# VARIOUS DEFINITIONS ON DIGITAL SIGNAL PROCESSORS

## Digital signal processor

A digital signal processor (DSP) is a specialized microprocessor designed specifically for digital signal processing, generally in real-time.

## Real-time processing

- The ability to act as a direct memory access device for the host environment.
- Separate program and data memories (Harvard architecture).
- Only parallel processing, no multitasking

## Digital signal processing

Digital signal processing can be done on general-purpose microprocessors. Possible optimizations:

### Data operators

- Saturation arithmetic, in which operations that produce overflows will accumulate at the maximum (or minimum) values that the register can hold rather than wrapping around (maximum+1 doesn't equal minimum as in many general-purpose CPUs, instead it stays at maximum). Sometimes various sticky bits operation modes are available.
- Multiply-accumulate (MAC) operations, which is good for any kind of matrix operation, such as convolution for filtering, Dot product, or even polynomial evaluation (see Horner scheme, also Fused multiply-add). The general form of FIR filtering is given by the following equation:

$$y(n) = \sum_{i=0}^{L-1} b_i \cdot x(n-i) = \begin{bmatrix} b_o & b_1 & . & b_{L-1} \end{bmatrix} \cdot \begin{bmatrix} x(n) \\ x(n-1) \\ . \\ x(n-L+1) \end{bmatrix}$$

  Single cycle MAC is an assumption in many DSPs, thus a lot of the following properties are derived (esp. Harvard architecture and pipelining).
- Specialized instructions for modulo addressing in ring buffers and bit-reversed addressing mode for FFT cross-referencing.

### Program flow

- Deep pipelining. That makes wrong predicted branches costly.
- Branch prediction. Either with a dynamic table or hard coded as zero-overhead looping. To alleviate the branch impact for execution hi-frequent inner-loops, some processors provide this feature. There are two types of operation: single instruction repeating and multi-instruction loops.

## History

In 1978, Intel released the 2920 as an "analog signal processor". It had an on-chip ADC/DAC with an internal signal processor, but it didn't have a hardware multiplier and was not successful in the market. In 1979, AMI released the S2811. It was designed as a microprocessor peripheral, and it had to be initialized by the host. The S2811 was likewise not successful in the market.

In 1979, Bell Labs introduced the first single chip DSP, the Mac 4 Microprocessor. Then, in 1980 the first stand-alone, complete DSPs -- the NEC μPD7720 and AT&T DSP1 -- were presented at the IEEE International Solid-State Circuits Conference '80. Both processors were inspired by the research in PSTN telecommunications.

The Altamira DX-1 was another early DSP, utilizing a quad integer pipelines with delayed branches and branch prediction.

The first DSP produced by Texas Instruments (TI), the TMS32010 presented in 1983, proved to be an even bigger success, and TI is now the market leader in general purpose DSPs. Another successful design was the Motorola 56000.

General purpose CPU's have ideas and influences from digital signal processors with Extensions such as the MMX extensions in the Intel IA-32 architecture instruction set (ISA).

Most DSPs use fixed-point arithmetic, because in real world signal processing, the additional range provided by floating point is not needed, and there is a large speed benefit; however, floating point DSPs are common for scientific and other applications where additional range or precision may be required.

## See also

Generally, DSPs are dedicated integrated circuits, however DSP functionality can also be realised using Field Programmable Gate Array chips.

## Digital signal processing

Digital signal processing (DSP) is the study of signals in a digital representation and the processing methods of these signals. DSP and analog signal processing are subfields of signal processing. DSP has three major subfields: audio signal processing, digital image processing and speech processing.

Since the goal of DSP is usually to measure or filter continuous real-world analog signals, the first step is usually to convert the signal from an analog to a digital form, by using an analog to digital converter. Often, the required output signal is another analog output signal, which requires a digital to analog converter.

The algorithms required for DSP are sometimes performed using specialized computers, which make use of specialized microprocessors called digital signal processors (also abbreviated DSP). These process signals in real time and are generally purpose-designed ASICs.

### DSP domains

In DSP, engineers usually study digital signals in one of the following domains: time domain (one-dimensional signals), spatial domain (multidimensional signals), frequency domain, autocorrelation domain, and wavelet domains. They choose the domain in which to process a signal by making an educated guess (or by trying different possibilities) as to which domain best represents the essential characteristics of the signal. A sequence of samples from a measuring device produces a time or spatial domain representation, whereas a discrete Fourier transform produces the frequency domain information, that is the frequency spectrum. Autocorrelation is defined as the cross-correlation of the signal with itself over varying intervals of time or space.

### Signal sampling

With the increasing use of computers the usage and need of digital signal processing has increased. In order to use an analog signal on a computer it must be digitized with an analog to digital converter (ADC). Sampling is usually carried out in two stages, discretization and quantization. In the discretization stage, the space of signals is partioned into equivalence classes and discretization is carried out by replacing the signal with representative signal of the corresponding equivalence class. In the quantization stage the representative signal values are approximated by values from a finite set.

In order to properly sample an analog signal the Nyquist-Shannon sampling theorem must be satisfied. In short, the sampling frequency must be greater than twice the bandwidth of the signal (provided it is filtered appropriately). A digital to analog converter (DAC) is used to convert the digital signal back to analog. The use of a digital computer is a key ingredient into digital control systems.

### Time and space domains

The most common processing approach in the time or space domain is enhancement of the input signal through a method called filtering. Filtering generally consists of some transformation of a

number of surrounding samples around the current sample of the input or output signal. There are various ways to characterize filters; for example:

- A "linear" filter is a linear transformation of input samples; other filters are "non-linear." Linear filters satisfy the superposition condition, i.e. if an input is a weighted linear combination of different signals, the output is an equally weighted linear combination of the corresponding output signals.
- A "causal" filter uses only previous samples of the input or output signals; while a "non-causal" filter uses future input samples. A non-causal filter can be changed into a causal filter by adding a delay to it.
- A "time-invariant" filter has constant properties over time; other filters such as adaptive filters change in time.
- Some filters are "stable", others are "unstable". A stable filter produces an output that converges to a constant value with time, or remains bounded within a finite interval. An unstable filter produces output which diverges.
- A "finite impulse response" (FIR) filter uses only the input signal, while an "infinite impulse response" filter (IIR) uses both the input signal and previous samples of the output signal. FIR filters are always stable, while IIR filters may be unstable.

Most filters can be described in Z-domain (a superset of the frequency domain) by their transfer functions. A filter may also be described as a difference equation, a collection of zeroes and poles or, if it is an FIR filter, an impulse response or step response. The output of an FIR filter to any given input may be calculated by convolving the input signal with the impulse response. Filters can also be represented by block diagrams which can then be used to derive a sample processing algorithm to implement the filter using hardware instructions.

## Frequency domain

Signals are converted from time or space domain to the frequency domain usually through the Fourier transform. The Fourier transform converts the signal information to a magnitude and phase component of each frequency. Often the Fourier transform is converted to the power spectrum, which is the magnitude of each frequency component squared.

The most common purpose for analysis of signals in the frequency domain is analysis of signal properties. The engineer can study the spectrum to get information of which frequencies are present in the input signal and which are missing.

There are some commonly used frequency domain transformations. For example, the cepstrum converts a signal to the frequency domain through Fourier transform, takes the logarithm, then applies another Fourier transform. This emphasizes the frequency components with smaller magnitude while retaining the order of magnitudes of frequency components.

## Applications

The main applications of DSP are audio signal processing, audio compression, digital image processing, video compression, speech processing, speech recognition and digital communications. Specific examples are speech compression and transmission in digital mobile phones, equalisation of sound in Hifi equipment, weather forecasting, economic forecasting, seismic data processing, analysis and control of industrial processes, computer-generated animations in movies, medical imaging such as CAT scans and MRI, image manipulation, and digital effects for use with electric guitar amplifiers. A further application is very low frequency (VLF) reception with a PC soundcard [1].

## Techniques

- Bilinear transform
- Discrete Fourier transform
- Discrete-time Fourier transform
- Filter design
- LTI system theory
- Minimum phase
- Transfer function
- Z-transform
- Goertzel algorithm

## Real-time

An operation within a larger dynamic system is called a real-time operation if the combined reaction- and operation-time of a task is shorter than the maximum delay that is allowed, in view of circumstances outside the operation. The task must also occur before the system to be controlled becomes unstable. A real-time operation is not necessarily fast, as slow systems can allow slow real-time operations. This applies for all types of dynamically changing systems. The polar opposite of a real-time operation is a batch job with interactive timesharing falling somewhere in-between the two extremes.

Alternately, a system is said to be hard real-time if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. An operation performed after the deadline is, by definition, incorrect, and usually has no value. In a soft real-time system the value of an operation declines steadily after the deadline expires.

A typical example could be a computer-controlled braking system in a car. If the driver can stop a car before it hits a wall, the operation was in real-time; if the car hits the wall it was not. Many machines require real-time controllers to avoid "instability", which could lead to the accidental damage or destruction of the system, people, or objects.

Some real-time systems do not have such a constraint on delay as long as input data can be processed rapidly enough so that no backlog occurs. In a real-time Digital signal processing (DSP) system, the analyzed (input) and/or generated (output) samples (whether they are grouped together in large segments or processed individually) can be processed (or generated) continuously in the time it takes to input and/or output the same set of samples independent of the processing delay. Consider an audio DSP example: if a process requires 2.01 seconds to analyze or process 2.00 seconds of sound, it is not real-time. If it takes 1.99 seconds, it is (or can be made into) a real-time DSP process.

A common life example is that of standing in a queue (line of customers) waiting for the checkout in a grocery store. The queue is "real-time" if, on average, customers are being processed and their transactions completed as quickly as they arrive. Under these circumstances, a queue of customers may occasionally form, but the queue cannot grow without bound.

If, on the other hand, customers are not processed as quickly as they arrive, then the queue of waiting customers will grow without limits. This system is not real-time and would definitely be deemed a failure.

A more subtle failure might occur even if the system can, on average, process customers as quickly as they arrive. A burst of arrivals or some slow processing of the existing customers will cause the queue to fill. If the queue becomes sufficiently long, customers will be discouraged and leave without making a purchase. This system would also have to be deemed a failure, even though it meets the strict definition of real time as described above. In this case, the system has a limited "buffer capacity" (tolerable queue length) and it must process customers fast enough to keep the queue shorter than that tolerable queue length even under a worst-case load of arriving customers and the slowest processing of existing customers.

Real stores (and real real-time systems) often need better than real-time performance, and worst-case scenarios must be carefully evaluated. In the case of the hypothetical grocery store, they may define a given "service level", expect real-time processing up to that service level, and accept a certain loss of customers beyond that service level. A missile designer, on the other hand, may not be willing to define a service level beyond which failures can be expected; for the missile designer, no failure is acceptable.

In the economy, real-time systems are information technologies, which provide real-time access to information or data. The ability of a company to process its data in real time increases the competitiveness of the company. Real-time processing systems are new technologies and will improve during the next decades. Gartner forecasts a fast increase and use of these real-time systems.

## Direct memory access

Direct memory access (DMA) allows certain hardware subsystems within a computer to access system memory for reading and/or writing independently of the CPU. Many hardware systems use DMA including disk drive controllers, graphics cards, network cards, and sound cards.

### Principle

DMA is an essential feature of all modern computers, as it allows devices to transfer data without subjecting the CPU to a heavy load. Otherwise, the CPU would have to copy each piece of data from the source to the destination. This is typically slower than copying normal blocks of memory since access to I/O devices over a peripheral bus is generally slower than normal system RAM. During this time the CPU would be unavailable for other tasks.

A DMA transfer essentially copies a block of memory from one device to another. While the CPU initiates the transfer, it does not execute the transfer itself. For so-called "third party" DMA, as is normally used with the ISA bus, the transfer is performed by a DMA controller which is typically part of the motherboard chipset. More advanced bus designs such as PCI typically use bus-mastering DMA, where the device takes control of the bus and performs the transfer itself.

A typical usage of DMA is copying a block of memory from system RAM to or from a buffer on the device. Such an operation does not stall the processor, which as a result can be scheduled to perform other tasks. DMA transfers are essential to high performance embedded systems. It is also essential in providing so-called zero-copy implementations of peripheral device drivers as well as functionalities such as network packet routing, audio playback and streaming video.

### Examples

#### ISA

For example, a PC's ISA DMA controller has 16 DMA channels of which 7 are available for use by the PC's CPU. Each DMA channel has associated with it a 16-bit address register and a 16-bit count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. It then instructs the DMA hardware to begin the transfer. When the transfer is complete the device then interrupts the CPU.

"Scatter-gather" DMA allows the transfer of data to multiple memory areas in a single DMA transaction. It is equivalent to the chaining together of multiple simple DMA requests. Again, the motivation is to off-load multiple I/O interrupt and data copy tasks from the CPU.

DRQ stands for DMA request; DACK for DMA acknowledge. These symbols are generally seen on hardware schematics of computer systems with DMA functionality. They represent electronic signaling lines between the CPU and DMA controller.

## Harvard Architecture

The term Harvard architecture originally *referred to computer architectures that used physically separate storage and signal pathways for their instructions and data* (in contrast to the von Neumann architecture). The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24-bits wide) and data in relay latches (23-digits wide). These early machines had very limited data storage, entirely contained within the data processing unit, and provided no access to the instruction storage as data (making loading, modifying, etc. of programs entirely an offline process).

In a computer with a von Neumann architecture, the CPU can be either reading an instruction or reading/writing data from/to the memory. Both cannot occur at the same time since the instructions and data use the same signal pathways and memory. In a computer with Harvard architecture, the CPU can read both an instruction and data from memory at the same time. A computer with Harvard architecture can be faster because it is able to fetch the next instruction at the same time it completes the current instruction. Speed is gained at the expense of more complex electrical circuitry.

In recent years the speed of the CPU has grown many times in comparison to the access speed of the main memory. Care needs to be taken to reduce the number of times main memory is accessed in order to maintain performance. If, for instance, every instruction run in the CPU requires an access to memory, the computer gains nothing for increased CPU speed - a problem referred to as being memory bound.

Memory can be made much faster, but only at high cost. The solution then is to provide a small amount of very fast memory known as a cache. As long as the memory that the CPU needs is in the cache, the performance hit is much smaller than it is when the cache has to turn around and get the data from the main memory. Tuning the cache is an important aspect of computer design.

Modern high performance CPU chip designs incorporate aspects of both Harvard and von Neumann architecture. On chip cache memory is divided into an instruction cache and a data cache. Harvard architecture is used as the CPU accesses the cache. In the case of a cache miss, however, the data is retrieved from the main memory, which is not divided into separate instruction and data sections. Thus a von Neumann architecture is used for off chip memory access.

Harvard architectures are also frequently used in specialized DSPs, or digital signal processors, commonly used in audio or video processing products. For example, Blackfin processors by Analog Devices Inc make use of a Harvard architecture.

Additionally, most general purpose small microcontrollers used in several electronics applications, such as the PIC microcontrollers made by Microchip Technology Inc, are based on the Harvard architecture. These processors are characterized by having small amounts of program and data memory, and take advantage of the Harvard architecture and reduced instruction set to ensure that most instructions can be executed within only one machine cycle. The separate storage means the program and data memories can be in different bit depths. For example, the PIC microcontroller has an 8-bit data word but a 12-bit, 14-bit, or 16-bit program word (depending on specific PIC). This allows

a single instruction to contain a full-size data constant. Other RISC architectures, for example the ARM, typically have to use at least two instructions to load a full-size constant.

Another popular controller family are the Atmel AVR controllers.

## Saturation arithmetic

Saturation arithmetic is a version of arithmetic in which all operations such as addition and multiplication are limited to a fixed range between a minimum and maximum value. If the result of an operation is above the maximum it is set to the maximum, while if it is below the minimum it is set to the minimum. The name comes from how the value becomes "saturated" once it reaches the maximum value; further additions will not increase it.

For example, if the valid range of values is from -100 to 100, the following operations produce the following values:

- 60 + 43 = 100
- (60 + 43) - 150 = -50
- 43 - 150 = -100
- 60 + (43 - 150) = -40
- 10 × 11 = 100
- 99 × 99 = 100
- 30 × (5 - 1) = 100
- 30×5 - 30×1 = 70

As can be seen from these examples, familiar properties like associativity and distributivity fail in saturation arithmetic. This makes it unpleasant to deal with in abstract mathematics, but it has an important role to play in digital hardware and algorithms.

Typically, early computer microprocessors did not implement integer arithmetic operations using saturation arithmetic; instead, they used the easier-to-implement modular arithmetic, in which values exceeding the maximum value "wrap around" to the minimum value, like the hours on a clock passing from 12 to 1. In hardware, modular arithmetic with a minimum of zero and a maximum of 2n can be implemented by simply discarding all but the lowest n bits.

However, although more difficult to implement, saturation arithmetic has numerous practical advantages. The result is as numerically close to the true answer as possible; it's considerably less surprising to get an answer of 127 instead of 130 than to get an answer of -126 instead of 130. It also enables overflow of additions and multiplications to be detected consistently without an overflow bit or excessive computation by simple comparison with the maximum or minimum value (provided the datum is not permitted to take on these values).

Additionally, saturation arithmetic enables efficient algorithms for many problems, particularly in signal processing. For example, adjusting the volume level of a sound signal can result in overflow, and saturation causes significantly less distortion to the sound than wrap-around. In the words of researchers G. A. Constantinides et al:

*"When adding two numbers using two's complement representation, overflow results in a 'wrap-around' phenomenon. The result can be a catastrophic loss in signal-to-noise ratio in a DSP system. Signals in DSP designs are therefore usually either scaled appropriately to avoid overflow for all but the most extreme input vectors, or produced using saturation arithmetic components."*

Saturation arithmetic operations are available on many modern platforms, and in particular was one of the extensions made by the Intel MMX platform, specifically for such signal processing applications.

Saturation arithmetic for integers has also implemented in software for a number of programming languages including C, C++, Eiffel, and most notably Ada, which has built-in support for saturation arithmetic. This helps programmers anticipate and understand the effects of overflow better. On the other hand, saturation is challenging to implement efficiently in software on a machine with only modular arithmetic operations, since simple implementations require branches that create huge pipeline delays.

Although saturation arithmetic is less popular for integer arithmetic in hardware, the IEEE floating-point standard, the most popular abstraction for dealing with approximate real numbers, uses a form of saturation in which overflow is converted into "infinity" or "negative infinity", and any other operation on this result continues to produce the same value. This has the advantage over simple saturation that later operations which decrease the value will not end up producing a "reasonable" result, such as in the computation $\sqrt{x^2 - y^2}$ .

## Multiply-accumulate

The multiply-accumulate operation computes a product and adds it to an accumulator.

$$a \leftarrow a + b \cdot c$$

When done with integers this operation is typically exact (computed modulo some power of 2).

When done with floating point numbers it might be performed with two roundings (typical in many DSPs) or with a single rounding, called a fused multiply-add (FMA).

## Instruction pipeline

An instruction pipeline is a technology used on microprocessors to enhance their performance. Pipelining greatly improves throughput, the average number of instructions performed per second.

Instructions consist of a number of steps. Practically every CPU ever manufactured is driven by a central clock. Each step requires at least one clock cycle. Each step of an instruction is performed by a different piece of hardware on the CPU. Early, non-pipelined processors did only one step at a time. For example, they might perform these steps sequentially in order:

1. Read the next instruction
2. Read the operands, if any
3. Execute the instruction
4. Write the results back out

This cycle is called the *instruction cycle*. This approach, while simple, is wasteful. While the processor is adding numbers, for instance, the hardware dedicated to loading data from computer memory is idle, waiting for the addition to complete.

Pipelining improves performance by reducing the idle time of each piece of hardware. Pipelined CPUs subdivide various functional units within a processor into different stages, or relatively independent components, which can each be working on a different task. Stages are ordered in sequence with the output of each stage feeding the input of the stage after it. Because each stage performs only a small part of the overall computation, each function takes only a short time, and overall clock speed can be increased tremendously.

Unfortunately, not all instructions are independent. In a simple pipeline, completing an instruction may require 5 stages. To operate at full performance, this pipeline will need to run 4 subsequent independent instructions while the first is completing. If 4 instructions that do not depend on the output of the first instruction are not available, the pipeline control logic must insert a stall or wasted clock cycle into the pipeline until the dependency is resolved. Fortunately, techniques such as forwarding can significantly reduce the cases where stalling is required. While pipelining can in theory increase performance over an unpipelined core by a factor of the number of stages (assuming the clock frequency also scales with the number of stages), in reality, most code does not allow for ideal execution.

Early implementations typically had a few "stages", perhaps four, with a few designs using superpipelining with many more stages. Modern desktop processors almost universally use between 15 and 30 stages, and the term "super" has fallen from use.

## Examples

### Example 1

For instance, a typical instruction to add two numbers might be ADD A, B, C, which adds the values found in memory locations A and B, and then puts the result in memory location C. In a pipelined processor the pipeline controller would break this into a series of instructions similar to:

---

*LOAD A, R1*

*LOAD B, R2*

*ADD R1, R2, R3*

*STORE R3, C*

*LOAD next instruction*

---

The R locations are registers, temporary memory inside the CPU that is quick to access. The end result is the same, the numbers are added and the result placed in C, and the time taken to drive the addition to completion is no different from the non-pipelined case.

The key to understanding the advantage of pipelining is to consider what happens when this ADD instruction is "half-way done", at the ADD instruction for instance. At this point the circuitry responsible for loading data from memory is no longer being used, and would normally sit idle. In this case the pipeline controller fetches the next instruction from memory, and starts loading the data it needs into registers. That way when the ADD instruction is complete, the data needed for the next ADD is already loaded and ready to go. The overall effective speed of the machine can be greatly increased because no parts of the CPU sit idle.

Each of the simple steps are usually called pipeline stages, in the example above the pipeline is three stages long, a loader, adder and storer.

Every microprocessor manufactured today uses at least 2 stages of pipeline. (The Atmel AVR and the PIC microcontroller each have a 2 stage pipeline).

### Example 2

To better visualize the concept, we can look at a theoretical 3-stages pipeline:

---

| Stage | Description |
|---|---|
| Load | Read instruction from memory |
| Execute | Execute instruction |
| Store | Store result in memory and/or registers |

and a pseudo-code assembly listing to be executed:

---

```
LOAD  #40, A      ; load 40 in A
MOVE  A, B        ; copy A in B
ADD   #20, B      ; add 20 to B
STORE B, 0x300    ; store B into memory cell 0x300
```

---

This is how it would be executed:

Clock 1

| Load | Execute | Store |
|---|---|---|
| LOAD | | |

The LOAD instruction is fetched from memory.

Clock 2

| Load | Execute | Store |
|---|---|---|
| MOVE | LOAD | |

The LOAD instruction is executed, while the MOVE instruction is fetched from memory.

Clock 3

| Load | Execute | Store |
|---|---|---|
| ADD | MOVE | LOAD |

The LOAD instruction is in the Store stage, where its result (the number 40) will be stored in the register A. In the meantime, the MOVE instruction is being executed. Since it must move the contents of A into B, it must wait for the ending of the LOAD instruction.

Clock 4

| Load | Execute | Store |
|---|---|---|
| STORE | ADD | MOVE |

The STORE instruction is loaded, while the MOVE instruction is finishing off and the ADD is calculating.

And so on. Note that, sometimes, an instruction will depend on the result of another one (like our MOVE example). When more than one instruction references a particular location for an operand, either reading it (as an input) or writing it (as an output), executing those instructions in an order different from the original program order can lead to problems, also known as hazards. There are several established techniques for either preventing hazards from occurring, or working around them if they do.

## Complications

Many designs include pipelines as long as 7, 10 and even 31 stages (like in the Intel Pentium 4). The Xelerator X10q has a pipeline more than a thousand stages long [1]. The downside of a long pipeline is when a program branches, the entire pipeline must be flushed, a problem that branch predicting helps to alleviate. Branch predicting itself can end up exacerbating the problem if branches are predicted poorly. In certain applications, such as supercomputing, programs are specially written to rarely branch and so very long pipelines are ideal to speed up the computations, as long pipelines are designed to reduce clocks per instruction (CPI). However in many applications, such as office software, branching happens constantly so the extra CPI and a long pipeline offers doesn't necessarily speed up processing.

The higher throughput of pipelines falls short when the executed code contains many branches: the processor cannot know where to read the next instruction, and must wait for the branch instruction to finish, leaving the pipeline behind it empty. After the branch is resolved, the next instruction has to travel all the way through the pipeline before its result becomes available and the processor appears to "work" again.

Because of the instruction pipeline, code that the processor loads will not immediately execute. Due to this, updates in the code very near the current location of execution may not take effect because they are already loaded into the Prefetch Input Queue. Instruction caches make this phenomenon even worse. This is only relevant to self-modifying programs such as operating systems.

## MMX

MMX is a SIMD instruction set designed by Intel, introduced in 1997 in their Pentium MMX microprocessors. It developed out of a similar unit first introduced on the Intel i860. It has been supported on most subsequent IA-32 processors by Intel and other vendors.

MMX is rumoured to stand for MultiMedia, Multiple Math or Matrix Math eXtension, but officially it is a meaningless initialism trademarked by Intel. (Note that AMD, during one of its numerous court battles with Intel, produced marketing material from Intel indicating that MMX stood for "Matrix Math Extensions". The idea that it stands for nothing is an Intel corporate position meant to suggest that it is of trademarked status and cannot be used by AMD or other x86 clone manufacturers in their own marketing material.)

To simplify the design and to avoid modifying the operating system to preserve additional state through context switches, MMX re-uses the existing eight IA-32 FPU registers. This made it difficult to work with floating point and SIMD data at the same time. To maximize performance, programmers must use the processor exclusively in one mode or the other, deferring the relatively slow switch between them as long as possible.

Another problem for MMX is that it only provides integer operations. Each of the eight 64-bit MMX vector registers, aliased on the eight existing floating point registers, could represent two 32-bit integers, four 16-bit short integers, or eight 8-bit chars. When originally developed in the i860, the use of vectored-integer math made sense (both 2D and 3D setup required it), but as the systems moved to using graphics cards that did this, MMX fell out of favor and vectored-floating point became much more important. On the other hand, its new arithmetic operations did include saturation arithmetic operations, which could significantly speed up some digital signal processing applications.

Intel later addressed these shortcomings with SSE, a greatly expanded set of SIMD instructions with 32-bit floating point support and an additional set of 128-bit vector registers that made it easy to perform SIMD and FPU operations at the same time. SSE was in turn expanded with SSE2 and then SSE3. Support for any of these later instruction sets implies support for MMX.

Intel's competitor AMD enhanced Intel's MMX with the 3DNow! instruction set.

## IEEE floating-point standard

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations. The standard defines formats for representing floating-point numbers (including ±zero and denormals) and special values (infinities and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions (including when the exceptions occur, and what happens when they do occur).

IEEE 754 specifies four formats for representing floating-point values: single-precision (32-bit), double-precision (64-bit), single-extended precision (≥ 43-bit, not commonly used) and double-extended precision (≥ 79-bit, usually implemented with 80 bits). Only 32-bit values are required by the standard; the others are optional. Many languages specify that IEEE formats and arithmetic be implemented, although sometimes it is optional. For example, the C programming language, which pre-dated IEEE 754, now allows but does not require IEEE arithmetic (the C float typically is used for IEEE single-precision and double uses IEEE double-precision).

The full title of the standard is IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985), and it is also known as IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems (originally the reference number was IEC 559:1989).[1]

### Anatomy of a floating-point number

Following is a description of the standard's format for floating-point numbers.

#### Bit conventions used in this article

Bits within a word of width W are indexed by integers in the range 0 to W−1 inclusive. The bit with index 0 is drawn on the right. The lowest indexed bit is usually the least significant.

#### Single-precision 32 bit

A single-precision binary floating-point number is stored in a 32 bit word:

| *Width in bits* | | |
|---|---|---|
| 1 | 8 | 23 |
| Sign | Exponent | Fraction |
| *bit index (0 on right)* | | |
| 31 | 30 29 ..... 23 | 22 21 ..... 1 0 |

The exponent is biased in the engineering sense of the word – the value stored is offset (by 127 in this case) from the actual value. Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder. To solve this the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison. So, for a single-precision number, an exponent in the range −126 to +127 is biased by adding 127 to get a

value in the range 1 to 254 (0 and 255 have special meanings described below). When interpreting the floating-point number the bias is subtracted to retrieve the actual exponent.

The set of possible data values can be divided into the following classes:

- zeroes
- normalised numbers
- denormalised numbers
- infinities
- NaN (Not a Number)

(NaNs are used to represent undefined or invalid results, such as the square root of a negative number.)

The classes are primarily distinguished by the value of the Exp field, modified by the fraction. Consider the Exp and Fraction fields as unsigned binary integers (Exp will be in the range 0–255):

| *Class* | *Exponent Value* | *Fraction Value* |
|---|---|---|
| Zeroes | *0* | *0* |
| *Denormalised numbers* | *0* | *non zero* |
| *Normalised numbers* | *1-254* | *any* |
| *Infinities* | *255* | *0* |
| *NaN (Not a Number)* | *255* | *non zero* |

For normalised numbers, the most common, Exp is the biased exponent and Fraction is the fractional part of the significand. The number has value v:

$$v = s \times 2^e \times m$$

Where

s = +1 (positive numbers) when S (Sign) is 0

s = −1 (negative numbers) when S (Sign) is 1

e = Exp − 127

(in other words the exponent is stored with 127 added to it, also called "biased with 127")

m = 1.Fraction in binary

(that is, the significand is the binary number 1 followed by the radix point followed by the binary bits of Fraction). Therefore, $1 \leq m < 2$.

Note:

1. Denormalised numbers are the same except that e = −126 and m is 0.Fraction. (e is NOT -127 : The significand has to be shifted to the right by one more bit, in order to include the leading bit,

which is not always 1 in this case. This is balanced by incrementing the exponent to -126 for the calculation.)

2. −126 is the smallest exponent for a normalised number
3. There are two Zeroes, +0 (S is 0) and −0 (S is 1)
4. There are two Infinities +∞ (S is 0) and −∞ (S is 1)
5. NaNs may have a sign and a significand, but these have no meaning other than for diagnostics; the first bit of the significand is often used to distinguish signaling NaNs from quiet NaNs
6. NaNs and Infinities have all 1s in the Exp field.

### An example

Let us encode the decimal number −118.625 using the IEEE 754 system.

We need to get the sign, the exponent and the fraction.

- Because it is a negative number, the sign is "1". Let's find the others.
- First, we write the number (without the sign) using binary notation. Look at binary numeral system to see how to do it. The result is 1110110.101.
- Now, let's move the radix point left, leaving only a 1 at its left: 1110110.101 = 1.110110101 × 26. This is a normalised floating point number.
- The fraction is the part at the right of the radix point, filled with 0 on the right until we get all 23 bits. That is 11011010100000000000000.
- The exponent is 6, but we need to convert it to binary and bias it (so the most negative exponent is 0, and all exponents are non-negative binary numbers). For the 32-bit IEEE 754 format, the bias is 127 and so 6 + 127 = 133. In binary, this is written as 10000101.

Putting them all together:

| *Width in bits* | | |
|---|---|---|
| 1 | 8 | 23 |
| Sign | Exponent | Fraction |
| 1 | 1 0 0 0 0 1 0 1 | 1 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 |
| *bit index (0 on right)* (the exponent bias is +127 = $2^{8-1}$) | | |
| 31 | 30 29 ..... 23 | 22 21 ..... 1 0 |

### Double-precision 64 bit

Double-precision is essentially the same except that the fields are wider:

| *Width in bits* | | |
|---|---|---|
| 1 | 11 | 52 |
| Sign | Exponent | Fraction |
| *bit index (0 on right)* (exponent bias is +1023 $=2^{11-1}$) | | |
| 63 | 62 61 ..... 52 | 51.....1 0 |

- NaNs and Infinities are represented with Exp being all 1s (2047).
- For Normalised numbers the exponent bias is +1023 (so e is Exp − 1023).
- For Denormalised numbers the exponent is −1022 (the minimum exponent for a normalised number—it is not −1023 because normalised numbers have a leading 1 digit before the binary point and denormalised numbers do not). As before, both infinity and zero are signed.

### Comparing floating-point numbers

Comparing floating-point numbers is usually best done using floating-point instructions. However, this representation makes comparisons of some subsets of numbers possible on a byte-by-byte basis, if they share the same byte order and the same sign, and NaNs are excluded.

For example, for two positive numbers a and b, then a < b is true whenever the unsigned binary integers with the same bit patterns and same byte order as a and b are also ordered a < b. In other words, two positive floating-point numbers (known not to be NaNs) can be compared with an unsigned binary integer comparison using the same bits, providing the floating-point numbers use the same byte order (this ordering, therefore, cannot be used in portable code through a union in the C programming language). This is an example of lexicographic ordering.

### Rounding floating-point numbers

The IEEE standard has four different rounding modes.

- Unbiased which rounds to the nearest value, if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit. This mode is required to be default.
- Towards zero
- Towards positive infinity
- Towards negative infinity