6.3 SISD, MIMD, SIMD, SPMD, and vector

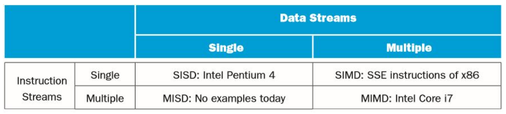
(Original section[1])

One categorization of parallel hardware proposed in the 1960s is still used today. It was based on the number of instruction streams and the number of data streams. The figure below shows the categories. Thus, a conventional uniprocessor has a single instruction stream and single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams. These two categories are abbreviated *SISD* and *MIMD*, respectively.

SISD or ***single instruction stream***, ***single data stream***: A uniprocessor.

MIMD or ***multiple instruction streams***, ***multiple data streams***: A multiprocessor.

Figure 6.3.1: Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD (COD Figure 6.2).



		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7

While it is possible to write separate programs that run on different processors on a MIMD computer and yet work together for a grander, coordinated goal, programmers normally write a single program that runs on all processors of a MIMD computer, relying on conditional statements when different processors should execute distinct sections of code. This style is called *Single Program Multiple Data (SPMD)*, but it is just the normal way to program a MIMD computer.

SPMD or ***single program, multiple data streams***: The conventional MIMD programming model, where a single program runs across all processors.

The closest we can come to multiple instruction streams and single data stream (*MISD*) processor might be a "stream processor" that would perform a series of computations on a single data stream in a pipelined fashion: parse the input from the network, decrypt the data, decompress it, search for match, and so on. The inverse of MISD is much more popular. *SIMD* computers operate on vectors of data. For example, a single SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALUs to form 64 sums within a single clock cycle. The subword parallel instructions that we saw in COD Sections 3.6 (Parallelism and computer arithmetic: Subword parallelism) and 3.7 (Real stuff: Streaming SIMD extensions and advanced vector extensions in x86) are another example of SIMD; indeed, the middle letter of Intel's SSE acronym stands for SIMD.

SIMD or ***single instruction stream, multiple data streams***: The same instruction is applied to many data streams, as in a vector processor.

The virtues of SIMD are that all the parallel execution units are synchronized, and they all respond to a single instruction that emanates from a single *program counter* (PC). From a programmer's perspective, this is close to the already familiar SISD. Although every unit will be executing the same instruction, each execution unit has its own address registers, and so each unit can have different data addresses. Thus, in terms of COD Figure 6.1 (Hardware/software categorization ...), a sequential application might be compiled to run on serial hardware organized as a SISD or in parallel hardware that was organized as a SIMD.

The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units. Another advantage is the reduced instruction bandwidth and space—SIMD needs only one copy of the code that is being simultaneously executed, while message-passing MIMDs may need a copy in every processor and shared memory MIMD will need multiple instruction caches.

SIMD works best when dealing with arrays in `for` loops. Hence, for parallelism to work in SIMD, there must be a great deal of identically structured data, which is called *data-level parallelism*. SIMD is at its weakest in `case` or `switch` statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data must be disabled so that units with proper data may continue. If there are *n* cases, in these situations, SIMD processors essentially run at 1/*n*th of peak performance.

Data-level parallelism: Parallelism achieved by performing the same operation on independent data.

The so-called array processors that inspired the SIMD category have faded into history (see COD Section 6.15 (Historical perspective and further reading)), but two current interpretations of SIMD remain active today.

SIMD in x86: Multimedia extensions

As described in COD Chapter 3 (Arithmetic for Computers), subword parallelism for narrow integer data was the original inspiration of the *Multimedia Extension* (MMX) instructions of the x86 in 1996. As **Moore's Law** continued, more instructions were added, leading first to *Streaming SIMD Extensions* (SSE) and now *Advanced Vector Extensions* (AVX). AVX supports the simultaneous execution of four 64-bit floating-point numbers. The width of the operation and the registers is encoded in the opcode of these multimedia instructions. As the data width of the registers and operations grew, the number of opcodes for multimedia instructions exploded, and now there are hundreds of SSE and AVX instructions (see COD Chapter 3 (Arithmetic for Computers)).

Vector

An older and, as we shall see, more elegant interpretation of SIMD is called a *vector architecture*, which has been closely identified with computers designed by Seymour Cray starting in the 1970s. It is also a great match to problems with lots of data-level parallelism. Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost. The basic philosophy of vector architecture is to collect data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using **pipelined execution units**, and then write the results back to memory. A key feature of vector architectures is therefore a set of vector registers. Thus, a vector architecture might have 32 vector registers, each with 64 64-bit elements.

Example 6.3.1: Comparing vector to conventional code.

Suppose we extend the LEGv8 instruction set architecture with vector instructions and vector registers. Vector operations use the same names as LEGv8 operations, but with the letter "V" appended. For example, `FADDDV` adds two double-precision vectors. Let's also expand the 32 V registers used for SIMD instructions from two 64-bit elements to sixty-four 64-bit elements. The vector instructions take as their input either a pair of vector (V) registers (`FADDDVS`) or a vector register and a scalar register (`FADDDVS`). In the latter case, the value in the scalar register is used as the input for all operations—the operation `FADDDVS` will add the contents of a scalar register to each element in a vector register. The names `LDURDV` and `STURDV` denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is the vector register to be loaded or stored; the other operand, which is a LEGv8 general-purpose register, is the starting address of the vector in memory.

Given this short description, show the conventional LEGv8 code versus the vector LEGv8 code for

$$\mathrm{Y} = a \times \mathrm{X} + \mathrm{Y}$$

where *X* and *Y* are vectors of 64 double precision floating-point numbers, initially resident in memory, and *a* is a scalar double precision variable. (This example is the so-called DAXPY loop that forms the inner loop of the Linpack benchmark; DAXPY stands for double precision *a x X* plus *Y*.). Assume that the starting addresses of *X* and *Y* are in `X19` and `X20`, respectively.

Answer

Here is the conventional LEGv8 code for DAXPY:

```
        LDURD    D0,[X28,a]       // load scalar a
        ADDI     X0,X19,512       // upper bound of what to load
loop:   LDURD    D2,[X19,#0]      // load x(i)
        FMULD    D2,D2,D0         // a x x(i)
        LDURD    D4,[X20,#0]      // load y(i)
        FADDD    D4,D4,D2         // a x x(i) + y(i)
        STURD    D4,[X20,#0]      // store into y(i)
        ADDI     X19,X19,#8       // increment index to x
        ADDI     X20,X20,#8       // increment index to y
        CMPB     X0,X19           // compute bound
        B.NE     loop             // check if done
```

Here is the vector LEGv8 code for DAXPY:

```
        LDURD    D0,[x28,a]        // load scalar a
        LDURDV   V1,[X19,#0]       // load vector x
        FMULDVS  V2,V1,D0          // vector-scalar multiply
        LDURDV   V3,[X20,#0]       // load vector y
        FADDDV   V4,V2,V3          // add y to product
        STURDV   V4,[X20,#0]       // store the result
```

There are some interesting comparisons between the two code segments in this example. The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only six instructions versus almost 600 for the traditional LEGv8 architecture. This reduction occurs both because the vector operations work on 64 elements at a time and because the overhead instructions that constitute nearly half the loop on LEGv8 are not present in the vector code. As you might expect, this reduction in instructions fetched and executed saves energy.

Another important difference is the frequency of **pipeline** hazards (COD Chapter 4 (The Processor)). In the straightforward LEGv8 code, every `FADDD` must wait for a `FMULD`, every `STURD` must wait for the `FADDD` and every `FADDD` *and* `FMULD` must wait on `LDURD`. On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector *operation*, rather than once per vector *element*. In this example, the pipeline stall frequency on LEGv8 will be about 64 times higher than it is on the vector version of LEGv8. The pipeline stalls can be reduced on LEGv8 by using loop unrolling (see COD Chapter 4 (The Processor)). However, the large difference in instruction bandwidth cannot be reduced.

Since the vector elements are independent, they can be operated on in parallel, much like subword parallelism for the Intel x86 AVX instructions. All modern vector computers have vector functional units with multiple parallel pipelines (called *vector lanes*; see COD Figures 6.2 (Hardware categorization and examples ...) and 6.3 (Using multiple functional units to improve the performance ...)) that can produce two or more results per clock cycle.

Elaboration

The loop in the example above exactly matched the vector length. When loops are shorter, vector architectures use a register that reduces the length of vector operations. When loops are larger, we add bookkeeping code to iterate full-length vector operations and to handle the leftovers. This latter process is called strip mining.

Vector versus scalar

Vector instructions have several important properties compared to conventional instruction set architectures, which are called *scalar architectures* in this context:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. The instruction fetch and decode bandwidth needed is dramatically reduced.
- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.
- Vector architectures and compilers have a reputation for making it much easier than when using MIMD multiprocessors to write efficient applications when they contain data-level parallelism.
- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save energy as well as time.

- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.
- Because a complete loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.
- The savings in instruction bandwidth and hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power and energy versus scalar architectures.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can often use them.

Vector versus multimedia extensions

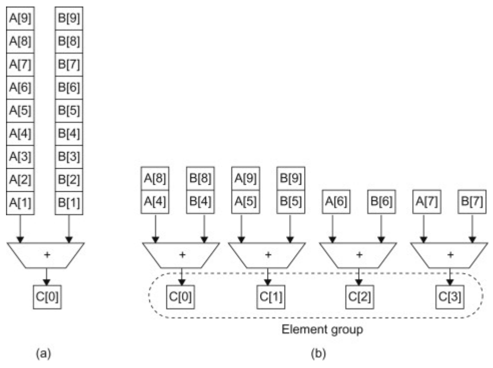
Like multimedia extensions found in the x86 AVX instructions, a vector instruction specifies multiple operations. However, multimedia extensions typically denote a few operations while vector specifies dozens of operations. Unlike multimedia extensions, the number of elements in a vector operation is not in the opcode but in a separate register. This distinction means different versions of the vector architecture can be implemented with a different number of elements just by changing the contents of that register and hence retain binary compatibility. In contrast, a new large set of opcodes is added each time the "vector" length changes in the multimedia extension architecture of the x86: MMX, SSE, SSE2, AVX, AVX2, ….

Also unlike multimedia extensions, the data transfers need not be contiguous. Vectors support both strided accesses, where the hardware loads every *n*th data element in memory, and indexed accesses, where hardware finds the addresses of the items to be loaded in a vector register. Indexed accesses are also called *gather-scatter*, in that indexed loads gather elements from main memory into contiguous vector elements and indexed stores scatter vector elements across main memory.

Like multimedia extensions, vector architectures easily capture the flexibility in data widths, so it is easy to make a vector operation work on 32 64-bit data elements or 64 32-bit data elements or 128 16-bit data elements or 256 8-bit data elements. The parallel semantics of a vector instruction allows an implementation to execute these operations using a deeply **pipelined** functional unit, an array of parallel functional units, or a combination of parallel and pipelined functional units. The figure below illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.

Figure 6.3.2: Using multiple functional units to improve the performance of a single vector add instruction, C = A + B (COD Figure 6.3).

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 or lanes and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four lanes.

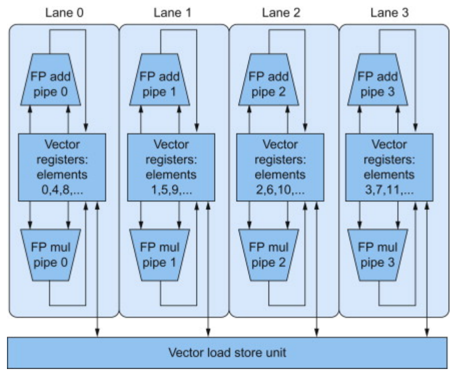

A[9] A[8] A[7] A[6] A[5] A[4] A[3] A[2] A[1]
B[9] B[8] B[7] B[6] B[5] B[4] B[3] B[2] B[1]
+
C[0]
(a)
A[8] A[4] B[8] B[4] A[9] A[5] B[9] B[5] A[6] B[6] A[7] B[7]
+
C[0] C[1] C[2] C[3]
Element group
(b)

Vector arithmetic instructions usually only allow element N of one vector register to take part in operations with element N from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel *vector lanes*. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. The figure below shows the structure of a four-lane vector unit. Thus, going to four lanes from one lane reduces the number of clocks per vector instruction by roughly a factor of four. For multiple lanes to be advantageous, both the applications and the architecture must support long vectors. Otherwise, they will execute so quickly that you'll run out of instructions, requiring instruction level **parallel** techniques like those in COD Chapter 4 (The Processor) to supply enough vector instructions.

Vector lane: One or more vector functional units and a portion of the vector register file. Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously.

Figure 6.3.3: Structure of a vector unit containing four lanes (COD Figure 6.4).

The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which acts in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough read and write ports (see COD Chapter 4 (The Processor)) for functional units local to its lane.



Generally, vector architectures are a very efficient way to execute data parallel processing programs; they are better matches to compiler technology than multimedia extensions; and they are easier to evolve over time than the multimedia extensions to the x86 architecture.

Given these classic categories, we next see how to exploit parallel streams of instructions to improve the performance of a *single* processor, which we will reuse with multiple processors.

Check yourself

True or false: As exemplified in the x86, multimedia extensions can be thought of as a vector architecture with short vectors that supports only contiguous vector data transfers.

Answer: True, but they are missing useful vector features like gather-scatter and vector length registers that improve the efficiency of vector architectures. (As an elaboration in this section mentions, the AVX2 SIMD extensions offers indexed loads via a gather operation but *not* scatter for indexed stores. The Haswell generation x86 microprocessor is the first to support AVX2.)

Elaboration

Given the advantages of vector, why aren't they more popular outside high-performance computing? There were concerns about the larger state for vector registers increasing context switch time and the difficulty of handling page faults in vector loads and stores, and SIMD instructions achieved some of the benefits of vector instructions. In addition, as long as advances in instruction-level parallelism could deliver on the performance promise of ***Moore's Law****, there was little reason to take the chance of changing architecture styles.*

Elaboration

Another advantage of vector and multimedia extensions is that it is relatively easy to extend a scalar instruction set architecture with these instructions to improve performance of data parallel operations.

Elaboration

The Haswell-generation x86 processors from Intel support AVX2, which has a gather operation but not a scatter operation.

(*1) This section is in original form.

Provide feedback on this section