

Vector Processing

Prof. Loh

CS3220 - Processor Design - Spring 2005

April 5, 2005

1 Computer Taxonomies

There are many possible ways to describe computer systems, and many different *taxonomies* or categorization systems have been proposed. The most common is due to Flynn, called Flynn's Taxonomy. This system divides machines up depending on (1) how many instruction streams there are and (2) how many data streams there are. By instruction streams, we mean separate and independent paths of control and state, that is an instruction stream has its own program counter, and its own registers. By data streams, we mean independent pieces of data. A vector of multiple pieces of data may comprise multiple data streams, and independent memory values that are handled concurrently may also comprise separate data streams. Each of the two attributes, **I**nstructions and **D**ata, may be classified as **S**ingle or **M**ultiple. This yields four possible combinations:

| Data Streams | Instruction Streams | |
|--------------|---------------------|----------|
| | Single | Multiple |
| Single | SISD | SIMD |
| Multiple | MISD | MIMD |

- **Single Instruction Single Data (SISD):** A single stream of instructions executes on one stream of data at a time. This includes the traditional CISC CPUs, pipelined CPUs and also superscalar CPUs (because the parallelism is hidden from the ISA).
- **Single Instruction Multiple Data (SIMD):** A single stream of instructions may operate on multiple pieces of data in parallel. Typical examples include vector architectures.
- **Multiple Instruction Single Data (MISD):** Almost no real examples fall into this category.
- **Multiple Instruction Multiple Data (MIMD):** Many independent instruction streams operate on separate sets of data concurrently. Examples include massively parallel processors ("supercomputers", such as those used for weather modelling), shared memory SMP machines (like the dual processor Xeon boxes you could buy yourself), or the upcoming dual-core processors (two cpus in on one chip).

Note that these are datastreams that are exposed to the programmer. Someone writing a program for a MIMD machine must explicitly write more than one *thread* of execution (instruction stream) and make sure that the different data handled in parallel are done so correctly. In a superscalar processor, multiple instructions may execute concurrently working on distinct sets of data, but all of this occurs "beneath the covers" and is something that the programmer can not see.

Why do we bother about taxonomies? The primary interest is that the categorization of systems can help understand many of the pertinent issues of systems. Not just only in computer systems, but taxonomies in general assist in evaluating different attributes in an organized manner which may shed some additional insight on tradeoffs or relative importance of different parameters.

Flynn's taxonomy is perhaps one of the most well known, and therefore commonly used. This is probably due to its simplicity. Unfortunately, it is often a little too coarse, and so other refinements or extensions to Flynn's taxonomy

have been introduced over time. This lecture we focused on SIMD systems, starting with one of the grand-fathers, the Cray-1 supercomputer, and then looking at MMX/3DNow!/SSE which are some of the more modern reincarnations of the SIMD technology that was used back in the 70's for the Cray.

2 Vector Processors

The pipelined and superscalar processors that we have been covering in this course are all *uniprocessors*, which are processors that have only a single *thread* of execution or simply a single program counter. Vector processors are also uniprocessors, and are in many way similar to more traditional SISD machines. There is a single program counter, and there are typically registers and functional units and so on. The difference is that in a vector machine, there are typically additional *vector registers*, *vector functional units*, and *vector instructions* to manipulate the vector data.

A vector register may be composed of a large number of smaller *scalar* registers. Each scalar register is an independent number, traditionally interpreted as a floating point value (you can interpret it however you want since it's all just a bunch of bits). Vector computations are more common for scientific computing where you may be modelling something or performing some mathematical computation (over continuous variables and functions), and so that is why floating point is typically used. High throughput (pipelined) vector functional units are used to process data from the vector registers at a high rate. To command the computer to perform these operations, special vector instructions must be used.

A typical vector operation would be:

$$\vec{A} = \vec{B} + \vec{C}$$

In a non-vector machine, such code would have to be compiled into a loop that iteratively adds the components of the vectors together in a point-wise fashion. In a vector machine, this is accomplished with a single instruction:

| | | | |
|-----|-------------------|-------------------|-------------------|
| VOP | V _{dest} | V _{arg1} | V _{arg2} |
|-----|-------------------|-------------------|-------------------|

Where VOP specifies a vector operation (such as add or multiply), and the V_{*} arguments name vector registers. The advantages of vector instructions are:

- Improved code density (more “work” fits in cache)
- Reduce number of instructions needed to execute the program
- Data are organized into regular patterns that can be efficiently handled by the hardware
- Simple loop constructs can be replaced with a few vector instructions, thus eliminating the loop overhead.

Since a single (or a few) instructions can replace the entire body and control overhead of a loop, the number of instructions needed to specify how to perform a computation is potentially reduced. This increase in code density (roughly the number of instructions to perform some task or work) allows more useful “work” to fit inside the cache which results in fewer cache misses.

Holding all other factors constant, reducing the number of instructions needed for a computation will decrease the runtime of the program (increase performance). Naturally all other factors are not exactly “constant”, but this can still help.

The vector organization of data is very helpful since it implies that there are no data dependencies when instructions operate on the data. This means that the hardware can be much simpler than the superscalar logic for checking resource availability, inter-instruction dependencies, pairing rules, etc., while achieving a fair amount of parallelism.

We saw that the control dependencies due to loops can cause degradations in performance due to interruptions in fetching instructions and the need for branch prediction. Furthermore, in a superscalar processor, the parallelism is hampered by the fact that the instructions can only be fetched into the core of the processor at a limited rate. A loop that is replaced by a few vector instructions basically allows every iteration of the loop to be fetched at the same time, thus exposing a large number of parallel operations that can be concurrently executed.

The downsides of vector processing are:

- Extensions/modifications to the instruction set architecture
- Extensions to the functional units and register files
- Extensions to the memory system

Either the processor (or ISA) needs to be defined from the get-go to support vector operations and data types, or an existing ISA can be extended to include such vector support. In either case, existing code will either not be able to run on this vector processor, or will see no benefit from the introduction of support for vector operations. One of the selling points of superscalar processors is that the microarchitecture may speed up existing code (although in many instances, a recompilation with a architecture aware compiler is still necessary to *maximize* the potential speedup).

Extensions to functional units and registers was a very big issue for earlier systems that were composed of discrete components. For modern systems implemented in VLSI, it is not as big of an issue but it can not be entirely ignored since a large number of long vectors of wide floating point numbers can still require substantial chip-area resources and results in more power consumption.

The memory system is perhaps one of the more critical issues for a vector machine. The first issue is simply a matter of bandwidth. To execute a large number of parallel operations in a short amount of time requires that the memory system can provide all of the arguments just as fast. Furthermore, caches may not provide much help with vector- or matrix-like data. This is because the access of items in a vector or matrix often occur in a regular pattern, often accessing all of the items in one row or one column of the matrix. This results in all of the memory references residing at memory locations that are a fixed distance or *stride* apart from each other. If the stride and the number of cache lines have common multiples, then what occurs is that all of the data get mapped into a small subset of the cache lines. For this reason, some vector processing systems do not even bother to store vector data in the cache hierarchy, but instead load the data directly from main memory.

2.1 Vector Registers and Functional Units

The set of vector registers typically consist of eight or more *register sets* (each set is a vector register). Each register typically contains 16-64 elements, and each element is typically a 64-bit floating point value. (Note the strong use of the word “typically”, all numbers may vary depending on the system and the specific design goals and implementation tradeoffs for that system.) Many vector processors are load/store architectures, which means special vector load and vector store instructions must be used to move data between memory and the vector registers. The memory functional units that move vector data are typically pipelined to provide a large data bandwidth. Since we are investing all of this hardware to perform vector operations, it is normally assumed that the length of the vectors we are operating on are reasonably large.

For both the memory units and the regular functional units (arithmetic), throughput is the main concern. The processor operates on long vectors of data, and so it is important that a large amount of data can be processed quickly, even if the amount of time to completely process a single item increases (latency). Contrast this to a scalar functional unit where the latency is the important characteristic.

Typically, vector functional units must be provided to perform vector floating-point addition/subtraction, multiplication, division or reciprocal, and logical operations such as comparisons. Note that division can be implemented by multiplication and reciprocal ($\frac{x}{y} = x \cdot \frac{1}{y}$). The reason to use a reciprocal function instead of a full division is that it is easier to pipeline the former. Note that using reciprocals for division must be done carefully to avoid running into situations where the precision of an answer is very low.

Vector instructions may also be overlapped. For instance, assume we want to execute the following two vector instructions:

$$\begin{aligned} \text{VADD } V1 &= V2 + V3 \\ \text{VMUL } V5 &= V4 + V1 \end{aligned}$$

In the first cycle, $V1_0$ is computed (the first/zeroth element of $V1$). On the second cycle, $V1_1$ is computed, and we can start computing $V5_0$ since $V1_0$ has already been computed. The timing is shown in Figure 1. Such an organization can potentially process a large number of arithmetic operations very quickly (i.e. high throughput). One burden that

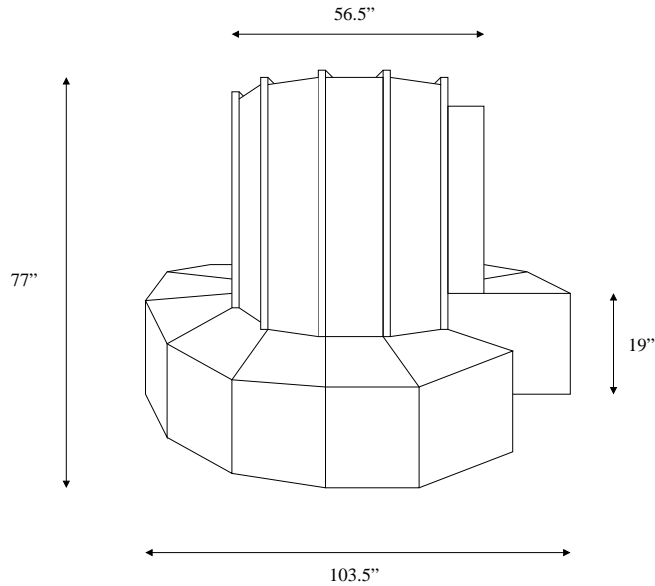


Figure 2: The physical dimensions of the CRAY-1.

- 1662 modules, 113 module types
- Up to 288 IC packages per modules
- For configuration with maximum allowable memory, power consumption is approximately 115 kw (over 380 modern computers which are about 300 watts each)
- Freon cooled with Freon/water heat exchange
- 10,500 lbs. (w/ maximum memory)
- composed of only three basic chip types:
 - 5- and 4-input NAND gates
 - Memory chips
 - Register chips
 - (That's All!)

The basic architecture includes 16 memory banks (interleaved/banked memory), 12 functional units, and more than 4KB worth of register storage (both scalar and vector). The other key implementation features of the CRAY-1 are that only four chip (IC) types are used, the speed of the main memory, the cooling system, and the computational core of course.

2.2.2 Four IC's

All logic in the CRAY is completely implemented with a mix of high- and low- speed 4- and 5-input NAND gates. Registers are implemented with 16×4 bit chips (with 6ns access time), and memory is implemented with 1024×1 bit memory chips (with 50ns access time). The overall clock cycle time is 12.5ns (80MHz), which is incredibly fast for the late 70's (consider that in 1990, an i386 at 16MHz was pretty fast)!

Each IC comes in a 16-pin flat package. The small size of the ICs were important since a single module (small PCB) may contain up to 288 ICs, and up to 72 such modules can be inserted into a 28" high chassis (there are 2 chassis

per “wedge”). Each module is 6 inches wide, and this distance requires about 1ns for a signal to get from one side of the board to the other. At these distances, the wires can start having transmission line behavior, and improper designs can induce standing waves in the ground plane. To solve this problem, all signal paths in the machine are of the same length. 10-20 percent of the IC packages are just there to pad out a signal line. The other part of the solution was using only simple gates and making sure both ends are properly terminated. More complex logic gates make it very difficult to ensure that signals are properly terminated on both ends.

2.2.3 Main Memory Speed

The CRAY-1’s main memory is organized into 16 banks, with 72 modules per bank. Each module contributes 1 bit out of a 64-bit word. The extra 8 modules provide bits for error detection and correction (8 bits allows SECDED - single error correction double error detection). Data words are stored in 1-bank increments, meaning that all of the bits for word n reside in the same bank, and then all of the bits for word $n + 1$ reside in the next bank. If data is not accessed with a 16-word stride, then the 16-way interleaved memory can be accessed without conflict.

2.2.4 Cooling System

The CRAY-1 generates a lot of heat (115 kw power dissipation!), and so cooling is more of a problem than for your overclocked Celeron sitting on your desk. The CRAY takes advantage of the actual metal conductors available in the machine itself (as far as the case and chassis go) to help conduct heat. Within each chassis, vertical aluminum/stainless steel cooling bars line each column wall. A Freon refrigerant flows through the stainless steel tube, and the tube is wrapped in a aluminum casing. Each individual module contains a copper heat transfer plate, which dissipates heat into the walls (and therefore into the cooling bars). The modules are also secured to the cooling bars with stainless steel pins that pinch the copper plate against the outer aluminum casing. Note that the term “cooling bar” should rightfully be replaced with “plumbing”.

A year and a half of trial and error was required before the first good cooling bar was built. The major problem was that an aluminum casting is porous, and so if there is a crack anywhere in the inner steel tube (these were happening at elbows), then Freon will leak through the aluminum casing. Loss of Freon is in itself not a major problem, but mixed in with the Freon is a little bit of oil, which can cause problems as it gets deposited on the modules. Long processes of temperature cycling were needed to minimize the porousness and the bubbles in the aluminum, and preheating of the steel was necessary to minimize cracking.

2.2.5 Functional Units

There are a total of 12 functional units, organized into the following four groups: address, scalar, vector, and floating point. All functional units are pipelined into single-cycle segments (i.e. throughput is one computation per cycle per unit). Parallelism is achieved by the high throughput of the vector units, i.e. processing lots of data at a high rate. Additional parallelism can also be achieved *across* the functional units. The CRAY-1 supports chaining of instructions, and so multiple functional units may be busy computing at the same time. The CRAY-1 also uses a reciprocal approximation unit instead of a division unit to maintain a fully pipelined functional unit.

All functional units are pipelined, and so one could imagine that all functional units could be used to process vector operations. Unfortunately, the data paths (operand and result busses) are limited, and so most functional units can only read from a limited number of sources. For example, the vector units can only process data from the vector register file, except for a single scalar argument to facilitate operations such as $5 \cdot \vec{x}$. The floating point functional units can read arguments from either the scalar or vector registers. Chaining of vector instructions can only occur between functional units that manipulate vector data (vector and FP units).

2.2.6 Registers

Figure 3 shows the registers in relation to the functional units, instruction buffers, and memory. The registers defined by the architecture include:

- Eight 24-bit address registers (A)

- Sixty Four 24-bit address-save registers (B)
- Eight 64-bit scalar registers (S)
- Sixty Four 64-bit scalar-save registers (T)
- Eight 64-**word** vector registers (V) - 4096-bits/vector

Unlike what was previously described, the vector registers in the CRAY-1 are not explicitly “floating-point”. They can hold any 64-bit value, regardless of whether the hardware interprets the values as 64-bit integers or floating point format numbers.

The address-save and scalar-save registers sort of act like a cache for the computer (there is no other cache hierarchy). The difference is that these locations are made explicit to the programmer. This would be analogous to a modern processor with instructions that can load a value from memory to a particular cache location, and other instructions to load from particular cache locations into a register (and the other direction for stores). It only takes a single clock cycle to move a value from a register to a save-register (same for the other direction), but the normal operational instructions (arithmetic, etc.) can not operate on the save-registers. This complicates the programming task somewhat since the allocation and management of data in the save registers is completely up to the programmer and/or compiler. Contrast this to a more traditional cache organization where everything is transparent to the programmer.

There are two additional registers to support vector operations. The register VL holds the length of the vectors being manipulated. For vectors that are shorter than 64 elements, cycles are not wasted processing the unused elements at the ends of the vectors. For vectors longer than 64 elements, the data are then split across multiple vector registers. The register VM is a 64-bit vector mask register. The result of a 64-bit vector comparison for example is placed in the VM register. The VM register interacts with the vector control logic for some operations to specify which elements of the vectors to process. For example, if the contents of the VM register are alternating 1’s and 0’s, then only every other vector element is processed.

There is no virtual memory in the system in the way we are used to, but a different technique is used to prevent different processes from accessing each others’ memory. Two registers BA and LA are used to define the memory boundaries for the current running process. BA is the base address, and LA is the limit address. Any time a memory reference is made, the address in question is checked to see that it is greater than or equal to BA, and less than LA (is the address in the allowed range?). If the address does not fall in this interval, than a memory access violation is flagged (interrupt), usually causing the process to terminate. The operating system must know exactly how much memory each process is going to use, and then it can set these registers accordingly.

3 Superscalar SIMD Extensions

Main stream processing (IE, Word, etc.) generally doesn’t exhibit the levels of parallelism that scientific computing often has. That is part of the reason why vector computers remained only in the supercomputing market, and was not integrated into commodity processors. Additionally, in the past it has been too expensive (either in terms of components or silicon area) to support vector operations in general purpose CPUs. More recently (over the past decade), relatively powerful processors have opened up new application areas to the mass market, such as audio and video manipulation (listening to MP3s, watching videos), high end graphics (photoshop, video games), and other areas that have higher levels of parallelism. For example, for many image processing tasks, the individual output pixels can be computed in parallel.

These applications have many small, tight inner loops in their codes, many of which can be transformed into more vector-like operations. Intel introduced an extension to their x86 ISA called MMX (originally “Matrix Manipulation eXtensions”, and then later rechristened as “Multi-Media eXtensions” by the marketing folks). The MMX extensions defines some new registers for holding vector data, as well as new vector instructions for manipulating the vector data. In the original MMX definition, eight new registers were added to the x86 architecture: MM0-MM7. Each register is 64-bits wide, and can hold either a single 64-bit value, two 32-bit values, four 16-bit values, or eight 8-bit values. So the scope of vectorization is somewhat less than what was used in the CRAY-1.

The new instructions operate of the MM* registers. Moving data into and out of the MM* registers requires MOVQ and MOVQ instructions (for move double- and move quad-word). The values may be moved from memory to MM*

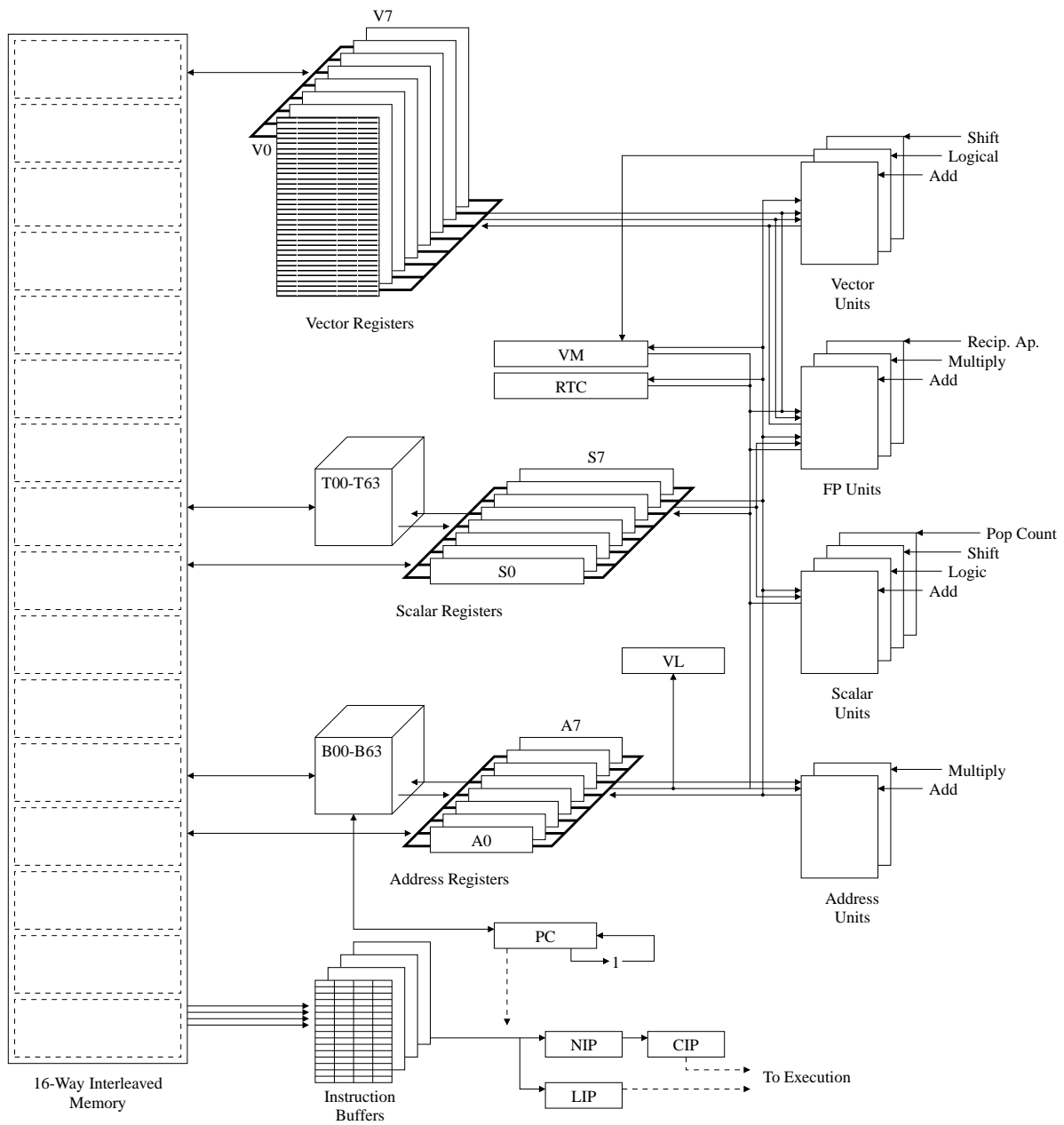


Figure 3: Block diagram of the CRAY-1 computer.

registers (and back), or between the regular integer registers and the MM* registers. The operational instructions are basically vector generalizations of the normal *integer* arithmetic/logic instructions. The arithmetic instructions can only operate on bytes, words and double words (no support for 64-bit arithmetic operations). An instruction such as PADDW is a parallel (vector) add, treating the arguments as vectors of (16-bit) words. This allows four independent 16-bit additions to be performed by a single instruction.

The MMX extensions also define a new arithmetic mode: saturating arithmetic. Normally, when (say) an 8-bit value is added to another 8-bit value (assume unsigned operations), it is possible that the result is greater than 255, which causes the result to overflow. Possible outcomes are either an overflow exception, or simply to ignore the carry (truncate the upper bits) which is called “wrap-around mode”. In some applications, the wrap around effect is not desirable. Consider for example an image processing task where the brightness of the image is increased. It is possible that the intensity gets increased to a level that would cause overflow. If wrap around mode is used, then the resulting value is something very small, which suddenly gets interpreted as a dark color instead of a light/bright color. In saturating arithmetic, instead of overflowing, the result is assigned the maximum allowable value. For example (for unsigned 16-bit arithmetic):

$$\begin{aligned}40,561 + 38,119 &= 13144 \quad (\text{in wrap around mode}) \\40,561 + 38,119 &= 65535 \quad (\text{in saturating mode})\end{aligned}$$

There are some bit-wise logical operations that are the only instructions (besides moves) that can operate on 64-bit data. PAND, PANDN (AND NOT), POR and PXOR can perform 64-bit bit-wise logical computations (this is easy to implement without any effect on the clock speed since all 64-bits are computed in parallel).

There are also some additional instruction for “packing” (and unpacking) data. This can take for example, a vector of four 16-bit words, and convert it into four 8-bit words (using wrap-around or saturation) and place the resulting “half-vector” in the lower or upper half of a vector register. Reverse operations can be performed with zero- or sign-extension to expand a low-precision format into a higher precision data format.

Since MMX, other vector/SIMD extensions have been introduced to the x86 ISA. SSE and SSE2 are the “Streaming SIMD Extensions”, which are basically floating point SIMD instructions. These include additional instructions and registers for 128-bit integer and double-precision floating point vector operations (2 doubles fit into 128-bits).