to move between register and memory after initial latency
- Scalar registers
  - Provide input to functional unit
  - 32 GP and 32 FP registers



Example: DAXPY (double-precision a*X plus Y)

MIPS instructions for DAXPY. X, Y, and vector registers are all 64 bits

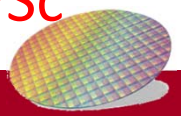
Y= a * X + Y

- X, Y are double precision, a is scalar
- A widely used execution in Linpack benchmark

```
        L.D    F0,a                ;load scalar a
        DADDIU  R4, Rx, #512       ;last address to load
Loop:   L.D  F2, 0(Rx)             ;load X[i]
        MUL.D F2, F2, F0           ; a * X[i]
        L.D  F4, 0(Ry)             ; load Y[i]
        ADD.D  F4, F4, F2          ;A * X[i] + Y[i]
        S.D   F4, F4, F2           ; store into Y[i]
        DADDIU  Rx, Rx, #8         ; increment index to X
        DADDIU  Ry, Ry, #8         ; increment index to Y
        DSUBU   R20, R4, Rx        ; compute bound
        BNEZ    R20, Loop          ; check if done
```

In MIPS Code, ADD waits for MUL, SD waits for ADD

almost 600 instructions for MIPSc



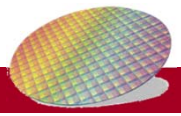
VMIPS Instructions for DAXPY

ADDVV.D V1, V2, V3: add two vectors V2 & V3 and V1 stores result
ADDVS.D V1, V2, F0: add F0 to each element of V2 and V1 stores result
LV V1, R1: load V1 from memory starting at address R1
SV V1 R1: store vector V1 to memory starting at address R1

```
L.D       F0,a       ; load scalar a
LV        V1, Rx     ; load vector X
MULVS.D   V2, V1, F0 ; vector-scalar multiply
LV        V3, Ry     ; load vector Y
ADDVV     V4, V2, V3 ; add
SV        Ry, V4     ; store the result
```

- Requires 6 instructions vs. almost 600 for MIPS
  - Vector operations work on 64 elements
- Overhead on loop are removed

In VMIPS
- Stall once for the first vector element, subsequent elements will flow smoothly down the pipeline.
- Pipeline stall required once per vector instruction!



Another Example for VMIPS Instructions

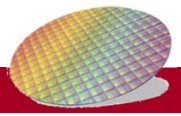
```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# Scalar Code
   LI     R4, 64
loop:
   L.D    F0, 0(R1)
   L.D    F2, 0(R2)
   ADD.D  F4, F2, F0
   S.D    F4, 0(R3)
   DADDIU R1, 8
   DADDIU R2, 8
   DADDIU R3, 8
   DSUBIU R4, 1
   BNEZ   R4, loop
```

```
# Vector Code
   LI      VLR, 64
   LV      V1, R1
   LV      V2, R2
   ADDV.D  V3, V1, V2
   SV      V3, R3
```

Remarks in VMIPS

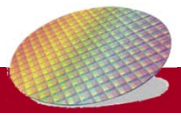
- VMIPS greatly reduces the number of instructions (almost 600 → 6)
  - A vector instruction works on 64 elements
- Operate on many elements concurrently
  - Allows use of slow but wide execution units
  - High performance
- No dependence within a vector
  - Pipeline, parallelization work well
  - Can have very deep pipeline
- Flexible
  - 64 64-bit / 128 32-bit / 256 16-bit, 512 8-bit
  - Matches the need of multimedia (16, 8bit), scientific applications that require high precision



Vector Execution Time

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

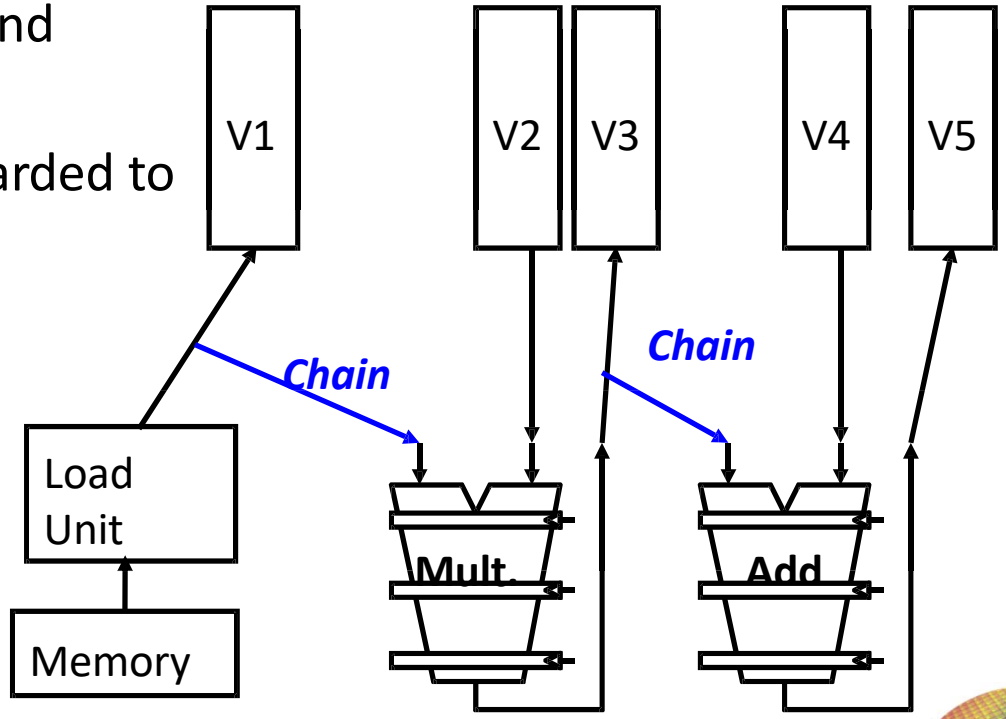
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length



Vector Chaining to improve execution time

- Chaining : Vector version of register bypassing
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
  - Results from the first functional unit are forwarded to the second unit

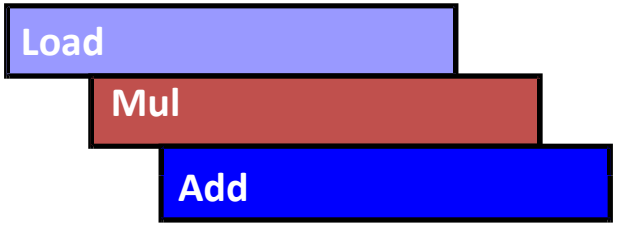
```
LV   v1
MULV v3, v1, v2
ADDV v5, v3, v4
```



Vector Chaining Advantage

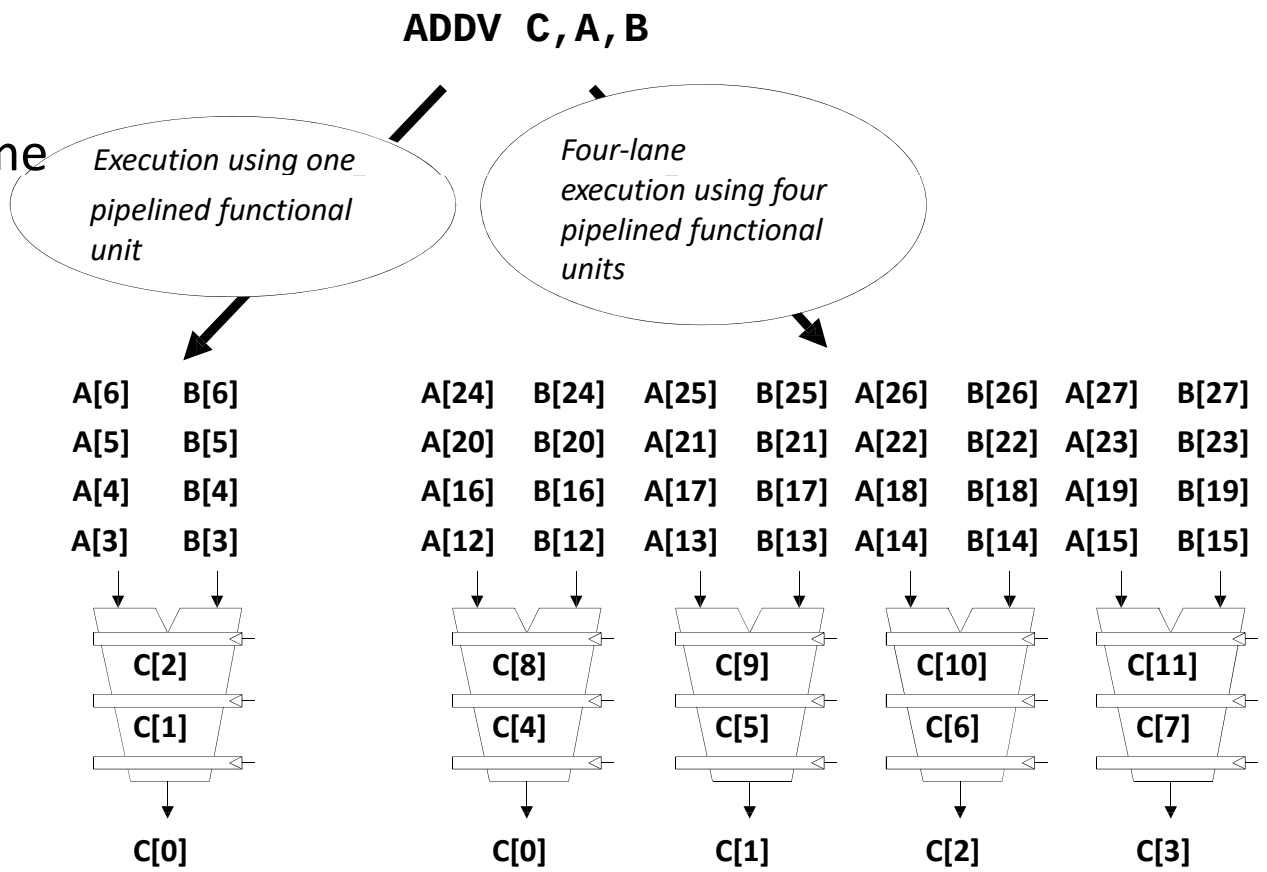
- Without chaining, must wait for last element of result to be written before starting dependent instruction

- With chaining, can start dependent instruction as soon as first result appears



Multiple Lanes to improve execution time -1

Idea: using multiple function unit to reduce execution time

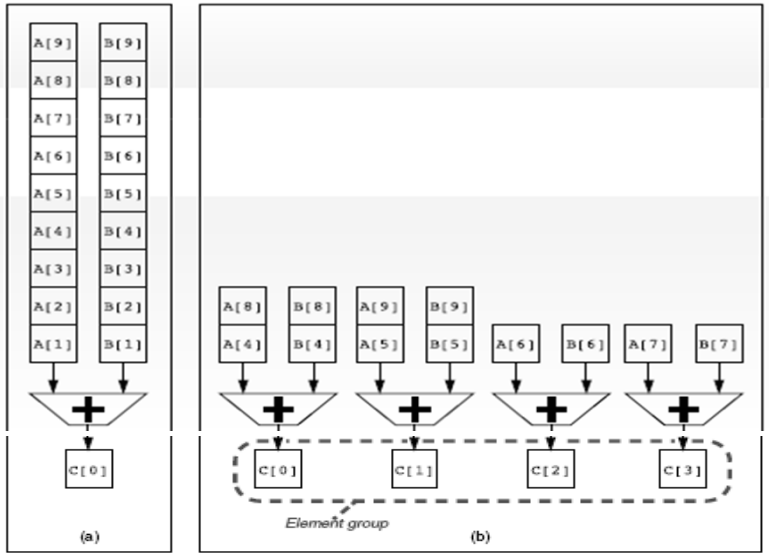


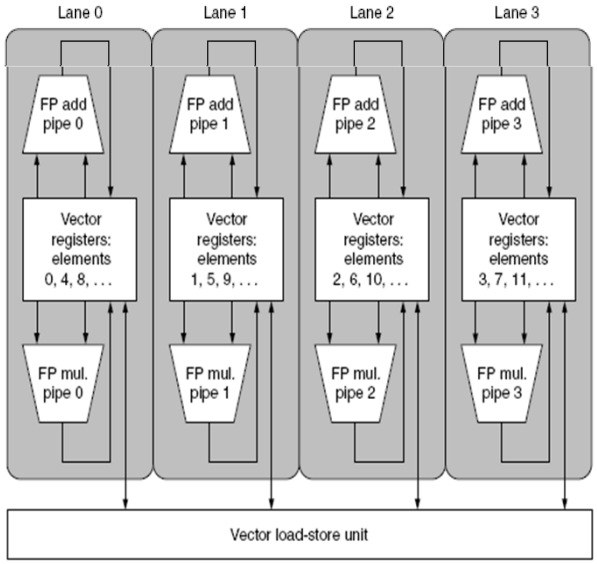
Multiple Lanes to improve execution time -2

Vector elements are divided across the lane

Element n of vector register A is “hardwired” to element n of vector register B

- Allows for multiple hardware lanes
- No communication between lanes
- Little increase in control overhead
- No need to change machine code

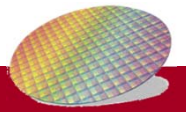




Adding more lanes allows designers to tradeoff clock rate and energy without sacrificing performance!

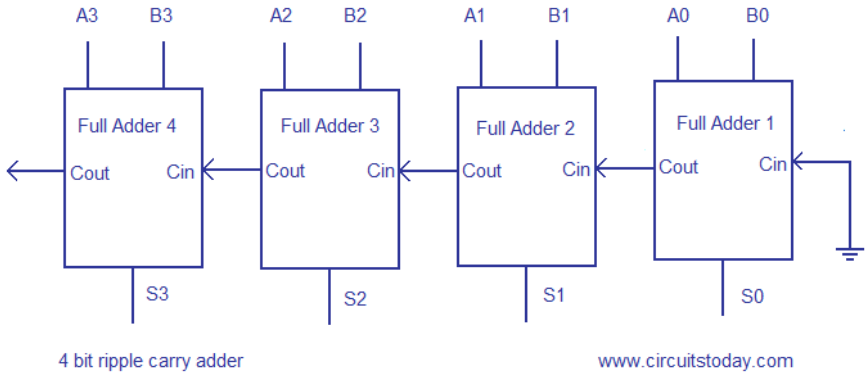
Vector Summary

- Vector is alternative model for exploiting ILP
  - If code is vectorizable, then simpler hardware, energy efficient, and better real-time model than out-of-order
  - More lanes, slower clock rate!
  - Scalable if elements are independent
  - If there is dependency
    - One stall per vector instruction rather than one stall per vector element
- Programmer in charge of giving hints to the compiler!
- Design issues: number of lanes, functional units and registers, length of vector registers, exception handling, conditional operations



Multimedia SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Example: 8bits to represent a color in RGB
  - Example: 256-bit adder can be divided to 32 8-bit adders
    - disconnect carry chains to “partition” adder



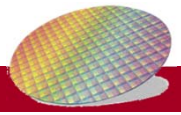
- SIMD extension is popular because additional cost to partition arithmetic circuit is small
  - Intel MMX,SSE, AVE (See next slides)

SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops

    128 bits total
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops

    256 bits total
  - Operands must be consecutive and aligned memory locations



Example: Pseudo SIMD Code for DAXPY

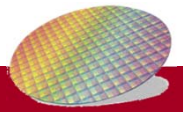
- Example DAXPY:

```
        L.D       F0,a          ;load scalar a
        MOV       F1, F0        ;copy a into F1 for SIMD MUL
        MOV       F2, F0        ;copy a into F2 for SIMD MUL
        MOV       F3, F0        ;copy a into F3 for SIMD MUL
        DADDIU    R4,Rx,#512    ;last address to load
Loop:   L.4D F4,0[Rx]           ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D    F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D      F8,0[Ry]      ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D    F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D      0[Ry],F8      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU    Rx,Rx,#32;increment index to X
        DADDIU    Ry,Ry,#32 ;increment index to Y
        DSUBU     R20,R4,Rx     ;compute bound
        BNEZ  R20,Loop ;check if done
```

- Number of data operands encoded into op code

149 instructions

- SIMD extension also specify operation on vector data,
- But SIMD extension specifies
  - fewer operands
  - use smaller register files
  - Number of data operands encode into op code



Backup slides