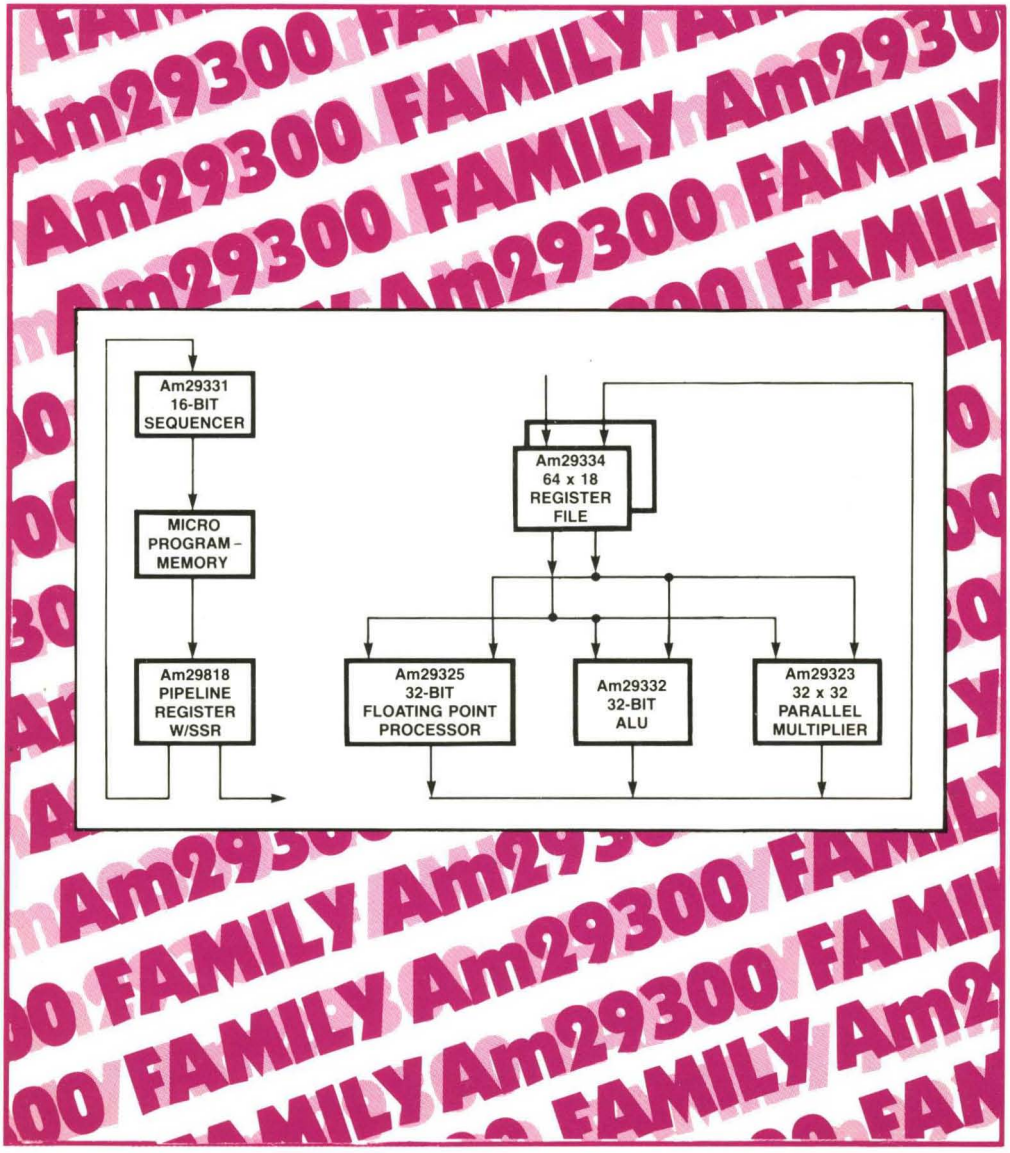
Am29300 Family Handbook

High Performance
32-Bit
Building
Blocks

April 1985



ADVANCED MICRO DEVICES

Advanced Micro Devices

Am29300 Family Handbook

The International Standard of Quality guarantees a 0.05% AQL on all electrical parameters, AC and DC, over the entire operating range.

INT·STD·500

901 Thompson Place, P.O. Box 3453, Sunnyvale, California 94088
(408) 732-2400 TWX: 910-339-9280 TELEX: 34-6306

Printed in U.S.A. 4/85 06584A

TABLE OF CONTENTS

Am29300 Family Overview

Advanced Micro Devices has developed a new VLSI family to support very high performance applications in general purpose computation, intelligent peripheral controllers and array and digital signal processing—the Am29300 family.

The family features high performance, greatly increased functionality relative to earlier approaches, and a high degree of architectural flexibility.

32-BIT VLSI

Historically, the Am2901 made a radical departure from conventional MSI functions by integrating several elements of a CPU into a *vertical 4-bit slice*. The combination of memory and ALU logic in a single package offered the user added functionality, reduced package count and data-path width flexibility.

The Am29300 family reverses the trend of vertical slice partitioning by integrating *complete 32-bit functions* into single VLSI devices.

There are several reasons for the choice of a wider data path. First, cycle time is improved significantly if carry lookahead is contained entirely on the the chip. Second, certain powerful on-chip functions, such as the funnel shifter, priority encoder and mask generator are extremely difficult to expand when using vertical slices. Third, a higher level of integration leads to a more cost-effective system solution. The wider data path also affords greater I/O bandwidth, higher precision and increased memory addressability. These and other advantages contributed to the decision to make a family of *complete 32-bit functions* rather than slices.

The Am29300 family currently consists of five members:

- Am29332 32-Bit ALU
- Am29331 16-Bit Microinterruptible Sequencer
- Am29334 62×18 Dual-Access Four-Port Register File
- Am29325 32-Bit Floating Point Processor
- Am29323 32×32 Parallel Multiplier

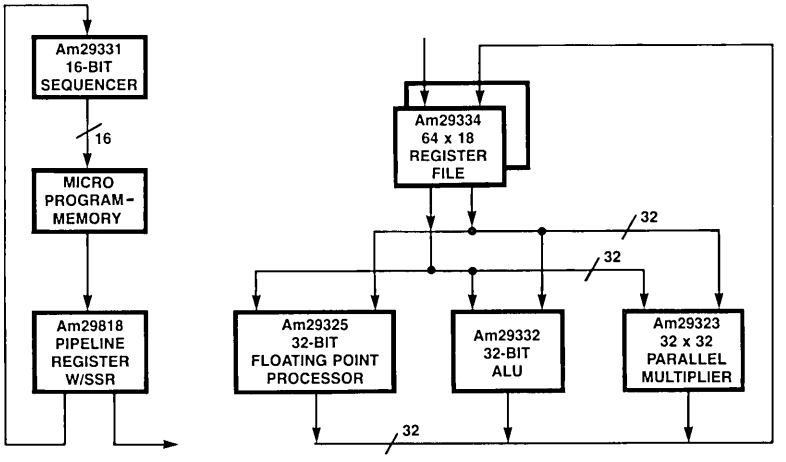


Figure 1. Am29300 Family High Performance System Block Diagram

FUNCTIONAL PARTITIONING MORE EFFICIENT

The Am29300 family departs from vertically partitioned bit-slice functions because it is divided into larger, horizontally partitioned building blocks. The ALU no longer contains a register file. Instead, there is a more flexible stand-alone register file, the Am29334, making expansion and regular addressing much easier.

The new partitioning results in a number of benefits. The user gets a powerful processor with two uncommitted input buses and gains the flexibility of adding storage elements to these buses. The overall organization is more structured. Also, a larger power budget is available for the register file thus making it faster and bigger than if it had been in the processor chip. Functional partitioning results in an open system, giving the designer the ability to easily connect external components, e.g. memory components or arithmetic accelerators. Also, each of the Am29300 components, while designed to work together in a system, can be used as a standalone functional block.

THREE-BUS FLOW-THROUGH ARCHITECTURE

The Am29300 family features a three-bus flow-through architecture. The Am29332 ALU, Am29325 Floating Point Processor and Am29323 Multiplier all have two input buses and one output bus. This contributes to high throughput by eliminating bus bottlenecks caused by turnaround delays. It provides unlimited register file expansion and regular addressability. Moreover, the unlimited bus accessibility gives the designer the ability to configure the optimal micro-architecture for the application. If the design objectives change, the micro-architecture can be easily reconfigured. The three-bus configuration also supports concurrent processing and pipelined architectures.

BALANCED TIMING

In previous generations of microprogrammed systems, the control path containing the sequencer has been the bottleneck because the sequencer was usually slower than the associated data path. Not so in the Am29300 family. The Am29331 sequencer has been designed so that the entire system timing is balanced between the control path and the data path leading to higher overall throughput.

POWERFUL INSTRUCTION SETS

Each device in the family executes its instructions in a single cycle.

The Am29332's instruction set is symmetric and *orthogonal.* Symmetric means that an operation that can be executed on port A can also be executed on port B and vice versa. Orthogonal means that all operations are independent of the data type. The Am29332 can operate both on multibyte data and on variable-width field data. This regularity of the Am29332's instruction set makes it easy to create "clean" interfaces to compilers for high level language support.

The Am29331's instruction set is comprised of instructions that resemble *high level language constructs.* This makes it possible to write *structured* microprograms.

COMPLETE INTERLOCKING FAULT DETECTION

The family supports both master/slave fault detection and data path parity to enhance system reliability by ensuring data integrity and correct hardware operation.

The system features byte parity checking on the inputs and byte parity generation on the outputs of the Am29332 ALU and the Am29323 Multiplier. Also, the organization of the Am29334 64×18 register file accommodates parity bits for each byte. The parity mechanism assures data path integrity.

Major functional blocks—the Am29332 ALU, Am29331 Sequencer and Am29323 Multiplier—also have master/slave fault detection to ensure correct device operation without having to carry parity through complex internal logic and without having to pay the resulting delay penalties. In master/slave mode, two functional units are connected in parallel with one unit performing the actual operation and the other checking the result, on a cycle-by-cycle, bit-by-bit basis.

The master is used in the normal data path. In the slave, however, all outputs become inputs, and the slave compares the outputs of the master with its own internally generated result. If the two don't match, an error signal is generated, which can trigger an interrupt at the microinstruction level. No specialized software is required. Also, the designer can choose to impose redundancy at the component or board level.

The parity and the master/slave provisions comprise a complete interlocking fault detection mechanism. Using cost-effective hardware rather than expensive software, they provide a comprehensive solution for fault tolerant systems.

PERFORMANCE/FLEXIBILITY/INTEGRATION

The Am29300 family achieves high performance and high integration but avoids architectural or pipelining restrictions. These become especially important in high performance parallel architectures or in emulations where the system is being optimized for particular instructions or processes.

The ECL-internal, TTL I/O Am29300 family minimizes the requirement for external components and achieves a *system* cycle time of well under 100 nsec.

32-bit bipolar building blocks debut at AMD

Alex Mendelsohn
Editor-in-Chief

The last six months have seen a procession of advanced microprocessors and peripheral support chips making the leap from NMOS fabrication technologies to CMOS. Although systems designers can now implement circuits with fast-running machines that dissipate less power than their NMOS forbears, true speed-demons still opt for bipolar devices.

Witness the success of the Advanced Micro Devices Type Am-29116, a microprogrammable 16-bit bipolar microprocessor whose 100 nanosecond microcycle bit slice speed has been an attractive calling card in recent years for designers seeking maximum system throughput. Most designers using the 29116 haven't felt compromised by power dissipation and supply requirements—the tradeoff has been a fair one.

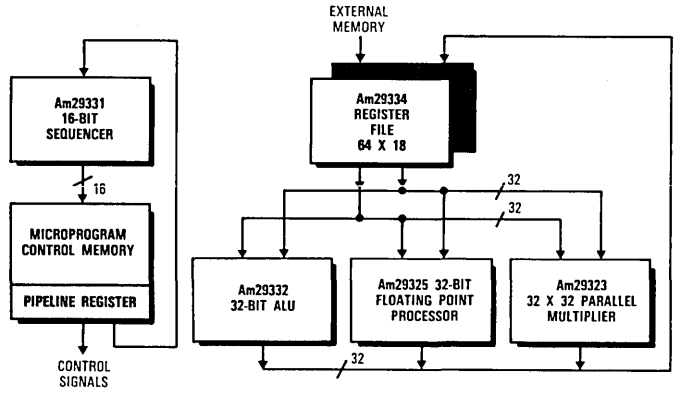
AMD is now at it again, but in addition to bipolar speed, AMD's latest chip set features an open 32-bit wide "building block" register file/ALU architecture that lends itself to unique general purpose implementations, freeing you from cast-in-silicon approaches.

Targeting designers looking for blazing speed for projects like parity-equipped fault tolerant processors, advanced graphics systems, image processors, large register file based RISC machines, and high-throughput simulators, AMD has just announced a "superset" of the flexibility of the venerable 29116. This first ECL-internal/TTL-I/O 32-bit architecture is the 29300 Family.

Partitioned for performance into five individual bipolar devices, the 29300 offers a one-chip ALU with access to three 32-bit buses. You, as a designer, can thus arrange your own unique system as you see fit. The 29300 busing provides a "flow through" architecture and virtually unlimited bus accessibility and register file expansion. No bidirectional busing is used, and apparently AMD chip designers were not concerned with conserving package pins.

An all-important orthogonal instruction set facilitates structured micrprogramming, permitting the machine to execute a number of functions on each microcycle in a regular symmetric way. Pins are available to tell whether an operation is byte width or a 16-, 24-, or 32-bit operation. No coding changes are required to perform at the byte or at the 32-bit level. The compiler is therefore very easy to generate, without exception handling complexities; a high level language interface is thus a "clean" one.

All instructions execute in single machine cycles. During one such cycle a 29300 system can do as much as it would take six or seven cycles to perform in one of today's MOS machines. For example, a shift and rotate could



be combined with logical-ORs, something a 68020, even with its on-chip cache, would need multiple cycles to perform.

Functionally, the 29300-family is horizontally partitioned to provide faster processing than previous vertically partitioned bit-slice approaches. Five bipolar VLSI circuits are to be introduced between now and next summer. AMD has already seen first silicon on one of the elements, a 32-bit math processor.

The five ICs are: a 32-bit arithmetic logic unit, dubbed the Am29332; a four-port dual-access 64-by-18 register file—the Am29334; the aforementioned high speed floating point processor, the Am29325; a Type Am29323 32-bit parallel multiplier with two Read and two Write ports; and lastly, a Type Am29331 16-bit microprogram sequencer. Let's take a look at each.

The 29332 arithmetic logic unit (ALU) is a 32-bit wide nonslice three bus IC that allows integration of functions that normally don't slice. Examples of these include shifters, priority encoders, and mask generators. Instructions are tailored to take advantage of these internal blocks (offering field logical operations or concatenation across word boundaries).

The Heart of the System

Cycle time for all 29332 ALU instructions are equal. Pipelined registers are avoided so that individual designers can build-in pipelining or not, according to their own schemes, paying no penalties for branching. The three bus architecture also allows ready design of parallel and reconfigurable architectures. The off-chip register file ensures unlimited expansion and regular addressability.

The 29332 also includes a unique 64-bit in/32-bit out funnel shifter block. It allows n-bit shift-up/down as well as a 32-bit barrel shifts (see *Integrated Circuits Magazine*, Jan./Feb. '84, page 34). The funnel shifter also permits 32-bit field extraction in conjunction with the mask generator. These unique functions can be combined with all logical instructions within the same cycle and with no increases in cycle time.

Shift control for the above functions can come from an external source or from the internal status register (generated on a previous instruction)—a useful feature for logical operations between non-aligned variable-length fields. It can also be used for floating point normalization.

Use of internal position and width status register fields can save eleven bits of microcode width. As mentioned previously,

A LOOK AT COMPETITIVE ALUs

In evaluating the new AMD 29300-family architecture, it may pay to look at some of the competitive devices recently introduced by other IC vendors. For example, Texas Instruments has been expanding their low-voltage high speed small-transistor advanced Schottky (AS) bipolar line to include a 20 MHz 8-bit slice.

Their 74AS888 features a parallel 8-bit ALU with expansion inputs and outputs, a 16 x 8 register file, and handles bit, byte, and word length operations. When used with their new 25 nanosecond 74AS890 microinstruction sequencer, architectures can be built without limit (i.e. 64-bits wide).

The 74AS890 controller has an address width of 14-bits, and can thus address up to 16,384 words of microcode. These ICs are designed to implement systems with narrow microcode word widths and very high throughput, and as such, should compete favorably.

Also in the realm of ALUs, but in MOS technologies, Analog Devices (Norwood, Massachusetts) has recently announced their 16-bit Type ADSP-1201. This device, as the name suggests, is targeted at digital signal processor (DSP) designers. Similarly, Weitek, of Santa Clara, California, introduced an NMOS two chip set early this year; their 32-bit data path Type WTL1032 multiplier and WTL1033 ALU.

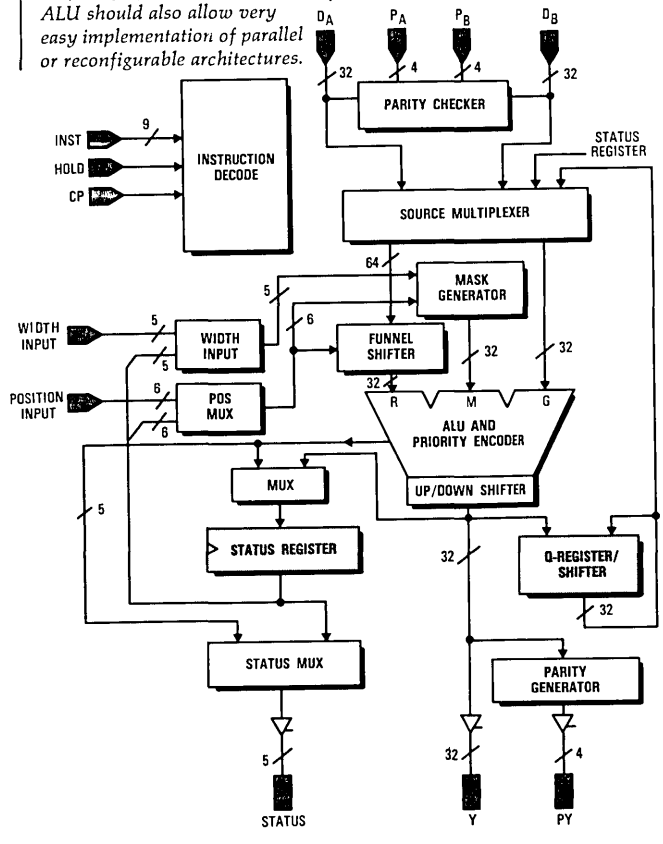
Both Weitek's and Analog Devices' ICs, while excellent for DSP applications, are somewhat limited for non-DSP circuits because of their extensive pipelining. Both include registered inputs and outputs, thus they both require more than one cycle to get results "off" the chip. Operands are entered, and the results taken, from the Weitek chip in about 125 nanoseconds, for example.

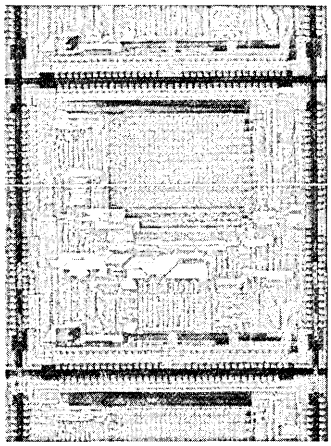
The ADSP-1201 also includes a single port 8-word register file in each data path, however this precludes expandability. The chip does include a barrel shifter, but use of the shifter and the ALU combinatorially in the same cycle isn't possible.

In contrast, 29332 ALU designers avoided the use of pipelining. There are no built-in penalties for branching. The simplicity of the three-bus ALU should also allow very easy implementation of parallel or reconfigurable architectures.

Another consideration in the comparison: the 29332's off-chip register file allows unlimited and regular addressability. In contrast, the ADSP-1201 has a single port eight-word register file in each input data path. These are not expandable.

The 29332 supports one-, two-, three-, and four-byte data for arithmetic and logic functions as well as multiprecision arithmetic and multiple-bit shift operations. Neither Analog Devices' nor Weitek's ALUs support all data types for arithmetic operations. Neither can they support field logical operations as used in graphics.





The vanguard of the 29300 family is the 29325 floating point processor. Here's a photomicrograph of first silicon.

the 29332 also supports one-, two-, three-, or four-byte data as well as multiprecision arithmetic and multiple bit shift operations.

For logical operations, the 29332 can accommodate variable length fields up to 32-bits. When fewer than four bytes are selected, the unselected bits are passed to the destination without modification. Support of all data types is highly important for arithmetic operations; field logical operations are very necessary for applications such as graphics.

Support is also provided for two-bit at a time modified Booth's algorithm multiplication and one-bit at a time divide with both signed and unsigned integers. Parity checking on data inputs and generation on outputs, plus master/slave fault detection enhances applicability in fault tolerant systems.

Separate Register File

The Type 29334, the 20 nanosecond access time RAM register file chip, supports these parity equipped designs. A byte parity storage feature, with a width of 18-bits, provides consistency with the ALU for parity check and generation. No other multiport register file presently on the market offers this parity storage.

The 29334, with its two Read and two Write ports, can read and write simultaneously on both. It is cascadable to support wider word widths or to form deeper register files, or both. You can use multiple 29334's in an interleaved configuration, or you can build one file as high and as wide as you like. Write enable timing and multiplexer selection are derived from a single-phase clock; the MUX eliminates one "layer" of I/O delay. Also, individual byte write enables allow choice of either an 8- or 16-bit data interface.

Math Chip Here and Now

The single-chip 144-pin Type 29325 floating point processor—the first VLSI family member to emerge from fab—performs very fast 32-bit single precision addition, subtraction, and multiplication. It conforms to the proposed IEEE P754 standard and also to Digital Equipment Corporation's (DEC) format.

Options for conversion between the 32-bit integer format and floating point are available, as are operations for converting between IEEE and DEC. Executing all instructions in a single cycle, the 29325's throughput equates to 8 MHz, regardless of the algorithm.

The 29325 features three 32-bit wide non-multiplexed buses for high I/O bandwidth. The use of two 32-bit operand feedforward data paths on-chip support accumulation operations, including sum-of-products and Newton-Raphson division. All buses are registered and each has a clock enable. Registers can be independently transparent to eliminate unwanted pipelining if desired. Software synchronization of pipelining is not needed because there are no multi-stage pipeline structures.

Go Forth and Multiply

Although the Type 29323 multiplier chip is still in the design stage, and no working silicon has yet been seen, AMD chip designers are confident that the device will be able to perform a 32 by 32 multiply in only two cycles. They're hoping to achieve an 80 nanosecond clock-to-clock multiply time.

With two 32-bit input and one 32-bit output ports, there is no need to multiplex operands. The chip will be controlled by only one clock with individual register enables, thus leading to simple timing requirements. Dual input port registers will enable multiprecision multiplication (a 32-bit multiply will occur in one cycle; a 64-bit in four).

Like the floating point chip, the 29323 multiplier's registers can be made independently transparent tc eliminate unwanted pipeline delays in non-pipelined systems. A master/slave mode allows two 29323's to operate in parallel. A parity check/generate feature will catch interdevice errors.

Advanced Program Control

The remaining chip in the 29300-family will be the dual bus 29331 microprogram sequencer. Controlling the sequence of microinstructions stored in microprogram memory, the 29331 aids structured microprogramming, handling sequential execution, branches, subroutines, and loops. It can access up to 64 Kwords of microcode, and integrates otherwise external critical-path conditional

test logic into internal high speed gates to improve microcycle time. There is no branching penalty.

The 29331 generates inequality evaluation branch conditions from four ALU status bits. It features an eight external-test-condition multiplexer plus parity control. An address comparator allows breakpoint in the microcode for debug or for gathering run-time statistics. This latter feature is something AMD has identified that most systems builders want but, so far, no vendor has implemented in silicon.

Other 29331 features include four sets of 4-bit multiway inputs to implement table look-up or to use external conditions as part of a branch address, master/slave error checking for parallel sequencer operation, real-time interrupt support, trap handling at any microinstruction boundary, and a 32-level stack.

The latter provides the ability to support interrupts and loops as well as subroutine nesting. The stack can be read to support diagnostics or to run multi-tasking at the micro-architecture level. The chip's instruction set is designed to resemble high level language constructs.

Development Support

AMD expects most customers will use Tektronix, Hewlett-Packard, or AMD development systems for microcode development, and AMD is introducing "M29" software that will run on VAX-size mini.

M29 uses a description language that can describe a variety of architectures. It consists of three programs: a microinstruction definition program, an assembler, and a relocation linker.

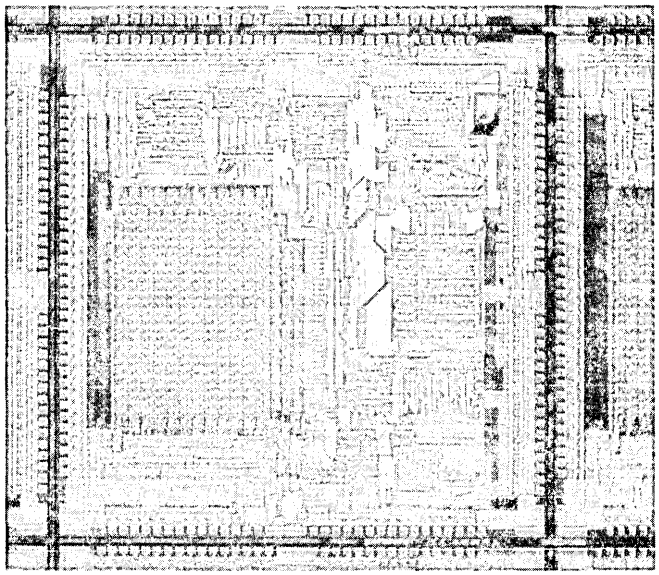
The definition program creates a file that describes each microinstruction field by field, defining its name and length, its fields and variations in format, and allowable values for each field. The file allows the assembler to be retargeted to support many different instruction formats.

The assembler allows you to create microcode in several styles depending on the amount of effort invested in the design of macros. The linker for the assembler is used to relocate assembly modules and link them together. Its output can be loaded into a writable control store or burned into PROMs.

For more details call AMD at 408-732-2400, or use the Reader Service card. ■

32-Bit ICs Enhance Array Processor Performance

by Dave Wilson, Executive Editor



Photomicrograph of AMD's 29325 floating point processor.

Future high performance processors/controllers require faster processing rates, higher machine densities, and greater system reliability. The need for virtual memory support, increased memory bandwidth and improved precision means a growing demand for 32-bit performance. Advanced Micro Devices' (Sunnyvale, CA) Am29300 family has been developed to address these needs in general purpose computation, intelligent peripheral control, and array and digital signal processing applications.

The Am29300 family evolved from the industry standard Am2900 bit-sliced family. A great number of the functional enhancements are the result of user feedback from existing Am2900-based designs. On the other hand, the Am29300 family has been designed from the ground up for higher performance and architectural flexibility. The Am29300 devices have internal ECL circuitry for speed, yet maintain TTL compatible inputs/outputs for ease of interface. A 32-bit Am29300 microprogrammed system has a system microcycle time of 70 to 80 nsec. In addition, the devices have regular and orthogonal instruction sets and contain built-in primitives to tackle crucial system issues such as fault tolerance/detection.

AMD offers several support chips for Am29300-based designs. For instance, systems requiring a 32-bit data path can be configured with the following devices: the Am29332 Integer Processor, a 32-bit arithmetic/logic and shift unit with built-in support for variable byte and bit field data; the Am29334 Register File, a true dual-ported register file which allows simultaneous read and write accesses, organized as 64 words by 18 bits; the Am29323 Parallel Multiplier, a 32 × 32 parallel multiplier capable of multiple cycle expansion to 64 × 64 and 128 × 128 without the use of external logic; and the Am29325 Floating Point Processor, which performs single cycle addition, subtraction, multiplication and conversions, using either the single precision IEEE or DEC format. Each of the above devices can be used in conjunction with or independent of the others. These devices can be configured in a variety of ways to tailor them to a specific application. All of the data path elements have single cycle instructions. The microinstructions are typically supplied by the control path. The key control path element is the Am29331 Microprogram Sequencer which supplies the next address to the control memory. The 16-bit Am29331 is also capable of handling interrupts or traps at the microinstruction level.

Historically, the Am2900 devices have been partitioned vertically, combining register file and ALU in a single package. The Am29300 devices, however, are partitioned horizontally, so that the register file is separated from the rest of the data path elements. The functional partitioning has two advantages. First, it allows for an easily expandable register file space. Second, it also enables arithmetic accelerators to add to the data path.

A major disadvantage of a bit-sliced architecture is the time lost in transmitting carries from one chip to another. To avoid this, the Am29300 family arithmetic elements are constructed with full internal 32-bit data paths. Although the ALU has limited carry capability for cascading, it will normally perform multi-precision expansion through multiple cycle opera-

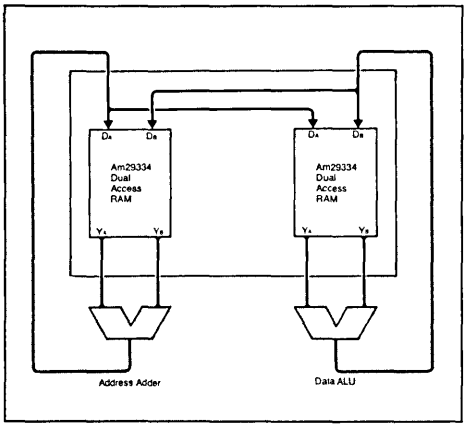


Figure 1: RAM with 4 read and 2 write ports.

tions. The other data path elements use this scheme exclusively.

A second benefit of the full 32-bit data path elements is that they can include functions not easily sliced. Two classic examples of this are shift arrays and multipliers. Both of these require an unacceptable amount of information to be transferred between slices. Other functions, such as prioritization and mask generation for byte and word operation, while feasible, expand clumsily. All these functions are provided in the Am29300 family, either in the ALU, or in the 32-bit multiplier.

Three-Bus Flow Through Architecture

In order to fully exploit the 32-bit data path devices, it is necessary to provide adequate data transfer bandwidth. In the Am29300 family this is achieved through the use of a three-bus architecture. The Am29334 Register File is a true two-ported file, allowing simultaneous access from each port. Output latches are provided to allow read and write operation within a single clock cycle. Each of the data path elements has two 32-bit operand input buses which can be sourced from the register file. The data path elements also have a 32-bit result bus which can return data to one input of the register file. With this organization, a three-address register to register operation may be completed within a single clock cycle.

Two-register files may be used to achieve still higher bandwidth. Connecting the input ports in parallel, and writing duplicate data into the two files, allows four operands to be sourced simultaneously from a single database. Two results may also be written into the file simultaneously. This provides adequate data transfer for two groups of arithmetic elements to operate concurrently (**Figure 1**). The flexibility of this three-bus architecture also allows the use of these parts in other configurations. In a signal or array processing application, the multiplier and ALU may be placed in series rather than parallel. This provides a "free operand," allowing the three-operand summation of products operation to proceed at maximum speed.

The cycle time of a microprogrammed system is dependent on both the control path (i.e., sequencer and microprogram memory) and the data path (i.e., register file and ALU). Traditionally, the system bottleneck has been the control path, especially the timing paths associated with conditional branching. The 16-bit Am29331 Microprogram Sequencer has been optimized for speed, so that the data path and control path timing are balanced. The previously external condition code multiplexer, test logic generator and polarity control logic (usually the system critical path), have been integrated on chip. Moreover, the Am29331 has several built-in features which enable it to respond to external stimuli with minimum latency. The sequencer can perform a 16-way branch, dependent on the simultaneous occurances of four external test conditions. The Am29331 Microprogram Sequencer can also handle interrupts or traps at the micro-level.

The system ARM concept (Availability, Reliability and Maintainability) is becoming increasingly important. The Am29300 addresses the problem of fault detection at the device level by a combination of two techniques – parity and master/slave. Parity at the byte level is generated on the 32-bit result bus of the data path elements, stored in the Am29334 Register File, and checked again going into any of the operand buses of the data path elements. Thus any interconnection failure in the data bus can be detected. The choice of even parity scheme also allows detection of an open TTL bus which defaults to high impedance all "ones" state, an error condition. For functional verification, a master/slave mode of operation permits two units to be connected in parallel, with one unit actually performing the computation and the other checking the results on a cycle by cycle basis. The slave unit therefore verifies correct operation of the master. In addition, the master unit checks its internal result with the data on the output bus to ensure that no other device is driving the external bus when it is not supposed to be. Any fault detected can trigger an interrupt at the microinstruction level. Unlike previous redundant schemes, no specialized software is required. No system degradation results from the communication between the redundant functional units. This combination of parity checking and master/slave operation, which uses cost-effective hardware, rather than expensive software, is the key to future redundant system design.

The functional and performance requirements of a general purpose supermini-computer and a digital signal processor are vastly different. Yet with functional partitioning and a simple three-bus architecture, the Am29300 devices are suited to address the needs of a diverse spectrum of applications. **Figure 2** depicts an example of a microprogrammed supermini built out of Am29300 components. The data path consists of the Am29332 Integer Processor, the Am29323 Parallel Multiplier as an accelerator, and the Am29334 Register File. In this configuration, address calculation and data computation are performed in series. Alternatively, the Am29334 can be paralleled to yield effectively a six-ported register file, allowing four read accesses and two write accesses per microcycle. Another Am29332 can be dedicated to perform address computation concurrent with the normal ALU execution, sharing the register space. With a 70 to 80 nsec microcycle time, a processor/controller subsystem capable of several times the performance of a typical supermini can be built with the Am29300 parts, occupying far less board space and dissipating significantly less power.

Figure 3 is a block diagram of a small array processor using the Am29325. A high-speed multi-port memory is used to provide storage for operands such that they may be accessed in simultaneous pairs. These operands may originate in the data memory, or may be intermediate results

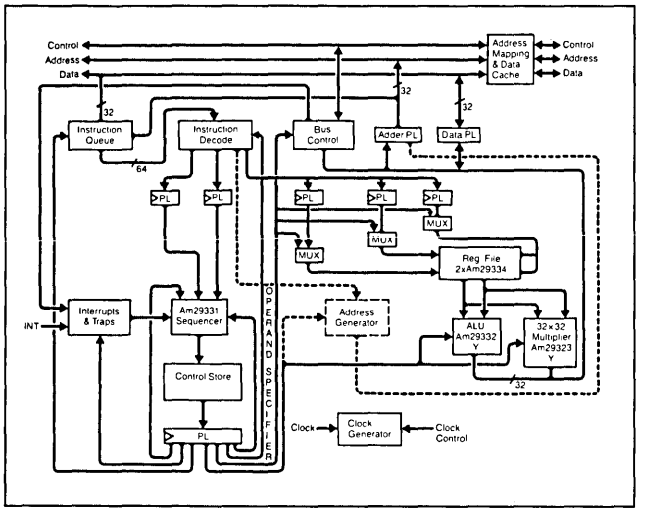


Figure 2: Am29300-based supermini emulation.

from the processor. One of these operands may be replaced with a value drawn from a non-volatile coefficient store.

The array processor is microprogram controlled, with memory addresses being derived directly from the microcode. This is probably inefficient for large programs, and some form of microprogammed address generator would need to be added. The interface to the host processor is deliberately undefined, as this is user dependent.

As a benchmark, this processor can perform FFT butterflies in the canonical time of 10 cycles. At a 100 nsec cycle time, this permits one butterfly every 1 μsec, or a 1024-pt complex transform in 5.12 msec. A simple modification to the architecture allows a second Am29325 to be incorporated to give a complex arithmetic processor. This doubles the throughput for the FFT, reducing the computation for the 1024-pt transform to 2.56 msec.

While the Am29325 only provides single-precision floating point operation, the Am29300 family also provides bus-compatible devices which may be used to enhance the capabilities of the array processor described. The Am29332 Integer Processor offers a wide range of arithmetic, logic and shift facilities. This device may be operated with a reduced width data path, allowing words of 1 to 4 bytes. The internal architecture is designed for efficient programming of floating point operations, and may therefore be used to support the Am29325 with double-precision operations. To assist in double-precision floating point multiplication, or for integer multiplication, the Am29323 32-bit Multiplier provides 32 × 32-bit multiplication in a single cycle, and has internal facilities for multi-cycle expansion to 128 × 128.

These additional arithmetic elements have the same 32-bit, three-bus architecture as the Am29325. This allows them to be added in parallel. The routing of operands to the appropriate arithmetic element is a simple microcode task.

The horizontal partitioning of this new family of parts has resulted in a number of benefits. First, the user gains the flexibility of adding storage elements to two uncommitted output buses from the processor. Second, more power budget is available for the register file making it faster and bigger than if it had been in the processor chip. The family addresses a number of crucial system issues such as fault detection, support of high-level languages in systems programming and large register file-based architectures like RISC. DD

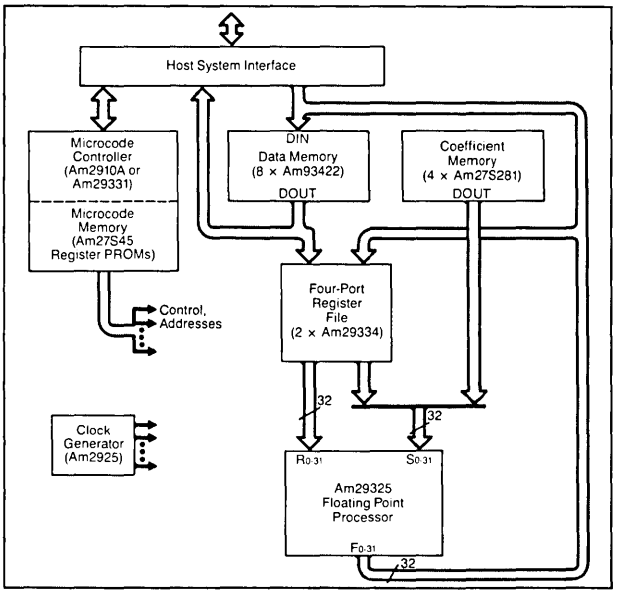


Figure 3: Am29300-based array processor.

References:

32-Bit Building Blocks for High Performance Processor/Controller, Paul Chu, Advanced Micro Devices, Sunnyvale, CA

A Very High Speed Floating Point Processor, B.J. New. Advanced Micro Devices, Sunnyvale, CA.

Microprogrammable chips blend top performance with 32-bit structures

Broken down into 32-bit functional blocks instead of being sliced into multiple-bit sections, five VLSI bipolar chips match a supermini's speed.

Designers of systems and subsystems for high-speed computation, intelligent peripheral control, and array and digital signal processing typically need higher performance than standard microcomputer parts can deliver. The required precision, speed, and virtual memory support has to some degree been supplied by dedicated VLSI components that are customized for particular applications. Yet an overwhelming need still remains for a set of building blocks that can bring extremely high performance to a large assortment of applications.

A new approach extends the bit-slice concept to 32 bits and also satisfies system designs that require cycle times of less than 100 ns. With a family of five VLSI chips, designers of microprogrammed systems can count on cycle times of 70 to 80 ns, using merely a handful of components. The building blocks for 32-bit systems functionally partition the chips and separate the register file from the rest of the data path.

The following two articles first explore the key members of the Am29300 family and then focus on a floating-point processor, which is the first chip scheduled for sampling. Details are given on how to use the chip and other devices in the series to build a fast Fourier transform computer, as well as more general-purpose digital signal-processing circuits.

The Am29300 family addresses the problem of fault detection through an interlocking checking scheme—parity and master-slave. Byte parity is generated, stored, and then checked on all data-path elements as a means of detecting interconnection failures. Moreover, to verify certain functions, the master-slave operating mode permits two units to be connected in parallel, with one unit actually handling the computation and the other checking the result cycle by cycle.

Detecting a fault triggers an interrupt at the microinstruction level. Unlike previous redundant schemes, no specialized software is required. Furthermore, communication among the redundant functional units causes no system degradation.

The five chips form a strong foundation for any system designer's work. For instance, a 16-bit sequencer can handle interrupts and traps at the microinstruction level. There is

Paul Chu and **Bernard J. New**
Advanced Micro Devices Inc.

Paul Chu is now department manager of programmable processors in the product planning division of Advanced Micro Devices in Sunnyvale, Calif. He holds several patents for microprogrammable devices and has a BSEE and an MSEE from Stanford University.

As product planning manager for array processors at AMD, Bernard J. New is responsible for conceiving and defining arithmetic computing devices. The holder of a BSc (Hons) in electronic engineering from England's University of Birmingham, New has two patents on Am29500 products.

Microprogrammable 32-bit chips

also a combined ALU and shifter that internally supports variable byte and bit fields. Together with the ALU-shifter chip, a true dual-port register file, organized as 64 words by 18 bits, can build a basic system. The register file, designed for simultaneous read and write accesses, is separated from the data-path elements, thereby avoiding the problem of addressing an internal register file differently from external memory. The benefits of that separation are uniform register addressing and unlimited depth expansion.

Two accelerator chips—a floating-point processor and a parallel multiplier—can be added to the basic system to raise the number of functions and cut processing time. The 32-by-32-bit parallel multiplier can, on successive cycles, expand to 64 by 64 or 128 by 128 bits, without help from external logic. For its part, the math chip can tackle single-cycle addition, multiplication, subtraction, and conversions—all in single-precision IEEE or DEC formats.

Because of functional partitioning, a three-bus flow-through architecture was chosen as the data path. For maximum bus accessibility, all data-path elements—the integer processor and the parallel multiplier, for example—share two operand and one result bus. The flow-through architecture not only transfers data extremely quickly but also avoids the complex timing control needed to turn around bidirectional buses. Above all, the simplicity of the three-bus architecture allows these components to be configured in a variety of ways to optimize micro-architectures for different jobs.

Bipolar building blocks deliver supermini speed to microcoded systems

As CMOS processes start to encroach on the performance of bipolar circuits, bipolar technology is taking the next step to keep itself in the lead for the highest speed systems. A family of five bipolar VLSI computational circuits—fabricated with a scaled, ion-implanted, oxide-isolated process and three levels of metal interconnections for high density—provides a set of functionally partitioned microprogrammable VLSI building blocks for systems such as superminicomputers, digital signal processors, high-speed controllers, and many others. The modularity of the system functions ensures that the chips can meet the performance requirements of a general-purpose superminicomputer, as well as those of an image processor, which are radically different from each other.

Included in the family are three parts that form the core of a general-purpose microprogrammed system: a 32-bit arithmetic and logic unit (ALU), a 16-bit microprogram sequencer, and a 64-by-18 four-port, dual-access RAM. And, for systems that do a large number of multiplications or floating-point

Dhaval Ajmera, Ole Moller, and **David Sorensen**
Advanced Micro Devices Inc.

Since the beginning of last year, Dhaval Ajmera has been a design engineer in product planning at Advanced Micro Devices in Sunnyvale, Calif. He holds an MSEE from the University of Florida.

Øle Moller is also a design engineer in AMD's product planning operation. He holds an MSEE from the Technical University of Denmark.

Another engineer in product planning, David Sorensen specializes in programmable processors. He holds a BSEE from Arizona State University.

operations, two performance accelerators—a 32-by-32-bit multiplier and a 32-bit floating-point processor will be available to tie onto the buses (see Design Entry, p. 246).

The chips offer high performance, a flexible architecture, and microprogrammability, and even address the problem of fault detection for data integrity. These circuits can thus support an extremely fast microcycle—about 80 ns (projected). That high speed is the result of several design considerations: Each part is designed internally with emitter-coupled logic but has TTL-compatible inputs and outputs. Second, more power was allocated to the logic circuits used in the critical paths than for logic in the noncritical paths on each chip, to maximize the speed. Third, by integrating highly specialized logic on chip it is possible to execute very complex operations in a single cycle.

The microprogrammability of this chip set offers several benefits to the system designer. It provides a structured and systematic approach for implementing the control mechanism of the system, and like the bit slices, it allows the instruction set to be customized to suit the designer's application (see "Architectural Limitations of Bit Slices," opposite). And several versions of the initial design can be tested, or current designs can be enhanced simply by changing the microcode.

Thus, the functionally partitioned Am29300 family overcomes all of the performance penalties of bit-slice structures, while maintaining its ability to form a wide variety of architectures. Even though the chips are designed to work together as a family, each can also be used independently in an application that requires its unique capabilities.

Pipelines are out

The flexibility of the Am29300 family is largely due to a decision not to place pipeline stages within the functional blocks. Not including the pipeline registers inside incurs some off-chip delays. This is a small price to pay to allow system designers to optimize the pipeline structure for their individual needs. Moving the register file out of the functional block for the ALU also slows things down. At the same time it does not force a fixed register size on the user, enabling systems to be created with dedicated registers, register windows, or register banks—all with neither fixed depth nor width.

Additionally, the high level of integration helps eliminate the propagation delays often encountered when signals must go from chip to chip. The use of VLSI also results in fewer parts at the system level, which, in turn, conserves power (usually many watts in the case of bipolar systems) and board space. Lastly, a complete 32-bit solution is provided for applications that require increased precision for arithmetic operations, high memory bandwidth, and a

Architectural limitations of bit slices

The limited performance of bit-slice circuits can be improved by increasing the width of the slices. That higher level of integration results in higher performance by reducing the number of off-chip delays while preserving the flexibility that has made bit-slice systems so attractive. However, as higher levels of integration become possible, two inherent problems with bit-slice architectures will limit their ultimate speed. The first involves the off-chip delays inherent in cascading. For example, the carry chain is usually the slowest path of an ALU. Breaking this chain between slices introduces off-chip delays into the critical path.

The second problem is that the functional needs of many systems do not slice well. Barrel shifters and prioritizers are especially difficult to cascade. Unfortunately, the ability to perform N-bit shifts and locate the position of leading 1s are of greatest importance in applications that require heavy number crunching and manipulation of data fields, such as image processing, graphics, database management, and controllers. These are precisely the applications whose need for speed forces the use of bit-slice devices. The system performance is compromised not only because these operations must be done bit by bit, but also because many high speed algorithms cannot be efficiently implemented.

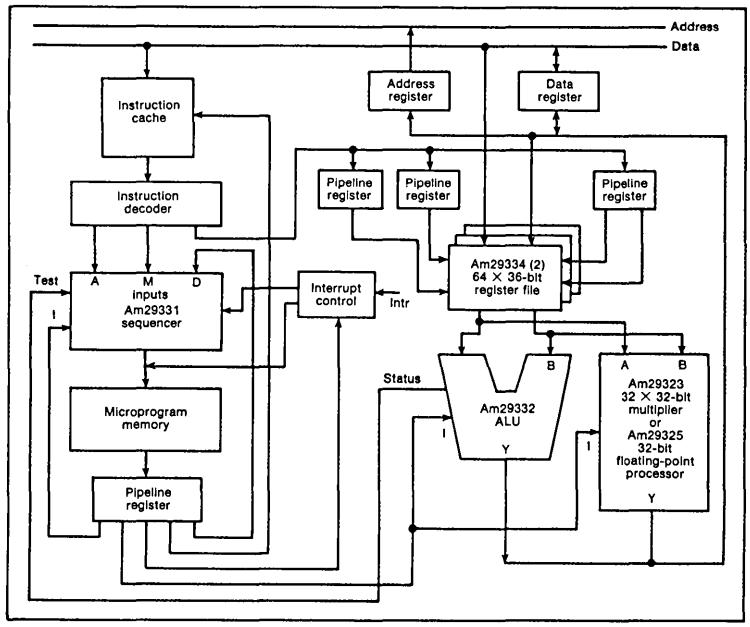
large addressing capability (4 billion bytes) to support virtual memory systems (Fig. 1).

The performance of a system depends, not just on its raw computing speed, but on its ability to respond to events such as interrupts and traps. For example, the Am29331 sequencer responds to both interrupts and traps at the microprogram level very quickly, and its response is completely transparent to the interrupted microroutine. Also, the Am29332 ALU indirectly supports the handling of these events by allowing its internal state to be saved or restored.

The Am29332, a noncascadable 32-bit-wide, ALU, provides fast number crunching, high data transfer rates, and powerful bit-manipulation capabilities. Intended to be used with the Am29334 dual-ported RAM, which serves as an external register file, the ALU has two 32-bit input buses (DA and DB) and one 32-bit output bus (Y).

Internally, the device has a 32-bit data path that interconnects its various functional blocks. These blocks include various shifters and multiplexers, a mask generator, a funnel shifter, the ALU proper, a priority encoder, a parity generator and checker, a master-slave comparator, and the status and Q registers (Fig. 2). The ALU proper has three 32-bit inputs: R, S and M. The R input comes from the funnel shifter, the M input from the mask generator, and the S input from a variety of sources —the DA or DB buses, status register, or the Q register.

The power and flexibility of the Am29332 comes partly from its ability to perform operations on various data types. It can operate on



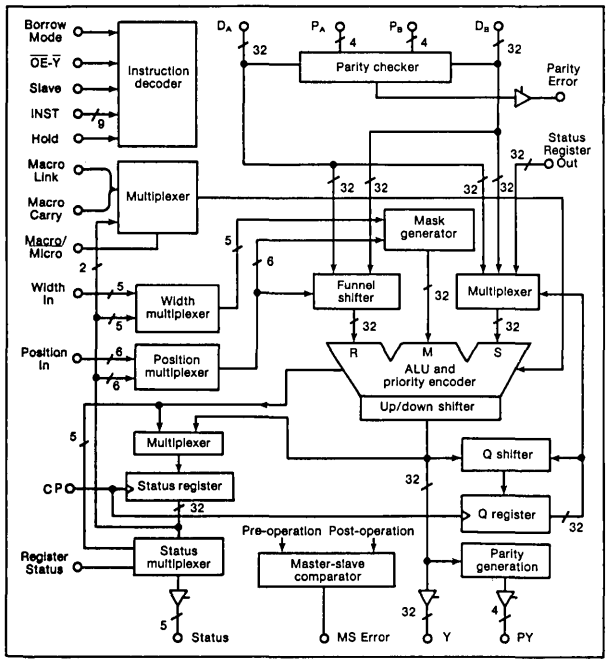
1. A conventional CPU, built with Am29300 building blocks, forms the focal point of an extremely compact system that cycles as fast as 80 ns.

variable bytes, variable-length bit fields, or single bits. This is made possible by the internal mask generator, which creates a 32-bit mask for each instruction (with no time overhead). The mask is used as an additional operand in each instruction to allow the operation on only selected data widths.

The type of mask generated depends on the type of instruction. For instructions that operate on variable bytes (1, 2, 3 or 4 bytes) the mask is a fence of 1s (bit 0 aligned) for all low-order selected bytes with a fence of 0s for all high-order unselected bytes. Instructions that operate on variable-length bit fields require a mask that is a string of contiguous 1s for all selected bit positions and 0s for all unselected bit positions. In cases where the field exceeds the 32-bit boundary, the mask does not wrap around, thus allowing operation on a contiguous field across a word boundary. For instructions that operate on a single bit, the mask is a 1 for the selected bit position and 0s for the other unselected bits.

For most single-operand instructions, the unselected bit positions pass the corresponding bits of the operand unmodified. For most two-operand instructions, the unselected bit positions pass the corresponding bits of the operand unmodified on the DB input. Thus, for two-operand instructions the mask allows the merging of two operands in a single cycle. In addition to being used internally, the mask can be sent out over the Y bus, permitting the generator to be used as a pattern generator for testing purposes.

To speed various mathematical and logical operations, many circuits have started to in-



2. To connect its various internal functional blocks, the Am29332 ALU employs a 32-bit bus. Among the chip's major features are a 64-bit funnel shifter, parity checking and generation, and a basic 32-bit ALU that has three input ports. The processor also has three 32-bit ports through which it transfers data into and out of the chip.

clude a barrel shifter, which has an N-bit input and an N-bit output. The barrel shifter would be used to shift or rotate the operand either up or down from 0 to N bits in a single cycle. Such high-speed shifting is very useful in operations such as the normalization of a mantissa for floating-point arithmetic or in applications in which the packing and unpacking of data are frequent operations.

However, a more useful circuit is a funnel shifter, which can be thought of as having two N-bit inputs and one N-bit output. Just such a circuit (with 32-bit-wide ports) was included on the 29332. The circuit can perform all the operations of a barrel shifter with capabilities extended to two operands instead of one. In addition, it can extract a 32-bit contiguous field across its two operands, a function very useful in several graphics applications. And any of its operations can be followed by a logical operation, with both completed in a single cycle.

Setting the priorities

Prioritization, useful to control N-way branches, perform normalizations, and in graphic operations such as polygon fills, can readily be handled by the ALU chip. The built-in priority encoder sends out a 5-bit binary weighted code that signifies the relative position of the most-significant 1 from the most-significant bit position of the byte width selected. That allows prioritization on either 8-, 16-, 24-, or 32-bit operands. The priority encoder output can be passed on to the Y bus or stored in the status register.

If, for example, prioritization is used to normalize a mantissa during a floating-point arithmetic operation, it requires two cycles. In the first, the mantissa is prioritized to determine the number of leading 0s that need to be stripped off. In the next cycle, the mantissa is shifted up by the amount specified by the priority encoder output.

Relevant information for each operation performed by the chip is stored in the 32-bit status register after each microcycle. Each byte of the status word holds different information. The least-significant byte holds the position specifier. The next most-significant byte holds the width specifier and three other bits that are used to test the comparison of unsigned and signed operands. The next byte contains the Carry, Negative, Overflow, Link, Zero, M and S flags. The M flag stores the multiplier bit for multiply or the sign compare bit for signed division, and the S flag stores the sign of the partial remainder for unsigned division. The most significant byte stores the nibble carries for BCD operations.

The states of the Carry, Negative, Overflow, Link and Zero flags are available on the status pins, and the status multiplexer allows the user to select either the status of the previous instruction (register status) or the status of the current instruction (raw status) to appear on the status pins. The raw status could be used to update an external macro status register. This also allows branching at either the micro- or macro-level.

The Q shifter and Q register are primarily used to assemble the partial product or partial quotient in multiplication and division operations. Variable bytes of the status and Q register can either be loaded via the DA and DB inputs or can be read over the Y bus. Thus saving and restoring of the registers allows efficient interrupt handling after any microcycle. It is also possible to inhibit the update of both these registers by asserting the Hold pin.

Powerful and orthogonal instructions

The power of the ALU chip's instruction set comes directly from the integration of several functional blocks mentioned earlier. The commands are symmetrical as well as orthogonal, to make it easier for a compiler to generate efficient code. Thus, any operation on the DA input is also possible on the DB input, and each instruction is completely independent of its data type.

Three-fourths of the instruction set consists of variable byte-width (one, two, three or four) operand instructions. The byte-width is selected by two bits in the instruction. For these operands, the instruction set supports all conventional arithmetic, logical and shift operations. Arithmetic operations can be performed on both signed and unsigned binary integers.

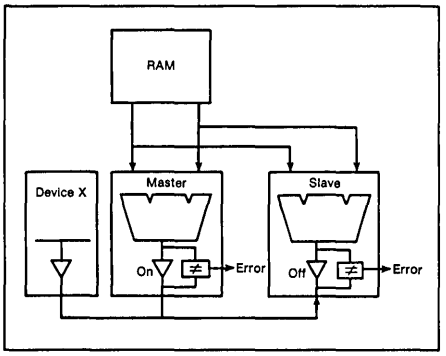
Additionally, the instruction set supports multiprecision arithmetic such as addition with carrying and subtraction with carrying or

Microprogrammable 32-bit chips

borrowing. For all subtract operations it provides the convenience of using borrowing instead of carrying by asserting the borrow pin. In this mode the carry flag is updated with the true Borrow. To allow efficient execution of macroinstructions the chip contains a Macro mode pin. When the chip asserts this pin, it allows the external Macro-Carry and Macro-Link bits instead of their microcounterparts to participate in the operation.

Instructions that execute algorithms for the multiplication and division of signed and unsigned integers are multiple cycles are also provided. For multiplication, the circuit supports the modified Booth algorithm, yielding two product bits in one cycle. Both single-precision and multiprecision division of signed and unsigned integers are supported at the rate of one quotient bit in every cycle.

Besides binary integers the instruction set provides basic arithmetic operations for binary-coded decimal (BCD) numbers. By operating directly on the decimal numbers created in most business applications, significant processing time is saved by eliminating the need to convert from binary to BCD and vice versa. Also, the round-off errors involved in converting from one base to the other are eliminated.



3. To help ensure system integrity, two Am29332 processors can be set for master and slave operation. Both chips perform the same operation in parallel, and any difference in their results is flagged as an error. The master also checks its internal result against the data on the output bus to make sure that no other device (such as device X) is turned on at the same time.

The last group of instructions was created to support variable-length bit fields (1 to 32) and single-bit operands. The position and width of the field can be specified by either the position and width inputs or by fields in the status register, thereby saving bits in the microcode. Most of the time, the position and width are determined dynamically. It is therefore difficult to supply them via the microinstructions. For single bit operations only the position specifier is needed.

Bit-manipulation instructions include setting, resetting, or extracting a single bit of the operand or the status register. Logical operations on either aligned or nonaligned fields in the two operands include OR, AND, NOT and XOR. In the case of nonaligned fields it is assumed that at least one of the fields is aligned to bit position 0. It is also possible to extract a field from one operand and insert it into another operand or extract a field across two operands.

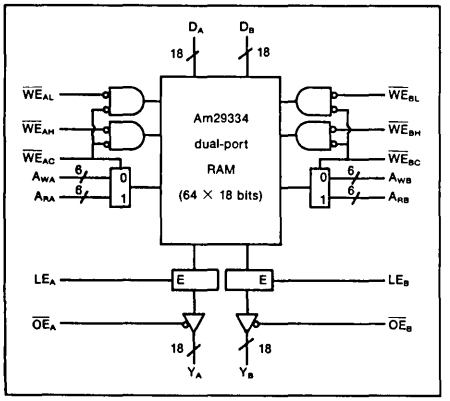
Enhancing system integrity

The growing need for data integrity has been addressed at both the system and the chip level by including hardware for fault detection. During calculations, byte-wide even parity is generated for the data result by the ALU and stored with the data in the external RAM. Byte-wide even parity is also checked at the ALU inputs and any error is flagged.

Even parity is specifically used to check for a floating TTL bus. Thus, all interchip connections are checked out. In addition, hardware for functional verification is also provided on the sequencer and the ALU functional verification can be implemented by using two similar devices in the master and slave mode (Fig. 3). In that setup, both chips perform the same operation, with any difference in their outputs being flagged as an error. The slave-mode chip's bidirectional buses operate in their input mode, allowing the master to compare its own internal result with that of the slave on every cycle. Additionally, the master checks the output bus to

make sure that no other device is turned on at the same time.

As mentioned earlier, the ALU architecture was designed to use an external register file. Keeping the file external to the chip permits the user to expand it to meet any system need. The Am29334, a high-speed 64-word-by-18-bit dual-access RAM, provides two independent data input ports and two independent data output ports (Fig. 4). Each port can be read from or written to using the separate inputs and outputs. The two accesses are independent except for the case when simultaneous write operations are done to the same word—in which case the result is undefined. The read address inputs and the write address inputs of each side are separate in order to save the cost and time delay of external multiplexing between a read address and a write address.



4. The dual-access RAM serves as an external register file for the arithmetic processor chip. The Am29334 holds 64 words, each 18 bits long. Two chips are often connected to build a RAM block with four data outputs, two data inputs, and six address lines. Each port of the RAM can be independently accessed to read or write.

The word width of 18 bits allows the RAM to store two bytes plus a parity bit for each. Each side has separate write enable for the lower and upper nine-bit bytes and a common write enable that also switches the address multiplexer. The actual write is delayed internally to allow the write address to set up internally before writing starts.

It is possible to build a RAM with four data outputs, two data inputs and six addresses by using two dual-access RAMs and on each side connecting the data input, write address and write enables of one RAM in parallel with the corresponding inputs of the other RAM. This expanded RAM may be used in concurrent processing applications in which an ALU and an adder (which generates the address) do their computations—this yields a result and an address in parallel. The two values can then be fed simultaneously to the multiport memory.

The sequencer controls the show

The cycle time of the microprogrammed system is dependent on both the control path (i.e., sequencer and microprogram memory) and the data path (i.e., register file and ALU). Traditionally, the system bottleneck has been the control path, especially the ciritical paths associated with conditional branching. Special care has been taken in the design of the Am29300 family to balance control and data-path timing.

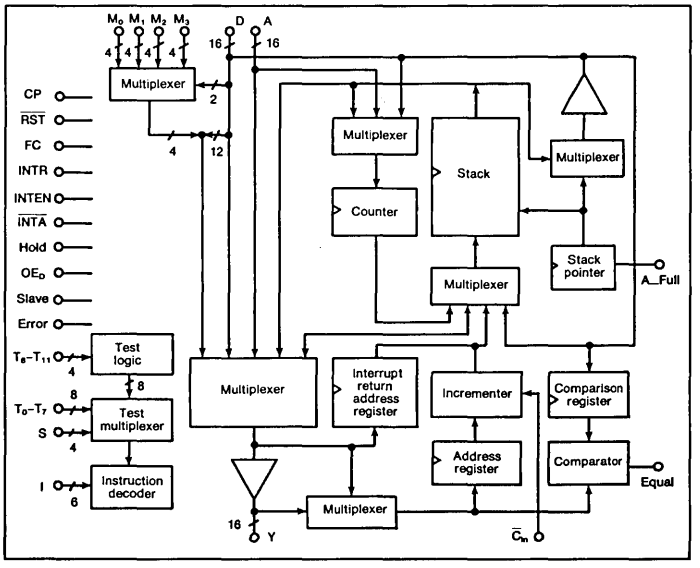
A key device contributing to the improved control-path timing is the Am29331 16-bit microprogram sequencer. It is designed for high speed, and that speed has been attained by the elimination of functions that would slow down the microaddress selection and by including the test logic and the test multiplexer in the sequencer (Fig. 5). As in most previous generation sequencers, the address register, the incrementer, the address multiplexer, the stack, and the counter are standard functions. The sequencer has multiway branch instructions that allow 1 of 16 consecutive addresses to be selected as the branch target in a single cycle.

The address register in most other sequencers is called a program counter, but this name is not correct if a strict definition is applied. In

the Am29331, the incrementing counter is placed after the address register, which thus allows for the handling of traps. The stack stores return addresses, loop addresses and loop counts. It has 33 levels to permit the deep nesting of subroutines, loops and interrupts. An output, Almost Full (A-Full), indicates when 28 or more of the levels are in use.

Available for use in iterative loops, the counter can be loaded with an iteration count at the beginning of a loop, and the count is tested and then decremented at the end of the loop. The loop is terminated if the count is equal to one; otherwise a jump to the beginning of the loop is executed.

There are three buses that carry microaddresses. The bidirectional D bus can be connected to the pipeline register, providing branch addresses or loop counts, or used for two-way communication with the data processing part of the system. The A bus, called an alternate bus, can be connected to a mapping PROM to provide starting microaddresses for instructions in a computer. The Y bus sends out



5. To aid in handling trap operations, the incrementer is placed after the address register in the Am29331 microsequencer. Additionally, the chip has a 16-bit address bus, which enables it to access up to 64 kwords of control memory and handle interrupts and multiple-path branches.

selected microaddresses to the microprogram memory and accepts interrupt or trap addresses if interrupt or trap is employed.

Four sets of 4-bit multiway inputs provide a simultaneous test capability of up to 4 bits. And, one way to use those inputs would be to decode mode bits in changing positions in macroinstructions. The four select lines select 1 of 16 tests to be used in conditional instructions. There are twelve test inputs. Four of these may be used for C (Carry), N (Negative), V (Overflow) and Z (Zero), generating internally the tests C+Z, C + Z, N XOR V, and N XOR V+Z, which are used for comparison of signed and unsigned numbers.

Relative addressing was the only somewhat useful function that was removed in order to maximize speed. The sequencer supports interrupts and traps with single-level pipelining, but may also be used with two levels of pipelining in the control path. It has a 16-bit-wide address path and cannot be cascaded, which thus limits the addressable memory depth to 64 kwords of microcode. That, however, is sufficient for the vast majority of applications—a typical computer, for instance, that has a microprogrammed instruction set, might use only about 1 to 2 kwords. However, for systems in which the microprogram is the sole program level, its size is generally larger.

Microprogram interrupts supported

The Am29331 sequencer supports interrupts at the microprogram level. Like polling, interrupts handle asynchronous events. However, polling requires explicit tests in the microprogram for events, thus leading to long response times, lower throughput, and larger microprograms. Interrupts, on the other hand, have a response time equal to the cycle time of the system (approximately 80 ns), measured from the Interrupt Request input (INTR). The sequencer accepts interrupts at every microinstruction boundary when the Interrupt Enable input (INTEN) is asserted.

An actual interrupt turns off the Y bus driver and asserts the Interrupt Acknowledge output (INTA), which should be used to enable an external interrupt address onto the Y bus, thus driving the microprogram memory. The interrupt also causes the interrupt return address to be saved on the stack; this permits nested interrupts to be handled (Fig. 6).

The Am29331 is also the first sequencer that can handle traps. A trap is an unexpected situation caused by the current microinstruction, which must be handled before the microinstruction completes and changes the state of the system. An attempt to read a word from memory across a word boundary in a single cycle is an example of such a situation. When a trap occurs, the current microinstruction must be aborted and re-executed after the execution of a trap routine, which will take corrective measures.

Execution of a trap requires that the sequencer ignore the current microinstruction and push the trap return address—the address of the ignored microinstruction—on the stack. The trap address must be transferred onto the Y bus at the same time. All this can be accomplished by disabling the carry-in to the incrementer ($\overline{C_{in}}$) and asserting the Force Continue input (FC) and the Interrupt Request input (INTR).

Also built into the sequencer is an address comparator, which allows detection of breakpoint in the microprogram. An output signal from the comparator indicates when the content of the comparator register is equal to the address on the Y bus. There is an instruction that loads the comparator register from the D bus and enables the comparator, which may later be disabled by another instruction.

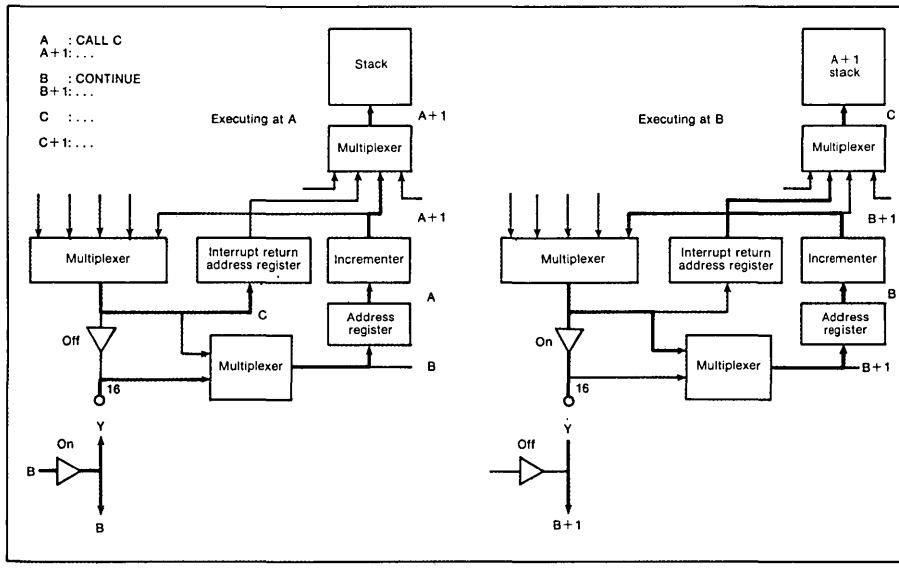
Parallel microprocesses are useful when the system must deal with peripheral devices that are controlled at the microcode level. Normally only one processor is present and it must be time multiplexed between the concurrent operations that must be performed. When a process is suspended its private state must be saved, so that it can be restored when the process resumes execution. That, in turn, requires that the state of the sequencer be saved and restored, or each process must have its own sequencer that is active when the associated process is active. The first approach is the least expensive, but the second offers the advantage of shorter response time, because no time is spent on saving and restoring the state.

The Am29331 supports the first approach with its bidirectional D bus, through which the

Microprogrammable 32-bit chips

entire state, with the exception of the comparator register, can be saved and restored. The sequencer also supports the multiple sequencer arrangement, in which the three-state Y buses from the sequencers are tied together driving a single microprogram memory. One of the sequencers is active, while the remaining sequencers are put on hold by asserting their Hold inputs. The Hold input disables most outputs (the D bus synchronously), disables the incrementer, and enables an internal Force Continue. This effectively detaches the sequencer from the system and preserves its state.

The sequencer has a 6-bit instruction input that is internally decoded to yield a set of 64 instructions. There are 16 basic branch instructions, each in an unconditional version, a conditional version, and a conditional version with complemented test. In addition there are 16 special instructions like Continue and Push C (push counter on stack). The branching instructions handle jumps, subroutines, various kinds of loops and exits out of loops, and FC actually overrides the instruction inputs with a continue



6. Because it can accept interrupts at any microinstruction boundary, the sequencer responds faster than most other microprogrammed systems. For example, while the instruction at point A in memory is being executed, the sequencer is directed to point B. The only restriction on the programmer is that the first instruction of the interrupt routine cannot use the stack, since the interrupt return address is pushed onto it at the start of the procedure.

instruction. FC is useful in field sharing and support for writable microprogram memory.

The Am29331 is one of the few sequencers where the stack is accessible from outside through the bidirectional D bus. This indirectly allows access to the whole state of the sequencer except the comparator register. This is useful when testing the device, and during system debugging, in which, for example, the contents of the counter and the stack may be examined and altered. By including the troubleshooting instructions in the microcode, the sequencer may aid in debugging itself and the rest of the system. The access to the state is also useful for changing context or extending the stack outside.□

Single-chip accelerators speed floating-point and binary computations

Complex multiplication or floating-point mathematical operations are frequently needed in most computer systems, but in many cases, not often enough to warrant the added cost of dedicating CPU hardware to the computational job. To speed up the calculations, many systems, though, allow for accelerator boards or boxes that can perform such operations at several megahertz speeds or more.

Already, many silicon designers have developed chips to simplify the design of such subsystems—16-bit parallel multipliers fabricated in bipolar, CMOS or NMOS processes, and single-chip or multichip floating-point processors made with CMOS or NMOS have been available for some time. However, they are low-performance solutions to the problem, or in some cases, have limited application since they are intended for highly pipelined systems.

Now, the ability to handle 32-bit binary multiplication or 32-bit floating-point multiplication, addition or subtraction can be added to a system with just a single chip. The Am29323 is a 32-bit parallel multiplier that accepts two 32-bit inputs and can deliver a 64-bit product in a single clock cycle of 80 ns. Alternatively, performing floating-point operations, the Am29325 accepts two 32-bit inputs and delivers a 32-bit result in less than 125 ns. It can operate with numbers represented in either the IEEE (P754) or Digital Equipment Corp. floating-point formats and can convert numbers from one format into the other.

Both chips are part of the just unveiled Am29300 series of 32-bit computational elements (Design Entry, p. 230). The multiplier is ideal for computer systems that do floating-point operations only infrequently but must often perform high-speed integer calculations such as those required in image manipulation. The floating-point processor enhances systems used for fast Fourier transform and scientific calculations. Systems could even contain both accelerators if a high-performance, general-

David Quong and **Robert Perlman**
Advanced Micro Devices Inc.

David Quong is a product planning engineer with the digital signal processing and array processing group at Advanced Micro Devices in Sunnyvale, Calif. He received a BSEE from California State University in Sacramento.

Robert Perlman is a senior product planning engineer with the digital signal processing and array processing group. He obtained a BSEE from the Rensselaer Polytechnic Institute and an MSEE from the Johns Hopkins University, and has previously done design work in airborne digital signal processing at Westinghouse.

purpose system were built (Fig. 1).

To speed the flow of data into and out of the chips, both circuits were designed with two 32-bit-wide input ports and one 32-bit output port. But the similarities end there, since the chips perform vastly different operations on the data. A fairly straightforward design, the multiplier uses a full Booth-encoded array to deliver a 64-bit product to the output register (Fig. 2). The output register feeds a multiplexer that sends the result, 32 bits at a time, to the output port.

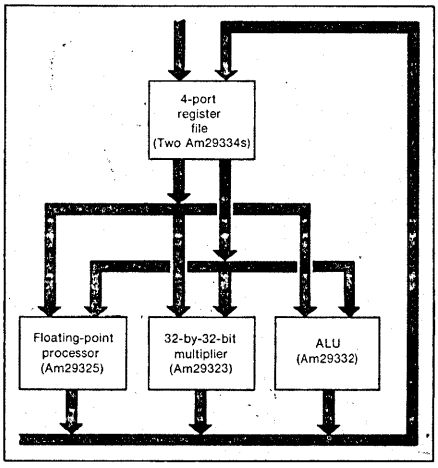
Double-precision operations can be done thanks to dual 32-bit input registers that are multiplexed into the multiplier array. A 67-bit partial-product adder allows new products to be summed with the contents of the output register. During this operation, the contents of the output register may be scaled by 32 bits, if necessary. Four partial products are formed and summed, and a temporary register assists in the scheduling of output transfers. The effective pipelining throughput in the double-precision mode is one 64-bit multiplication every four cycles. The accumulator can also support 96- and 128-bit multiplications. However, for such operations, input data must be repeatedly applied.

The input and output registers of the multiplier have independent control signals so that they can be optimally timed in pipelined systems. However, in unpipelined systems, the registers can independently be made "transparent" so that data encounters no delays when entering or leaving the chip. Like the other chips in the Am29300 family, the multiplier has parity checking and generating circuits to ensure system data integrity. And, the circuit offers a slave mode in addition to its normal mode—if two chips are tied together to operate in parallel with one set to operate in the slave mode, the circuits will generate an error flag if unequal results are obtained.

In the world of floating-point computations, several single-chip units, designed to be general-purpose math coprocessors for microprocessor systems have achieved close to microsecond operating speeds. However, to achieve higher throughput rates, several recently announced two-chip sets have cut that speed by a factor of 10, achieving data throughput rates of 10 MHz for pipelined operations. But, if operated in nonpipelined systems, these chips lose considerable speed—often by a factor or two or three—since data must ripple through the stages of pipeline registers.

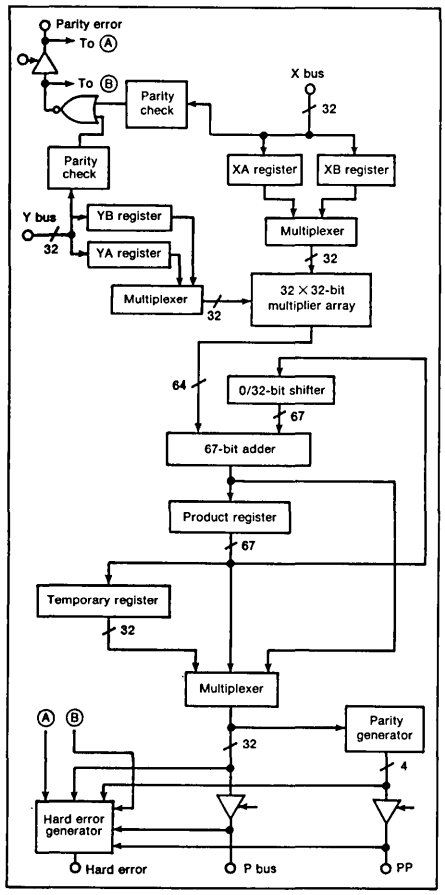
To cut the data delays, the Am29325 took a direct approach and eliminated all the pipelining. It is the first floating-point processor to contain a 32-bit floating-point adder/subtractor, multiplier, and flexible 32-bit wide data path on a single chip (Fig. 3). Additionally, support for division operations is included on the chip as well as a status flag generator.

Fabricated with the IMOX-S bipolar process and three levels of metal interconnections and



1. The 32-bit multiplier and the 32-bit floating-point processor can be used together in a system. Either chip also functions without the other if just one of the capabilities is needed.

housed in a 144-lead pin-grid-array package, the Am29325 can replace one to two boards of SSI and MSI logic typically used in general-purpose computers, array processors and graphics engines, to provide high-speed floating-point math capability. When used in concert, the on-chip functions will meet the computational and data-routing needs of these and many other applications.



2. Surrounding the 32-by-32-bit multiplier array on the Am29323 are multipliers for the two 32-bit input buses, which permit 64-bit multiplications to be done in just four cycles. The multiplier checks parity on the input data and generates parity bits for the output result.

Integrating these functions into a single device greatly reduces data routing problems and minimizes processing overhead that would otherwise be incurred when shuffling data on and off the chip. The internal data path is ideally suited for multiplication and accumulation, Newton-Raphson division, polynomial evaluation, and other often-used arithmetic sequences. Placing the data path on chip also dramatically reduces the number of ICs needed to interface the device to the rest of the system.

The three-port floating-point arithmetic unit at the chip's core can perform any of eight instructions in a single clock cycle. The absence of pipeline delay in the arithmetic unit means that the result of an operation is available for use as an input operand in the very next operation, a crucial feature when performing algorithms with tight feedback loops. Instructions and other operating modes are selected with dedicated input signals, an approach ideally suited to microprogrammed environments. The device easily interfaces with a variety of 16- and 32-bit systems using one of three programmable bus modes.

Delving into the operation

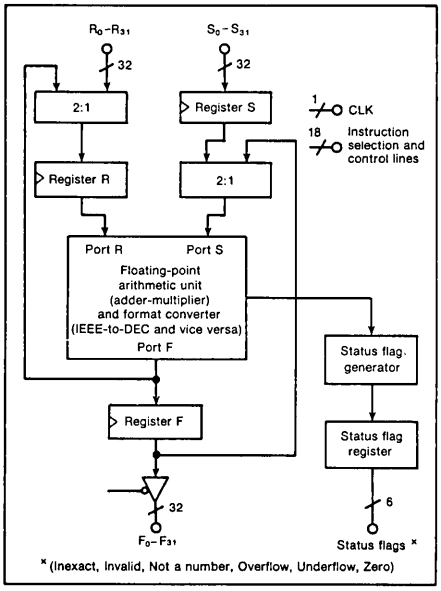
At the heart of the arithmetic unit are a high-speed adder-subtracter, a 24-by-24-bit multiplier, an exponent processor, and other logic needed to implement the floating-point operations. Two input ports, R and S, provide operands for the instruction to be performed; the result appears on port F. One of eight instructions is selected by placing a 3-bit code on lines I_0, I_1, and I_2. The first three instructions—R + S, R − S, and R × S—operate on both input operands; the remaining instructions need only one input operand.

The fourth instruction, 2 − S, forms the core of the Newton-Raphson division algorithm, in which the quotient A/B is calculated by first evaluating 1/B, then postmultiplying by A. The reciprocal value 1/B is derived by using an external lookup table to provide an approximation of 1/B; this approximation is refined using the iterative equation:

$$x_n = x_{n-1}(2 - Bx_{n-1}),$$

where x_n is the nth approximation of 1/B.

Once B and the approximation of 1/B are loaded into the Am29325, the approximation is refined using a sequence of R × S and 2 − S instructions; no additional I/O operations are needed for reciprocal refinement. The remaining four instructions perform data format conversions. Instruction INT to FP converts a 32-bit, two's complement integer to floating-point form, useful when processing data initially generated in fixed-point format; conversion from floating point to integer format is handled by instruction FP to INT. Two other instructions convert between IEEE and DEC floating-point formats.



3. Also using separate 32-bit buses for the inputs and output, the AM29325 floating-point processor handles either IEEE or DEC formatted data and can translate between formats, if necessary.

The arithmetic unit recognizes two single-precision floating-point formats—the IEEE format as specified in proposed standard P754, draft 10.0, or the DEC format used in VAX minicomputers. The eight instructions can be performed using either format; the desired format is selected with the IEEE/DEC pin on the processor chip. The formats are broadly similar—each has an 8-bit biased exponent, a 24-bit significand comprising a 23-bit mantissa appended to an implied or "hidden" most-significant bit (MSB), and a sign bit.

There are, however, a number of subtle differences. The IEEE format has an exponent bias of 127 and a binary point placed to the right of the hidden bit, while the DEC format has an exponent bias of 128 and a binary point placed to the left of the hidden bit—these variances result in a slightly different range of representable values. Each format has its own set of operands reserved for special uses. The IEEE format reserves operands to represent non-numerical values (referred to as Not a Number, or NaN), $+\infty$, $-\infty$, and plus and minus 0; the DEC format reserves only two types of operands to represent non-numerical values and 0. In addition to format differences, there are a number of minor differences in the manner in which operands are handled during the course of a calculation. These differences are automatically accounted for when the desired format is selected.

The need for rounding

When performing a floating-point operation, it is sometimes possible to generate a result whose value cannot be precisely expressed as a floating-point number. If, for example, the single-precision floating-point values 2^{23} and 2^{-1} are added, the infinitely precise result, $2^{23} + 2^{-1}$, cannot be represented exactly in the single-precision floating-point format. Some means, then, must be provided for mapping the infinitely precise result of a calculation to a representable floating point value. The arithmetic unit implements four IEEE-mandated

rounding modes to afford the user some flexibility when performing this mapping; the desired rounding mode is selected with signals RND_0-RND_1.

Of the four modes, the round-to-even mode is most often used; it maps the infinitely precise result of an operation to the closest representable floating-point value. The round-toward $-\infty$ mode maps to the nearest representable value less than or equal to the infinitely precise result; similarly, the round to $+\infty$ mode maps to the nearest value greater than or equal to the infinitely precise result. A fourth mode, Round toward zero, maps to the closest representation whose magnitude is less than or equal to that of the infinitely precise result. As one would expect, if the infinitely precise result of an operation is representable in the floating-point format, it passes through the rounding operating unchanged, regardless of rounding mode.

As the result of an operation, various status flags are set or reset by the status flag generator. Six flags are used to note the occurrence of overflow, underflow, zero, not-a-number, invalid, or inexact conditions. Because the flags are generated as the operation is performed, the user can greatly reduce processing overhead that would otherwise be needed to test the results of operations. The flags are fully decoded, minimizing the amount of hardware needed to interpret them.

Flagging the status

Four of the status flags report exception conditions stipulated in IEEE standard P754. The Invalid flag indicates that an input operand or operands are invalid for the operation to be performed. The Underflow and Overflow flags are active when a result is too small or too large for the operation's destination format. The fourth exception flag, Inexact, tells the user that the result of an operation is not infinitely precise. Although these flags are primarily an adjunct to operation in the IEEE format, they also produce valid results when the DEC format is selected. The Am29325 generates two additional flags not provided for in the IEEE standard. Flags Zero and NaN identify zero-valued or nonnumerical results for both IEEE and DEC formats.

A floating-point processor whose arithmetic unit performs millions of operations per second can maintain that operating speed only if the correct operands can be routed to the arithmetic unit at that rate; if not, the specification is meaningless. To meet this crucial requirement, the core of the Am29325 is supported by a 32-bit data path comprising two input buses, a three-state output bus, and two data feedback paths. These data paths give the user the means to get the operands to where they are needed without devouring extra clock cycles.

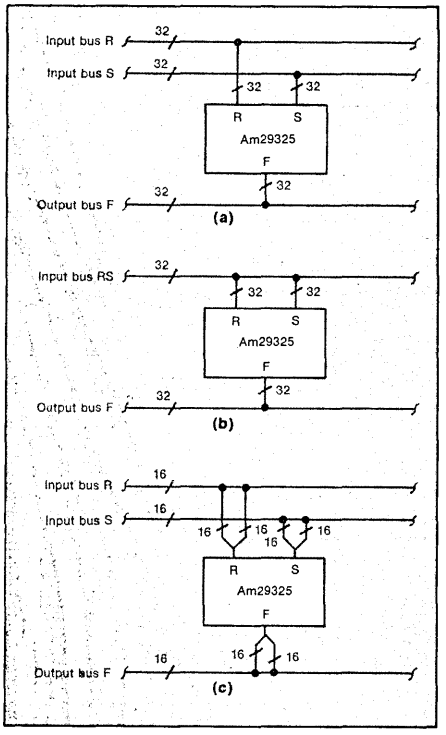
Data enters through input buses R_0-R_{31} and S_0-S_{31}; results exit through three-state output bus F_0-F_{31}. Each bus has a 32-bit edge-triggered register for data storage; data is stored on the rising edge of common clock input, CLK. An independent clock enable is provided for each register, so that new data can be clocked in or old data held; the clock enables are well-suited to a microprogrammed environment, and make the gating of clocks, always a risky business, unnecessary. The ability to clock or hold any register is a powerful tool for performing algorithms with conditional operations, or algorithms in which intermediate results must be delayed for one or more cycles before reentering the calculation.

In many applications,the internal registers will be used to store input and output operands; it is in this register-to-register mode that the chip shows its top speed. Some users, however, may wish to bypass one or more of the internal registers. The input and output registers can be made transparent independently using feedthrough controls FT0 and FT1. If all three registers are made transparent the device operates in a purely combinatorial "flow-through" mode. That mode, through, is somewhat slower than the register-to-register mode, but is useful in systems that need a register structure substantially different from that provided in the Am29325, or in systems where floating point operations must be concatenated with other combinatorial functions.

The two feedback data paths greatly simplify the task of moving data from one calculation to the next. One path routes data from the output of the arithmetic unit to a multiplexer at the input of register R; the multiplexer selects the operation result or R_0-R_{31}. The result of any operation can therefore be loaded into register

32-bit math accelerators

R, register F, or both. The second path feeds the output of register F to a multiplexer at the arithmetic unit's S port; the multiplexer selects either register S or register F as the port S input. This path effectively increases the number of commands—instruction R Plus S, for example, can also be performed as R Plus F.



4. Three programmable I/O bus modes permit the floating-point processor to operate with dual 32-bit input buses (a), a single, shared 32-bit input bus (b), or even two 16-bit buses (c) so that it can easily connect to most 16-bit microprocessor systems.

Thanks to the inclusion of three programmable I/O modes, the circuit readily interfaces with both 16- and 32-bit sytems. The most straightforward of these options is the 32-bit, two-input bus mode (Fig. 4a). The advantage of this mode is its high I/O bandwidth—no multiplexing of I/O buses is required, thus improving system speed and easing critical timing constraints. R and S operands are taken from their respective buses and clocked into the R and S registers on the rising edge of CLK; register F is also clocked on this transition.

Another choice sets up a 32-bit, single-input bus, in which both the R and S buses are connected to a single input bus (Fig. 4b). The R and S operands are multiplexed onto this bus by the host system; the R register clocks its operand on the rising edge of CLK, the S register on the falling edge. The S operand is double-buffered on chip, so that the new S operand is presented to the arithmetic unit on the rising edge of CLK. Operation of register F and the F bus is the same as in the 32-bit, two-input bus mode.

The last option has targeted 16-bit systems—a 16-bit, two-input bus mode (Fig. 4c). In this mode the R, S, and F buses are 16 bits wide; 32-bit operands are placed on the buses by time-multiplexing the 16 MSBs and LSBs of each data word. The LSBs of the R and S operands are double-buffered on chip, so that the complete 32-bit operands are presented to the arithmetic unit on the rising edge of CLK. Internal data paths and registers remain 32 bits wide, thus giving the 16-bit system designer the benefits of the simple interface and the speed of the wide internal data paths.

Putting the part through its paces

Multiplication and accumulation—a combination of operations very commonly used in digital filtering, image processing, matrix manipulation, and many other applications—can readily show the capability of the floating-point processor. In such a combination of operations, N input terms x_i are multiplied by constants k_i; the products are then added, producing the weighted sum:

$$s = \sum_{i=0}^{N-1} k_i \, x_i$$

To do this with the Am29325 is a simple two-step process, with two additional steps for ini-

tialization. In the first step data and coefficient values x_0 and k_0 are clocked into registers R and S. During step two the values x_0 and k_0 are multiplied and the product placed in register F; at the same time, data and coefficient values x_1 and k_1 are clocked into R and S. Third, values x_1 and k_1 are multiplied and the product placed in R. In step four, products x_1k_1 and x_0k_0 are added and the sum placed in F, and x_2 and k_2 are clocked into R and S.

The third and fourth steps are then repeated for as many iterations as needed to complete the operation. Once the part has been loaded with the first two sets of operands, the internal data path routes partial results to keep the arithmetic unit busy with a multiplication or addition every clock cycle; a new multiplication and accumulation is performed every two clock cycles. The partial results remain on-chip until the multiplication and accumulation is completed, thus eliminating I/O delays and the more complex programming that would result from having the adder and multiplier on separate chips.

Some real applications

A more specific application for the Am29325 could be its use as the computational engine in a fast Fourier transform (FFT) processor. During a FFT operation, word growth is incurred in the butterfly calculation, and if the FFT processor uses integer arithmetic, word growth can cause a system overflow. To prevent overflow, a scaling operation must be performed on the data. The overhead involved in checking for word growth overflow and scaling of data can be avoided by using floating-point arithmetic. Floating-point provides not only greater dynamic range but in most cases also provides greater precision (24 bits of significance versus 16 bits in a typical integer system).

A powerful, low-cost system that executes FFTs can be built around the floating-point processor (Fig. 5). It consists of a floating-point arithmetic processing unit, a data and coefficient address generator, a data and address storage block, high-speed data and coefficient memories, a system controller, clock generator, and host interface. Input operands to the R port are fed from the data store, while data to the S port is fed from the coefficient memory. The result of an arithmetic operation may be stored back in the data memory. An exclusive-OR gate is also available to complement the sign of the result, effectively multiplying the operand on the F bus by -1. For most operations, intermediate results can be held within temporary registers in the floating-point unit; only the final result need be sent off chip.

The high-speed data memory is made up of RAMs, the coefficient memory of PROMs. The data memory can be loaded with data from the host or can store results that have been processed through the floating-point chip. Once all data or results have been stored, the data memory is ready for use in an operation, or for transfer back to the host system. The coefficient PROMs contains the sine and cosine data required for an FFT, while the data store holds frequently used operands.

During the calculation of a butterfly, the same operands must be used in several different cycles—and since the data store reduces the number of memory read operations required, it speeds up data access. As the butterfly sequence progresses, the appropriate address is available from the address store, which consists of two more multilevel pipelined registers.

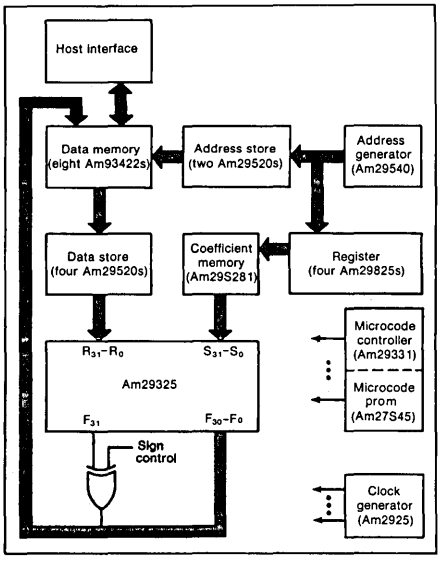
The host interface consists of a DMA channel that can perform high-speed block data transfers between the host system and the data memory. The system controller communicates with the host to receive or transfer data. It governs which operations are to be performed and how to perform them. Instructions are issued by the host computer, via the host interface, to the system controller, and the system controller informs the host when the operation is done.

The system controller consists of an Am29331 or similar microsequencer, and a microcode program stored in registered PROMs. The system clock generator uses an Am2925. The architecture allows a ten-cycle butterfly FFT to be executed (see Fig. 5 again) using a radix-2 decimation-in-time (DIT) algorithm. The equations for a radix-2 DIT algorithm are:

$$A' = A + BW\,B'$$
$$B' = A - BW, \text{where all values are complex}$$

In cycles 1, 2, and 3, the first three operands are read from the data memory. Because of the

overlapping butterflies, this read takes place while the previous butterfly is still being processed. In the following two cycles, data writes of the previous butterfly occur while the complex multiplications of (BW) are being performed. Cycle 6 reads in a new operand for the present butterfly and sums together the two products from the two previous cycles. In cycles 7 and 8, the real part of A′ and B′ is formed. In cycles 9 and 10, the real part of A′ and B′ is written to memory. Also, during these two cycles the other product pairs of (BW) are formed.



5. To build a fast-Fourier transform processor that uses the floating-point processor as its heart requires only a few control chips and some memories. Use of the Am29540 and Am29332 LSI building blocks helps keep the circuitry simple.

During cycles 11, 12, and 13, data for the next butterfly is read, and as part of cycles 12 and 13, the imaginary part of A′ and B′ is formed. In the following cycles the imaginary part of A′ and B′ is written to memory and processing of the next butterfly is initiated. The real and imaginary components of B′ have a negative sign, and can be corrected by complementing the sign. Counting the number of cycles from the first read or write of one butterfly to the next, it can be seen that a butterfly is computed every 10 cycles.

The big system picture

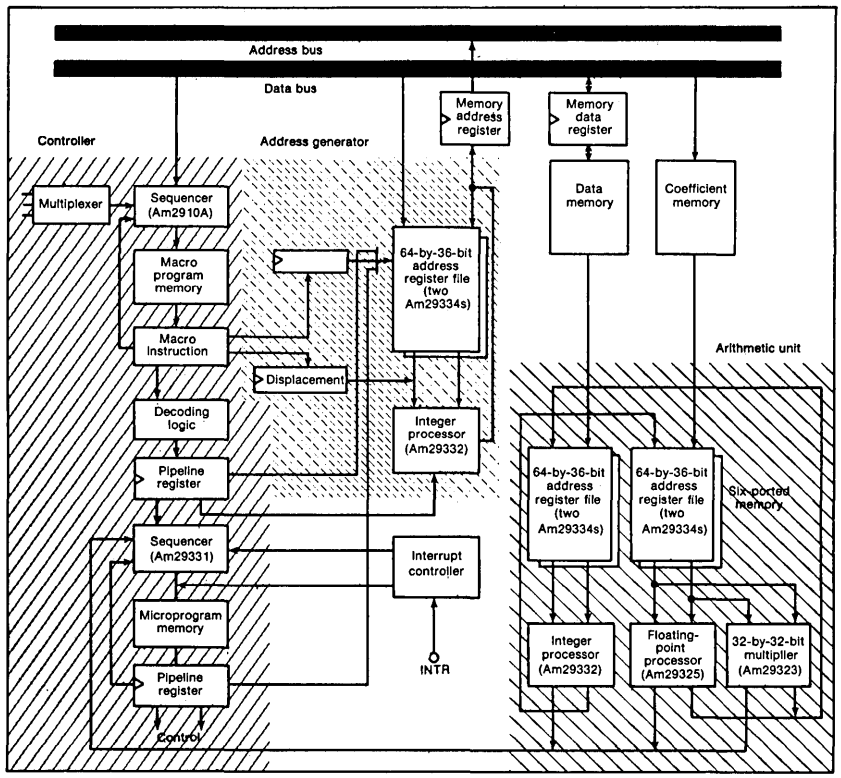
Although the floating-point chip fits well in small systems, it is also easily incorporated in larger, more powerful configurations. In one such system, a high-speed, microprogrammed integer and floating-point processor can be readily tailored to implement signal processing, image processing, or graphics algorithms (Fig. 6). The processor consists of a two-level controller, data and coefficient memory, address generator, and arithmetic unit. These functional blocks are considerably more flexible than their counterparts in the simpler FFT system.

The controller is divided into two levels, or sections: program and microprogram. In the topmost or program section, an Am2910A microprogram controller addresses a program memory that contains high-level instructions, or macros. These macros implement building-block operations; a graphics processor, for example, might have macros called Translate and Rotate that move objects in three-dimensional space. Each macro would carry with it parameters relevant to its operation, such as memory pointers or iteration count.

The program section passes address-related parameters to the address generator, and passes the iteration count and the decoded microinstruction start address to the microprogram section of the controller; this section then provides cycle-by-cycle control of processor resources during the execution of a

macro. The heart of the microprogam section is an Am29331 microprogram controller—it addresses a microcode memory, in which the microprogram sequence for each macro type is stored.

The microprogram controller was chosen for three reasons: first, it can address up to 64 kwords, which makes possible a deep microprogram memory that can store many operation sequences. Second, its high speed permits the use of slower, less expensive microprogram memory, a particularly important considera-



6. A versatile, yet high-performance microprogrammable system can be built by including both the floating-point processor and the 32-bit multiplier into a system that uses the other Am29300 building blocks to form the control and address generation sections.

tion when the microprogram is large. And third, its micro-interrupt feature can be used to efficiently implement exception handling for arithmetic operations. By using interrupts for these exceptions, the overhead otherwise incurred in testing status flags can be greatly reduced.

The data and coefficient memories store input data, output data, and constants. In this application, data and coefficient memory have been separated from program memory. Sometimes referred to as a Harvard architecture, this approach increases throughput by allowing instruction fetch and operand fetch operations to proceed in parallel.

The address generator comprises a Am29332 ALU and two Am29334 register files. The register file stores up to sixty-four 32-bit base addresses and pointers. The Am29332 creates a 32-bit effective address from these bases and pointers, with the calculation assuming the forms:

base + pointer
base − pointer
base
or pointer

In addition, the Am29332 can perform mask, shift, and merge operations in a single cycle. This feature can be used to quickly calculate matrix addresses of the form:

$$a2^N + b,$$

where a and b are the row and column indices of the matrix element to be accessed. The combination of a 32-bit effective address and efficient matrix addressing makes this address generator particularly attractive for applications such as image processing, in which matrices must be plucked out of very large data arrays.

The arithmetic unit contains three arithmetic facilities—an Am29325 for floating-point operations, and the Am29332 and Am29323 for integer and logical operations. These devices accept data from a six-port register file made of four Am29334s. The register file has three purposes—it acts as a fast, temporary scratchpad for data, it routes data among arithmetic devices (the output of one arithmetic device can be written to the register file, and be used as an input operand by another such device during the following clock cycle), and it provides access to four data words every clock cycle, so that two or more arithmetic device can operate in parallel.

An example of this parallelism is integer multiplication-accumulation: because the Am29323 and the Am29332 receive operands independently, an integer product and sum can be calculated every clock cycle. The register file can then pass products from the Am29323 to the Am29332, for a throughput of one clock cycle per multiplication-accumulation.

Operation of the processor might be best understood by considering the execution of a typical macro. For graphics applications, one such macro is Translate, with which a set of points in three-dimensional space is moved in a given direction. The set of points is described by a list of vectors (X_i, Y_i, Z_i), while the translation is described by vector (S_T, Y_T, Z_T); each vector is stored in three contiguous data memory locations. Translation is performed by adding the translation vector to each entry in the vector list.

The translation process begins when the microprogram controllers encounters a Translate instruction in program memory. The Translate instruction is accompanied by three parameters: the start address of the translation vector, the start address of the vector list, and the number of vectors in the list. The first two parameters are passed to the address generator, the third to the iteration counter.

The microprogram section of the controller then assumes command, accessing the microcode for the Translate instruction. The microcode controls the address generator and arithmetic unit, specifying the operations needed to fetch each vector from the vector list, add the translation vector, and return the modified vector to the data memory. After all vectors in the list have been processed (as indicated by the iteration counter), control is returned to the Am2910A program sequencer, which then accesses the next macro from program memory.□

BUILDING BLOCKS STACK UP TO HIGH PERFORMANCE

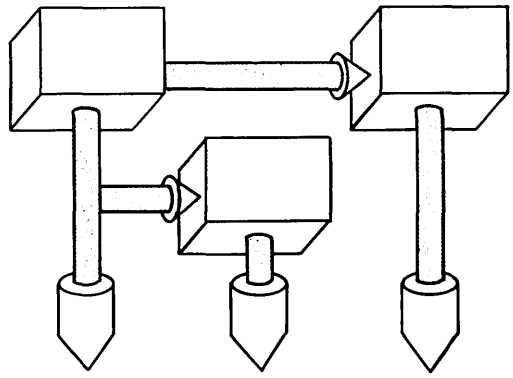
Designed using concepts of functional partitioning, three-bus architecture, and fault detection, a family of 32-bit building blocks can satisfy the needs of both general-purpose computing and signal processing.

by Timothy J. Flaherty

As processing rates, machine densities, and system reliability requirements increase, functional integration at the device level becomes mandatory in high performance controllers and processors. When used as standalone devices or combined in a high speed system, functional building blocks can provide solutions to a wide range of design problems. They fit equally well into a general-purpose computer and a digital signal processor, despite the great functional differences between these two systems.

As device densities have increased over the years, system word widths have grown, bringing greater precision and allowing a larger memory space to be addressed. The jumps from 4- to 8-bit and from 8- to 16-bit systems occurred relatively quickly. The leap to 32-bit systems has already taken place, bringing with it a slowdown in the quest for wider system words. Partitioned into 32-bit building blocks, Advanced Micro Devices' Am29300 family integrates functions that are difficult, if not impossible to implement with bit-slice devices. These functions include barrel shifting, priority encoding, and mask generation.

Whenever carry-lookahead logic can be contained in the same device as the arithmetic logic it supports, cycle time is improved. In fact, by reducing the amount of intrafunction communication across chip boundaries, cycle time no longer has to depend on the speed of the interface between components. Because of this, all intrafunction communications were eliminated in the Am29300 family. Pipelining can result in the faster execution of certain highly repetitive operations, but system latency increases. In some cases, this latency will actually degrade throughput. In a recursive algorithm, where a calculation depends on the immediately preceding result, true throughput can be lost while waiting for intermediate results to work their way through the pipe. To maximize performance without sacrificing architectural flexibility, intrafunction pipelining was also eliminated in the Am29300 devices.



A three-bus, flow-through architecture complements functional partitioning in this chip family.

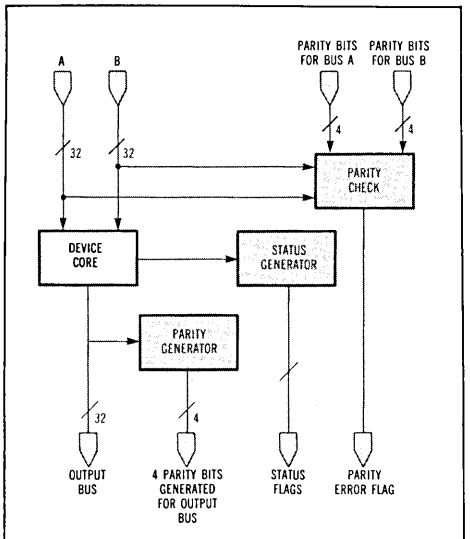
Timothy J. Flaherty is a product planning engineer at Advanced Micro Devices, Inc (Sunnyvale, Calif). He holds a BS in electrical engineering from the University of Santa Clara.

The data path members share a common bus configuration with two input operand buses and one output bus. Independent of each other (neither bidirectional nor shared), these buses provide maximum accessibility.

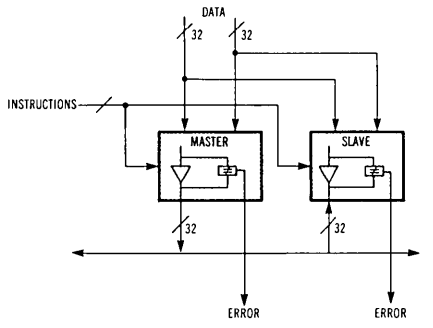
Bidirectional or shared I/O buses limit the speed at which information can be transferred between different parts in the system. Achieving rapid turnaround of bidirectional TTL data buses is often an arduous task. And shared input buses require greater timing restrictions than do nonshared buses. These limitations have been eliminated in the Am29300 data path devices by removal of shared or bidirectional buses.

A high data transfer bandwidth is achieved with the flow-through architecture. This direct access allows the designer to tailor the system's register file to the specific application rather than forcing use of a fixed, more general memory organization. The beauty of the three-bus architecture lies in its simplicity. This straightforward structure permits many possible component configurations optimized for different micro-architectures.

Simple, internal I/O registers may introduce unwanted pipeline delays. A flexible register structure requires that any I/O registers can be made transparent. The input and output registers on both the Am29323 parallel multiprecision multiplier and the Am29325 floating point processor can be made transparent independently, providing a number of different register configurations including flow-through.



The three-bus, flow-through architecture affords the greatest access to the device cores. Byte-parity checking detects connection failures between devices.



Master/slave checking provides device failure detection using redundant parallel devices. This happens without incurring the delay of a "voting" scheme often used in high reliability designs.

Fault detection

The philosophy governing this chip family is maximum functionality with minimum impact on system cycle time. The methods used for fault detection put this idea into practice.

The 32-bit family addresses fault detection at the component level using a twofold scheme—byte parity and master/slave checking. To detect interconnection failures, byte parity is both generated and checked by the data path elements of the family. The byte parity circuitry checks for single bit failures across each byte of the two input operands. Even parity checking was chosen for this family of TTL-compatible parts instead of odd parity to provide the additional check for bus failure. Any parity faults detected cause assertion of the parity error (PARERR) flag.

Master/slave checking detects failures at the device level. When using this mode, two devices are operated in parallel, each receiving the same data and instructions. The master device generates its result and transfers this information to the output bus. The slave device generates its own result from the same inputs; instead of delivering this data, however, the slave reads the output bus and compares the master's results with its own. The hard error (HARDERR) flag indicates any discrepancies between the two outputs. Moreover, the assertion level of both the parity and hard error flags indicates device failure due to loss of power and error signal faults.

Both error checking schemes operate on a cycle-by-cycle basis so any detected fault triggers an interrupt at the microinstruction level. Unlike other redundant schemes, specialized software is not required, and system performance is not affected by the communication between redundant functional units.

Cycle time and control paths

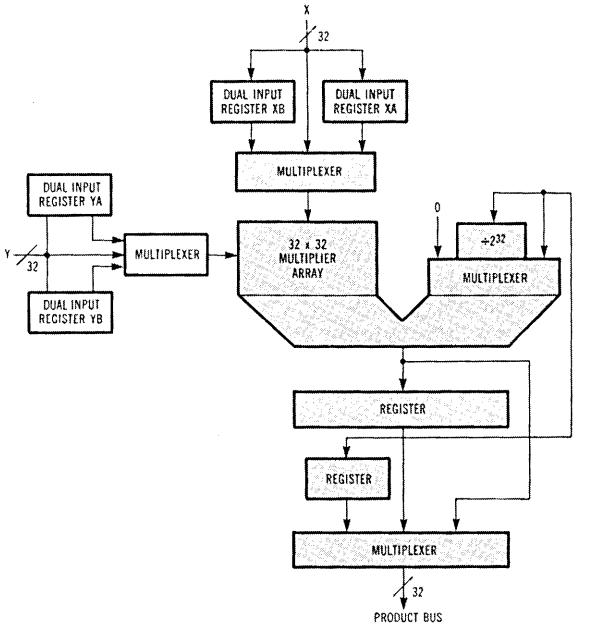
In high performance system design, the system's intended operations must be given, with careful consideration paid to required cycle time. The cycle time depends on the type of operation the system performs. The design should be optimized for quick execution of the instructions that make up the largest percentage of the system's operations.

Complex operations requiring long cycle times, but used infrequently, should be performed over multiple cycles. For example, a complicated arithmetic procedure such as division should not determine the cycle time of the system if the operation only used a small portion of the time. On the other hand, if this operation is used frequently, the system should be made to handle it efficiently.

Comparing the instruction mixes of a general-purpose processor and a dedicated array processor illustrates this point well. Multiplication operations dominate the array processor's instruction set, while the general-purpose machine's set would be less multiplication intensive. The Am29323 parallel multiplier would enhance an array processor by providing a high speed, single cycle 32- x 32-bit multiplication. The general-purpose machine might not need a dedicated multiplier and could fare well with the Am29332 ALU chip and its multiple cycle multiplication capability.

When optimizing the system for speed, a designer should remember the control path. By causing a change in the normal flow of information in the control path, conditional branching often becomes the system bottleneck. Conditional codes must be checked to determine the next address, but this checking can extend the cycle time. The speed of the control path must remain on a par with the speed of the data path. The Am29331 microprogram sequencer architecture balances the timing between the control and data paths.

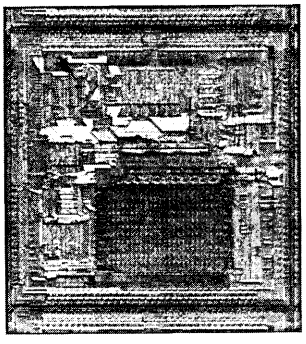
By integrating the conditional code multiplexer, test logic, and polarity control logic in the same device, cycle time is reduced by eliminating intrafunction delays. The microprogram sequencer can perform four sets of 16-way branches upon the simultaneous



The 32- x 32-bit parallel multiprecision multiplier provides dual input registers to support extended multiplications. An internal wrap-back path and shifter eliminate transferring data offchip for shifting.

A family gathering

A member of the Am29300 family, the Am29332 32-bit noncascadable ALU chip, was designed for systems requiring fast number crunching, high data transfer rates, and powerful bit-manipulation capabilities. The internal data path of the ALU chip interconnects several functional blocks. These blocks include a mask generator, a funnel shifter, an ALU, and a priority encoder. The ALU chip allows operations that once took multiple cycles to be executed in a single cycle.



The ALU chip uses a 64- to 32-bit funnel shifter to perform a full complement of N-bit shifts, N-bit rotates, field extractions, and field logical operations in a single cycle. This funnel shifter works on either one or both input operands. Such shifting is extremely useful in such operations as floating point mantissa normalization or denormalization, and in applications where packing and unpacking of data is a frequent task. Also, the ability to extract a 32-bit contiguous field from two operands provides a useful function in many graphics-related operations. The output of the funnel shifter is directed to the R input of the ALU, allowing logical operations to then be performed on the shifted word. The ALU section of the Am29332 has three input ports. One input comes from the funnel shifter, another from the mask generator, and a third can be selected from various sources including both input buses. This three-input ALU allows merger of two instructions into a single cycle.

The Am29325, a single-precision floating point processor, integrates a fully combinatorial 32-bit floating point adder/subtractor, multiplier, and data path in a single chip. This integration minimizes processing overhead. The floating point processor supports both IEEE P754 and Digital Equipment Corp floating point formats. All instruction—addition, subtraction, multiplication, floating point/integer conversions, and IEEE/DEC conversions—are performed in a single clock cycle. There are no internal pipeline delays to limit true throughput.

The core of the floating point processor is a 3-port arithmetic unit containing a mantissa processor, an exponent processor, and additional logic required to implement floating point operations.

The Am29323 is a 32- x 32-bit parallel multiplier with multiprecision capabilities designed to perform a 32- x 32-bit multiplication in a single cycle. The parallel multiplier also supports multiple cycle, multiprecision multiplications. Using a 67-bit onboard accumulator and internal wrapback paths, this device can perform a 64- x 64-bit multiplication every four cycles. This part also supports 96- x 96-bit and 128- x 128-bit multiplications. These expanded multiplications offer support to extended and double-precision format floating point multiplications.

To provide a flexible interface for a variety of applications, the parallel multiplier has dual 32-bit registers on each input bus. Both halves of a 64-bit input word can be loaded, stored, and selected as needed when extended multiplications are performed. The input registers can be made transparent and the outputs can be selected directly from the array core to provide a high speed multiplier accelerator in a system designed with the other members of the Am29300 family.

The task of reducing cycle time in a microprogrammed system—a primary goal for the Am29300 family—is assisted by the Am29331 program sequencer. The critical path in the control section of a system typically passes through the status register through test logic, test multiplexer, sequencer, and microprogram memory. The microprogram sequencer removes the "control bottleneck" by integrating the test logic, multiplexer, and sequencer.

Handling interrupts

Interrupts and polling both handle asynchronous events. But in interrupts, unlike in polling, explicit tests in the microcode are not required. Quicker response times and less microcode are the reasons the microprogram sequencer uses interrupt handling. When an interrupt is received, the interrupt return address is pushed on an internal 33-level stack allowing nested interrupts.

Interrupts are handled by the sequencer at the end of a microcycle. Traps, on the other hand, must be handled before the end of the microcycle. Because they indicate an unexpected condition caused by the current microinstruction, traps cause the sequencer to halt the operation before the current instruction changes the state of the system. When a trap occurs, the current microinstruction must be aborted and re-executed after the trap handling routine has taken corrective measures.

The Am29334, a high speed, 64- x 18-bit, dual-access RAM, provides the Am29300 family with flexible, configurable memory. The device's dual read/write ports allow simultaneous access for two operations every cycle: two operand fetches, a read and write to two locations, or two write operations.

Because it can be expanded in both width and depth, the register file allows several memory configurations. Two of these devices may be hooked together in an expanded 6-port configuration, for example. This setup allows two processors to operate on the same memory simultaneously. Four reads and two writes every cycle could provide high speed local memory, possibly configured as a cache.

occurrence of four external test conditions. This ability to handle multiway branching greatly reduces the branching delay penalty.

Data routing

A system should be able to route data punctually to the proper location—a task as important as reducing cycle time. Cycles wasted while waiting for results to work their way out of the pipeline and into the arithmetic unit where they are needed degrade performance. Bus bandwidth is lost by the redundant transferring of intermediate results back and forth from memory. And cycles lost shuffling data reduce true throughput.

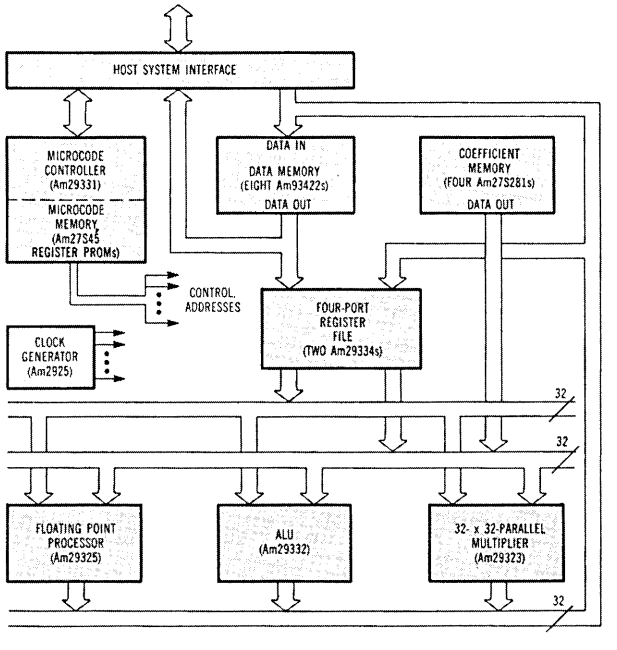
Often data is fetched from one memory location, processed, and the result of the operation returned to the original memory location. The Am29334 register file supports these read/modify/write operations by allowing a single cycle read and write memory operation to the same location. The register file's internal circuitry makes this operation possible without requiring external hardware to store the modified data temporarily.

Maximum bus bandwidth requires operands to be in the right place at the right time without monopolizing the bus structure. Redundant data transfers, such as returning intermediate sums from a sum-of-products operation to memory, only congest the bus structure and reduce bandwidth. The Am29325 floating point processor provides internal wrap-back paths and handles such data routing onchip. These internal wrap-back paths for sum-of-products operations with intermediate results double the bandwidth of a bus shared between multiple processors.

The Am29323 parallel multiplier also provides internal wrap-back paths and shifting circuitry for extended multiplications. These elements eliminate the delays resulting from data leaving the chip, being adjusted by an external shifter, and then returning to the device. The parallel multiplier has dual 32-bit input registers to support the cross-products needed for multiprecision multiplications. These registers also reduce bus congestion by eliminating the need for redundant memory fetches.

System bus structures

A general-purpose CPU falls short of today's number crunching requirements because it cannot take advantage of highly structured array and digital signal processing algorithms. The differences



A 32-bit floating point/ integer processor can be designed using a parallel configuration. In the system shown, the floating point processor shares three 32-bit buses with the ALU chip and the parallel multiplier.

between a general-purpose CPU design and an array processor design allow optimization of the configurations to serve specific needs.

Different system designs often have different data bus requirements. Parallel configurations are useful and easily implemented. In a general-purpose machine, for example, several processing units might share the same data buses. The Am29332 ALU chip can share three 32-bit buses with the parallel multiplier and the floating point processor. Data could be passed from one processor to another through the shared register file.

Although parallel configurations fit a general-purpose design, specific processors may require different bus structures. With a dedicated bus structure and paired Am29331 RAMs configured as a 6-port register file, for example, a high speed system that is well-suited for matrix processing can be designed. For array processing this arrangement offers a distinct advantage over the shared bus system.

To perform a multiplication/accumulation, the Am29323 multiplies two 32-bit numbers, the product is passed through the register file and added to the previous product by the ALU chip. Performing the multiplication and the addition in parallel, results in an effective throughput of one multiplication/accumulation per clock cycle—twice that of the system with shared buses.

Status generation

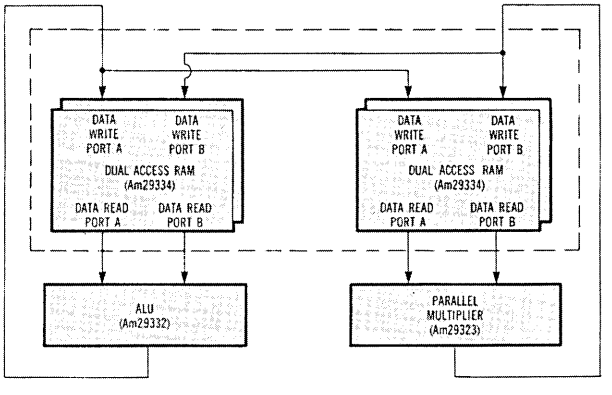
The status flag generators on Am29300 data path devices create flags that indicate where significant events occur during the calculations. These flags, generated as the operation is performed, reduce the processing overhead. The fully decoded flags minimize the amount of hardware needed for status interpretation. Such special conditions as a zero result or a byte carry may provide the user with important information about the calculation. A zero result—reported via the ZERO flag—is useful in comparison operations, for example.

Many of the status flags report such exception conditions as underflow, overflow, and invalid. Each of these conditions would indicate that the result obtained is not correct. These flags are active whether or not the output bus is enabled. In this way, the status of such iterative operations as floating point multiplication/accumulation can be monitored without enabling the output bus to check each intermediate calculation for exception conditions. This will reduce both hardware requirements and bus congestion.

Many of the conditions reported by the status flags indicate a problem with the current operation. The INVALID flag on the floating point processor, for example, indicates an invalid operation has been attempted. This flag can be used to generate an interrupt. The microprogram sequencer handles this interrupt at the microprogram level. After accepting the interrupt, the sequencer allows an external interrupt handling address to gain access to the microprogram address bus. This address begins the interrupt handling routine. The microprogram sequencer saves the interrupt return address on an internal stack.

Traps are unexpected situations caused by the current microinstruction, which must be handled before the end of the current microcycle. Conditions such as overflow can be trapped so corrective action can be taken.

Suppose the current instruction requires a read from memory locations A and B, a floating point addition using this data, and a write of the result



Two dual-port register files can be configured as a six-port register file. With this setup, four reads and two writes may be performed every cycle.

back into location A. If this addition operation results in an overflow and the result is written back into location A, information may be lost. This may happen if the OVERFLOW flag is used to generate an interrupt. A trapping setup, however, offers a different scenario.

If the OVERFLOW flag is used to indicate a trap, the operation can be interrupted before the overflow result can be written over the data in memory location A. An overflow trap handling routine scales both operands. Upon re-execution of the addition operation, the result does not overflow. The microprogram sequencer pushes the address of the current microinstruction onto its internal stack and allows the trap handling address to gain access to the microprogram address bus. After completion of the trap handling routine, the trapped instruction address is popped from the stack and re-executed.

Multiway branching

The Am29331 address sequencer's multiway branch instructions allow the selection of 16 consecutive addresses as a branch target. Generated in a single cycle, the address consists of the upper 12 bits from the D bus concatenated with 4 bits from the multiway inputs. This type of branching allows the testing of up to four conditions in a single clock cycle.

Four multiway sets of 4 bits each allow designers to group test conditions according to type. The 2 least significant bits of the D bus, D0 and D1, control which 4-bit multiway is selected.

The multiway-branch feature provides designers with a hardware solution to the problem of performing certain high level software instructions in a single cycle. Many combinations of conditions could be arranged. For example, CARRY, ZERO, and NEGATIVE flags from the Am29332 ALU chip might be used to perform If (AND/OR)-Then (AND/OR)-Else operations. By using multiway branching, the AND/OR functions in the If-Then-Else statement do not incur the penalty of additional gates or additional delay.

BIBLIOGRAPHY—LITERATURE—BIBLIOGRAPHIE—BIBLIOGRAFIA— 参考文献

Belgium

Chu, P., New, B.J.—"Des blocs 32-bit microprogrammables destinés à un large éventail d'applications." *Panelectronics,* Novembre 1984.

France

Chu, P., New, B.J.—"Des blocs 32-bit microprogrammables destinés à un large éventail d'applications." *Electronique Techniques et Industries*, Octobre 1984.

Grosvalet, F.—"AMD introduit une famille de processeurs bipolaires 32 bits." *Electronique Actualites*, 9 Novembre 1984.

"La Famille Am29300 d'AMD: pour réaliser des systèmes 32-bit a hautes performances." *Minis et Micros*, Novembre 1984.

Germany

"Bipolare 32-Bit-Baustein-Familie," *Elektronik Informationen*, November 1984.

"Bipolare 32-Bit-Prozessorfamilie," *Markt & Technik*, 9 November 1984.

Chu, Paul and Bernard J. New. "Mikroprogrammierbare 32-Bit-Bausteine," *Elektronik*, 2 November 1984.

Renz, Udo. "μP-Building-Blocks-neue 32-Bit-Familie von AMD," *Der Elektroniker*, Dezember 1984.

"Zentraleinheit Künftiger Minicomputer," *Markt & Technik*, 16 November 1984.

Italy

Galleni, S. "Componenti bipolari AMD ad alte prestazioni," *Elettronica Oggi*, Gennaio 1985.

Chu, Paul and New, Bernard J. "Una direttissima verso i 32 bit," *Elettronica Domani*, Novembre 1984.

Japan

ポール・チュー

バーナード・J・ニュー,

機能単位の分割法を採用した32ビット・

バイポーラLSIファミリAm29300,

・日経エレクトロニクス, 1984年11月19日号, NO. 356, PP. 275-290.

32ビットバイポーラプロセッサー

米AMDが発表

Dempa Shinbun, 1984年11月14日号.

United Kingdom

"AMD to Introduce 32-Bit Bipolar Processors," *Microforecast*, 16 November 1984.

Chu, Paul and Bernard J. New. "32-Bit Microprogrammable Building Blocks," *Electronic Product Design*, November 1984.

Holder, Keith. "Manufacturers Enthuse At 32-Bit Bipolar Debut," *Computer Weekly*, 15 November 1984.

Parry, Simon. "AMD Bipolar Trio Cuts Design Limits," *Electronics Times*, 8 November 1984.

Parry, Simon. "Chip Chain for 32-Bit Microprocessor," *Electronics Times*, 28 February 1985.

Sylvester, David. "Instant Answer From High Speed Chips," *Electronics Times*, 28 February 1985.

"32-bit bipolars at AMD," *Electronics Weekly*, 28 November 1984.

United States

"Advanced Micro Devices Offers 32-Bit Bipolar MPU Family," *Electronic News*, November 12, 1984.

Baker, Stan. "AMD: 1st 32-Bit Bipolar μP Line," *Electronic Engineering Times*, November 12, 1984.

Barney, Clifford. "32-Bit Chip Integrates Bit-Slice Functions," *Electronics Week*, November 12, 1984.

DiDio, Laura. "AMD Unveils Bipolar 32-Bit MPU," *Electronic Buyer's News*, November 12, 1984.

"First 32-Bit Bipolar CPU," *Engineering Manager*, December 1984.

"μPs Soar to New Performance Dimensions," *EDN*, November 15, 1984.

Sylvester, David. "AMD Introduces Some New Chips Off the Old Block," *San Jose Mercury News*, December 31, 1984.

"32-Bit Floating-Point IC Heralds Appearance of High-Performance Family," *Electronic Products*, February 15, 1985.

Williams, Tom. "Bipolar Building Blocks Well-Suited for Fast, Flexible 32-Bit CPUs," *Computer Design*, December 1984.

Am29323

32-Bit Parallel Multiplier

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- **32-Bit Three-Bus Architecture**
  - The device has two 32-bit input ports and one 32-bit output port with maximum multiply time of 80ns
- **Single Clock with Register Enables**
  - The Am29323 is controlled by one clock with individual register enables
- **Supports Multiprecision Multiplication**
  - The device has dual 32-bit registers on each data input port to perform multiprecision multiplication
- **Registers can be made transparent**
  - Input and output registers can be made transparent independently to eliminate unwanted pipeline delay
- **Supports Two's Complement, Unsigned or Mixed Numbers**
- **Data Integrity Through Master-Slave Mode and Parity Check/Generate**
  - Parity check/generate catches inter-device connection errors and master/slave mode provides complete function check

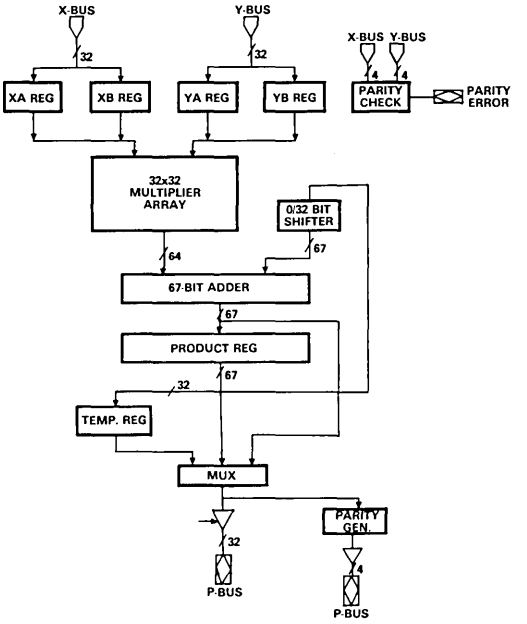
GENERAL DESCRIPTION

The Am29323 is a high-speed 32 x 32-Bit Parallel Multiplier with 67-Bit Accumulator. The part is designed to maximize system level performance by providing a 32-bit three bus architecture and a single clock with register enables.

The Am29323 further enhances the system throughput by providing individual register feedthrough controls, byte parity checking on both input ports and generation on the output port, and dual input registers on each data input bus to support multiprecision multiplication. The Am29323 can manage a wide variety of data types, including two's complement, unsigned, or mixed mode input formats. A 64 x 64-bit multiplication can be performed in seven clock cycles, including input and output. Additional features provided are a format adjust control allowing for standard output or left shifted output suitable for fractional two's complement arithmetic, rounding, and master/slave operation.

The Am29323 is designed with the IMOX* process, which allows internal ECL circuits with TTL-compatible I/O. The device is housed in a 168-lead pin-grid-array package.

SIMPLIFIED BLOCK DIAGRAM



BD005250

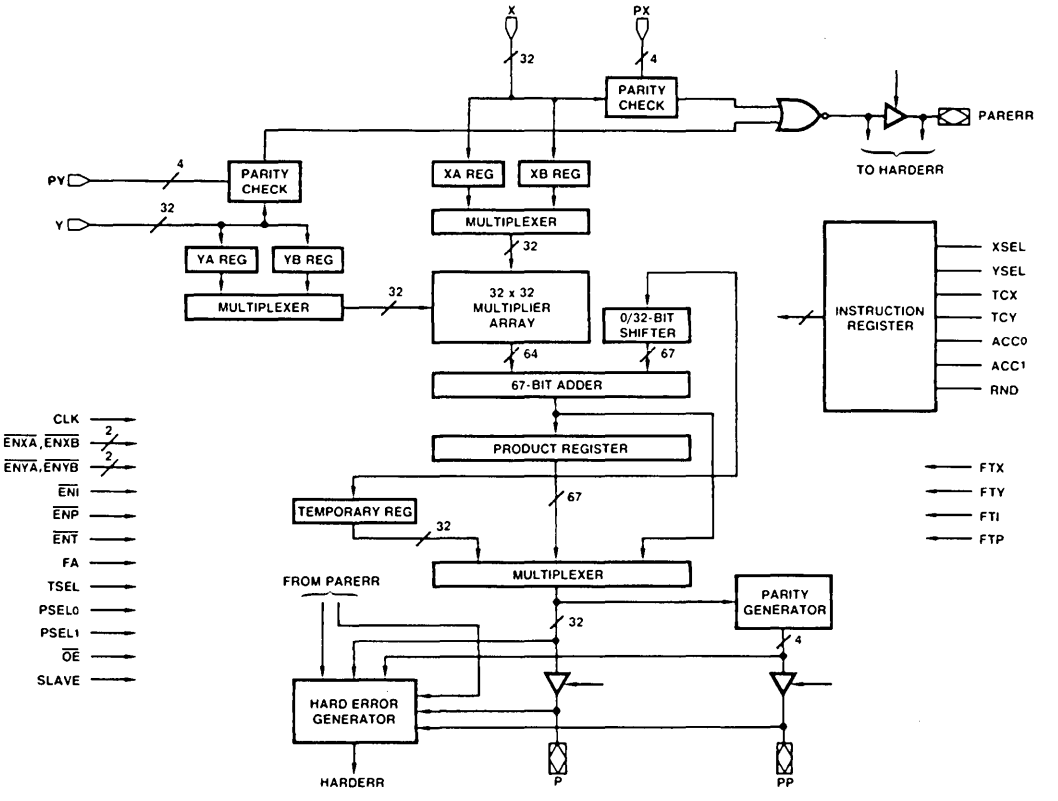
*IMOX is a trademark of Advanced Micro Devices, Inc.

Order # 05763B

RELATED PRODUCTS

Part No.	Description
Am29325	32-Bit Floating Point Processor
Am29331	16-Bit Microprogram Sequencer
Am29332	32-Bit Extended Function ALU
Am29334	64 x 18 Four-Port, Dual Access Register File

BLOCK DIAGRAM



BD003041

PIN DESCRIPTION

X_{31}-X_0	Multiplicand data input port.
Y_{31}-Y_0	Multiplier data input port.
P_{31}-P_0	Product output port.
TCX, TCY	Mode control inputs for each input data word; LOW for unsigned data and HIGH for two's complement format.
ACC1, ACC0	Accumulator control lines used to determine accumulator function; PASS, ACCUMULATE, SHIFT/ACCUMULATE.
RND	Round control for rounding the most significant product.
CLK	Clock; all registers.
$\overline{ENXA}$, $\overline{ENXB}$	Register enables for multiplicand data input registers (XA and XB).
$\overline{ENYA}$, $\overline{ENYB}$	Register enables for multiplier data input registers (YA and YB).
$\overline{ENP}$	Register enable for accumulator product register (P).
$\overline{ENI}$	Register enable for instruction register (I).
$\overline{ENT}$	Register enable for temporary register (T).
XSEL	Control line used to route the contents of either the XA register (HIGH) or XB register (LOW) into the multiplier array.
YSEL	Control line used to route the contents of either the YA register (HIGH) or YB register (LOW) into the multiplier array.
FA	Format adjust select either a full 64-bit product (HIGH) or a left-shifted 63-bit product suitable for fractional two's complement arithmetic (LOW).
TSEL	Select control line used to route the most significant product register (HIGH) or the least significant product register (LOW) into the temporary register.
FTX, FTY, FTI	Feedthrough control lines for X, Y, and I registers.
FTP	Bypass control for output multiplexer.
PSEL1, PSEL0	Product control lines used to select desired output including disabling P output port.
PX_3-PX_0	Byte parity inputs on X input port.
PY_3-PY_0	Byte parity inputs on Y input port.
PP_3-PP_0	Byte parity outputs on P output port.
PARERR	Parity error flag indicates a parity error on the input buses.
$\overline{OE}$	Output enable control line used to disable the P output port.
SLAVE	$\overline{Master}$/Slave control line used to determine mode of operation.
HARDERR	Hard error flag used when two Am29323s are configured as master and slave to indicate hardware errors.

FUNCTIONAL DESCRIPTION

Architecture

The Am29323 comprises a high speed 32 by 32-bit multiplier array, a 67-bit accumulator, and a 32-bit data path.

Multiplier Array

The multiplier is a 32 by 32-bit array which produces a 64-bit product. This product is then fed to the accumulator section.

Accumulator

The accumulator is 67 bits wide. It performs accumulation for sum of product operations and multiprecision multiplication operations. The accumulator can perform three operations: store product without accumulation, accumulate product, and shift accumulator value and accumulate with product.

Data Path

The 32-bit data path consists of X and Y input buses; the P output bus; data registers XA, XB, YA, YB, and the product accumulator; two multiplier input multiplexers; byte parity input checkers; byte parity output generators; and master/slave comparators. Input operands enter the device through the two 32-bit input buses, $X_0 - X_{31}$ and $Y_0 - Y_{31}$. These operands may then be stored in one of the two registers for each bus (XA or XB for X, YA or YB for Y) or they may be fed directly through to the multiplier array. Input parity checking is performed as soon as the operands are put on the input buses. The signals used for output parity generation are taken from the input side of the output translator.

Operational Modes

The Am29323 can perform signed, unsigned, or mixed mode multiplication. These different numerical representations are controlled by TCX and TCY. A HIGH input on one of these lines indicates to the device that the respective input should be treated as a two's complement number; a LOW, an unsigned number. The output format is unsigned when both inputs are unsigned. The output format is two's complement when either or both inputs are two's complement.

Command Description and Formats

The accumulator is controlled by ACC0 and ACC1. These lines are used to select any of the three operations that the accumulator can perform. This instruction set is described in Table 1.

The temporary output register is controlled by TSEL and FA. These lines are used to select any of the four different sets of data that can be stored in the temporary register. This instruction set is described in Table 2.

The output multiplexer is controlled by PSEL0, PSEL1, and FA. These lines are used to select any of the five different sets of data that can be output through the P port. PSEL0 and PSEL1 can also be used to disable the outputs. (This instruction is independent of $\overline{OE}$.) This instruction set is described in Table 3.

Format Adjust (FA) is used to select either a full 64-bit product or a left-shifted 63-bit product suitable for fractional two's complement arithmetic. This shifting increases the precision of the upper half of the product word by eliminating the redun-

dant sign bit. Output Data Formats shows the effect of FA. (page 5).

User Visible Register Descriptions

The Am29323 contains seven different register sets, each with its own clock enable. Two 32-bit registers are attached to each of the input data buses. These registers are differentiated by the suffix A or B. For example, the X bus has registers XA and XB. The 67-bit accumulator register can be used as a regular product register when the part is used as a multiplier only or as the register part of the accumulator section. The 32-bit temporary output register is included to aid in the pipelining of multiprecision multiplication operations. An instruction register is also provided.

All of these registers can be made transparent with the exception of the accumulator register and the temporary register. The product from the multiplier can be fed directly to the output by using the FTP control line.

TABLE 1. ACCUMULATOR OPERATION INSTRUCTIONS

ACC1	ACC0	Accumulator Operation
0	0	PASS
0	1	ACCUMULATE
1	0	INVALID
1	1	SHIFT AND ACCUMULATE

TABLE 2. INPUT SELECT INSTRUCTIONS FOR TEMPORARY (T) REGISTER

TSEL	FA	Temp Reg Input
0	0	P_{i-1}
0	1	P_i
1	0	P_{i+31}
1	1	P_{i+32}

TABLE 3. OUTPUT SELECT INSTRUCTIONS FOR PRODUCT (P) PORT

PSEL1	PSEL0	FA	P Port Output
0	0	X	TEMP REGISTER
0	1	0	P_{i-1}
0	1	1	P_i
1	0	0	P_{i+31}
1	0	1	P_{i+32}
1	1	X	DISABLE

Am29323 X AND Y INPUT DATA FORMATS

Fractional Two's Complement

TCX, TCY = 1

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}						2^{-28}	2^{-29}	2^{-30}	2^{-31}

Integer Two's Complement

TCX, TCY = 1

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
-2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}						2^3	2^2	2^1	2^0

Unsigned Fractional

TCX, TCY = 0

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}						2^{-29}	2^{-30}	2^{-31}	2^{-32}

Unsigned Integer

TCX, TCY = 0

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}						2^3	2^2	2^1	2^0

Am29323 P-PORT OUTPUT DATA FORMATS

Fractional Two's Complement (Shifted)*

FA = 0, PSEL1 = 1, PSEL0 = 0

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
-2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}						2^{-28}	2^{-29}	2^{-30}	2^{-31}

FA = 0, PSEL1 = 0, PSEL0 = 1

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{-32}	2^{-33}	2^{-34}	2^{-35}	2^{-36}	2^{-37}						2^{-60}	2^{-61}	2^{-62}	2^{-63}**

Fractional Two's Complement

FA = 1, PSEL1 = 1, PSEL0 = 0

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
-2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}						2^{-27}	2^{-28}	2^{-29}	2^{-30}

FA = 1, PSEL1 = 0, PSEL0 = 1

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{-31}	2^{-32}	2^{-33}	2^{-34}	2^{-35}	2^{-36}						2^{-59}	2^{-60}	2^{-61}	2^{-62}

Integer Two's Complement

FA = 1, PSEL1 = 1, PSEL0 = 0

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
-2^{63}	2^{62}	2^{61}	2^{60}	2^{59}	2^{58}						2^{35}	2^{34}	2^{33}	2^{32}

FA = 1, PSEL1 = 0, PSEL0 = 1

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}						2^{3}	2^{2}	2^{1}	2^{0}

Unsigned Fractional

FA = 1, PSEL1 = 1, PSEL0 = 0

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}						2^{-29}	2^{-30}	2^{-31}	2^{-32}

FA = 1, PSEL1 = 0, PSEL0 = 1

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{-33}	2^{-34}	2^{-35}	2^{-36}	2^{-37}	2^{-38}						2^{-61}	2^{-62}	2^{-63}	2^{-64}

Unsigned Integer

FA = 1, PSEL1 = 1, PSEL0 = 0

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{63}	2^{62}	2^{61}	2^{60}	2^{59}	2^{58}						2^{35}	2^{34}	2^{33}	2^{32}

FA = 1, PSEL1 = 0, PSEL0 = 1

31	30	29	28	27	26	–	–	–	–	–	3	2	1	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}						2^{3}	2^{2}	2^{1}	2^{0}

*In this format, an overflow occurs in the attempted multiplication of the two's complement number −1.000 with itself, yielding a product of +1.000 which cannot be represented in this format. **This bit position (2^{-63}) equals zero in this format.

264 x 64 Multiplication

To perform a 64 x 64-bit multiplication using the Am29323, each 64-bit input must be split into two 32-bit inputs; a most significant half and a least significant half (XW1 and XW0 or YW1 and YW0, respectively.) These 32-bit inputs are then used to perform the four multiplications needed to obtain the 128-bit product. This product is represented in four 32-bit words, $PW_3 - PW_0$. The least significant word being PW_0. The product is output 32 bits at a time through the product (P) port. The following equation shows the required multiplications:

$$X * Y = ((XW1 * YW1) * 2^{64}) + ((XW0 * YW1) * 2^{32} + ((XW1 * YW0) * 2^{32}) + ((XW0 * YW0) * 2^{0})$$

$$P = (PW3 * 2^{96}) + (PW2 * 2^{64}) + (PW1 * 2^{32}) + (PW0 * 2^{0})$$

The Am29323 uses an internal accumulator to sum these intermediate products. The previous equation, in a slightly different form, is shown with the necessary instructions below:

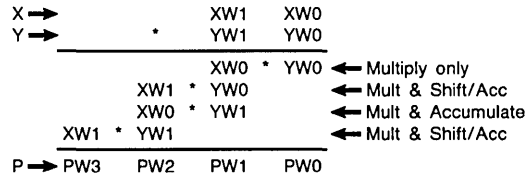


Table 4 details the movement of the input operands through the Am29323. Table 5 defines the microcode required to perform a signed 64 x 64-bit multiplication. For an unsigned multiplication, TCX and TCY are LOW for all cycles. The operations and data movement are scheduled to produce a single product in seven clock cycles or a new pipelined product every four clock cycles.

TABLE 4. BUS AND REGISTER CONTENTS FOR A 64 x 64-BIT SIGNED MULTIPLICATION WITH ONE COMPLETE EXTENDED MULTIPLICATION SHOWN IN THE UNSHADED CYCLES

Cycle	0	1	2	3	4	5	6
X BUS	XW0	XW1			XW0	XW1	
XA REG	XW0	XW0	XW0	XW0	XW0	XW0	XW0
XB REG	XW1	XW1	XW1	XW1	XW1	XW1	XW1
Y BUS	YW0	YW1			YW0	YW1	
YA REG	YW0	YW0	YW0	YW0	YW0	YW0	YW0
YB REG	YW1	YW1	YW1	YW1	YW1	YW1	YW1
MPY OP	W1*W1	W0*W0	W1*W0	W0*W1	W1*W1	W0*W0	W1*W0
ACC OP	S/A	PASS	S/A	ACC	S/A	PASS	S/A
T REG		P3	P0			P3	P0
P BUS	P1	P2	P3	P0	P1	P2	P3

Note: MPY OP = Operation of multiplier array (X*Y)
ACC OP = Operation of internal accumulator
PASS = Pass through multiplier product
ACC = Add previous result to current product
S/A = Shift previous result then add to current product

TABLE 5. INSTRUCTION MICROCODE FOR 64 x 64-BIT SIGNED MULTIPLICATION WITH ONE COMPLETE EXTENDED MULTIPLICATION SHOWN IN THE UNSHADED CYCLES

Cycle	0	1	2	3	4	5	6	7	8	9	A	B	C	D
$\overline{ENXA}$	0	1	1	1	0	1	1	1	0	1	1	1	0	1
$\overline{ENXB}$	1	0	1	1	1	0	1	1	1	0	1	1	1	0
TCX	0	1	0	1	0	1	0	1	0	1	0	1	0	1
XSEL	1	0	1	0	1	0	1	0	1	0	1	0	1	0
$\overline{ENYA}$	0	1	1	1	0	1	1	1	0	1	1	1	0	1
$\overline{ENYB}$	1	0	1	1	1	0	1	1	1	0	1	1	1	0
TCY	0	0	1	1	0	0	1	1	0	0	1	1	0	0
YSEL	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$\overline{ENI}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{ENT}$	1	0	0	1	1	0	0	1	1	0	0	1	1	0
TSEL	X	1	0	X	X	1	0	X	X	1	0	X	X	1
ACC0	0	1	1	1	0	1	1	1	0	1	1	1	0	1
ACC1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$\overline{ENP}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PSEL0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
PSEL1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Am29325

32-Bit Floating Point Processor

PRELIMINARY

DISTINCTIVE CHARACTERISTICS

- Single VLSI device performs high-speed floating-point arithmetic
  - Floating-point addition, subtraction and multiplication in a single clock cycle
  - Internal architecture supports sum-of-products, Newton-Raphson division
- 32-bit, 3-bus flow-through architecture
  - Programmable I/O allows interface to 32- and 16-bit systems
- IEEE and DEC formats
  - Performs conversions between formats
  - Performs integer ◄► floating point conversions
- Six flags indicate operation status
- Register enables eliminate clock skew
- Input and output registers can be made transparent independently

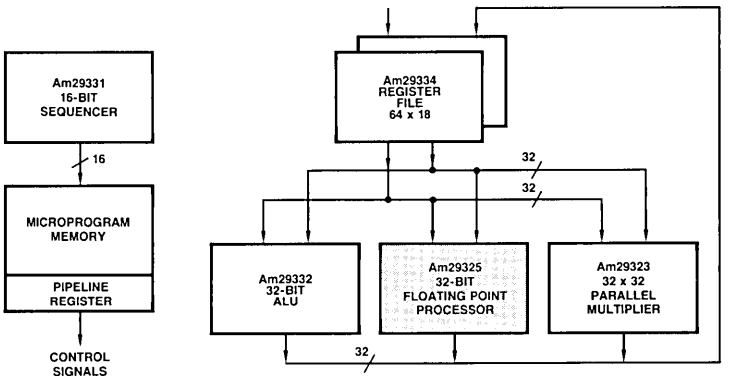
GENERAL DESCRIPTION

The Am29325 is a high-speed floating-point processor unit. It performs 32-bit single-precision floating-point addition, subtraction, and multiplication operations in a single LSI integrated circuit, using the format specified by the proposed IEEE floating-point standard P754. The DEC single-precision floating-point format is also supported. Operations for conversion between 32-bit integer format and floating-point format are available, as are operations for converting between the IEEE and DEC floating-point formats. Any operation can be performed in a single clock cycle. Six flags – invalid operation, inexact result, zero, not-a-number, overflow, and underflow – monitor the status of operations.

The Am29325 has a 3-bus, 32-bit architecture, with two input buses and one output bus. This configuration provides high I/O bandwidth, allows access to all buses and affords a high degree of flexibility when connecting this device in a system. All buses are registered, with each register having a clock enable. Input and output registers may be made transparent independently. Two other I/O configurations, a 32-bit, 2-bus architecture and a 16-bit, 3-bus architecture, are user-selectable, easing interface with a wide variety of systems. Thirty-two-bit internal feedforward data paths support accumulation operations, including sum-of-products and Newton-Raphson division.

Fabricated with the high-speed IMOX™ bipolar process, the Am29325 is powered by a single 5-volt supply. The device is housed in a 144-pin pin-grid-array package.

Am29300 FAMILY HIGH PERFORMANCE SYSTEM BLOCK DIAGRAM



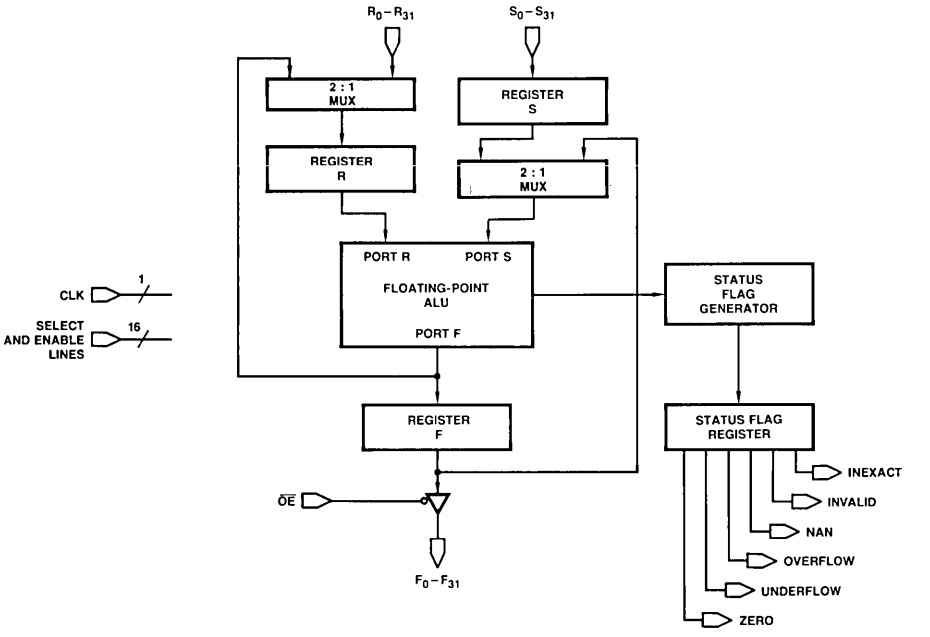
05621A-1

RELATED PRODUCTS

- Am29323 – 32 x 32 Parallel Multiplier
- Am29332 – 32-Bit ALU
- Am29331 – 16-Bit Sequencer
- Am29334 – 64 x 18 Four-Port Dual-Access Register File

Order # 05621B

BLOCK DIAGRAM
Am29325



05621B-2

DEFINITION OF TERMS

AFFINE MODE

One of two modes affecting the handling of operations on infinities – see the **Operations with Infinities** section under **Operation in IEEE Mode** below.

BIASED EXPONENT

The true exponent of a floating-point number, plus a constant. For IEEE floating-point numbers, the constant is 127; for DEC floating-point numbers, the constant is 128. See also **True Exponent**.

BUS

Data input or output channel for the floating-point processor.

DEC RESERVED OPERAND

A DEC floating-point number that is interpreted as a symbol and has no numeric value. A DEC reserved operand has a sign of 1 and a biased exponent of 0.

DESTINATION FORMAT

The format of the final result produced by the floating-point ALU. The destination format can be IEEE floating-point, DEC floating-point or integer.

FINAL RESULT

The result produced by the floating-point ALU.

FRACTION

The twenty-three least-significant bits of the mantissa.

INFINITELY PRECISE RESULT

The result that would be obtained from an operation if both exponent range and precision were unbounded.

INPUT OPERANDS

The value or values on which an operation is performed. For example, the addition 2 + 3 = 5 has input operands 2 and 3.

MANTISSA

The portion of a floating-point number containing the number's significant bits. For the floating-point number 1.101 x 2^{-3}, the mantissa is 1.101.

DEFINITION OF TERMS (Cont)

NAN (Not-a-Number)

An IEEE floating-point number that is interpreted as a symbol, and has no numeric value. A NAN has a biased exponent of 255_{10} and a non-zero fraction.

PORT

Data input or output channel for the floating-point ALU.

PROJECTIVE MODE

One of two modes affecting the handling of operations on infinities – see the **Operations with Infinities** section under **Operation in IEEE Mode** below.

ROUNDED RESULT

The result produced by rounding the infinitely precise result to fit the destination format.

TRUE EXPONENT (or Exponent)

Number representing the power of two by which a floating-point number's mantissa is to be multiplied. For the floating-point number 1.101×2^{-3}, the true exponent is -3.

PIN DESCRIPTION

R_0-R_{31}	R operand bus, input. R_0 is the least-significant bit.
S_0-S_{31}	S operand bus, input. S_0 is the least-significant bit.
F_0-F_{31}	F operand bus, output. F_0 is the least-significant bit.
CLK	Clock input for the internal registers.
$\overline{ENR}$	Register R clock enable, input. When $\overline{ENR}$ is LOW, register R is clocked on the LOW-to-HIGH transition of CLK. When $\overline{ENR}$ is HIGH, register R retains the previous contents.
$\overline{ENS}$	Register S clock enable, input. When $\overline{ENS}$ is LOW, register S is clocked on the LOW-to-HIGH transition of CLK. When $\overline{ENS}$ is HIGH, register S retains the previous contents.
$\overline{ENF}$	Register F clock enable, input. When $\overline{ENF}$ is LOW, register F is clocked on the LOW-to-HIGH transition of CLK. When $\overline{ENF}$ is HIGH, register F retains the previous contents.
FT_0	Input register feedthrough control, input. When FT_0 is HIGH, registers R and S are transparent.
FT_1	Output register feedthrough control, input. When FT_1 is HIGH, register F and the status flag register are transparent.
I_0-I_2	Operation select lines, inputs. Used to select the operation to be performed by the ALU. See the **ALU Operation Select Table** for a list of operations and the corresponding codes.
I_3	ALU S port input select, input. A LOW on I_3 selects register S as the input to the ALU S port. A HIGH on I_3 selects register F as the input to the ALU S port.
I_4	Register R input select, input. A LOW on I_4 selects R_0-R_{31} as the input to register R. A HIGH selects the ALU F port as the input to register R.
IEEE/$\overline{DEC}$	IEEE/DEC mode select, input. When IEEE/$\overline{DEC}$ is HIGH, IEEE mode is selected. When IEEE/$\overline{DEC}$ is LOW, DEC mode is selected.
INEXACT	Inexact result flag, output. A HIGH indicates that the final result of the last operation was not infinitely precise, due to rounding.
INVALID	Invalid operation flag, output. A HIGH indicates that the last operation performed was invalid, e.g., ∞ times 0.
NAN	Not-a-number flag, output. A HIGH indicates that the final result produced by the last operation is not to be interpreted as a number. The output in such cases is either an IEEE Not-a-Number (NAN) or a DEC reserved operand.
$\overline{OE}$	Output enable, input. When $\overline{OE}$ is LOW, the contents of register F are placed on F_0-F_{31}. When $\overline{OE}$ is HIGH, F_0-F_{31} assume a high-impedance state.
ONEBUS	Input bus configuration control, input. A LOW on ONEBUS configures the input bus circuitry for two-input bus operation. A HIGH on ONEBUS configures the input bus circuitry for single-input bus operation.
OVERFLOW	Overflow flag, output. A HIGH indicates that the last operation produced a final result that overflowed the floating-point format.
PROJ/$\overline{AFF}$	Projective/affine mode select, input. Choice of projective or affine mode determines the way in which infinities are handled in IEEE mode. A LOW on PROJ/$\overline{AFF}$ selects affine mode; a HIGH selects projective mode.

PIN DESCRIPTION (Cont)

RND_0, RND_1 Rounding mode selects, inputs. RND_0 and RND_1 select one of four rounding modes. See the **Rounding Mode Select Table** for a list of rounding modes and the corresponding control codes.

S16/$\overline{32}$ Sixteen- or thirty-two-bit I/O mode select, input. A LOW on S16/$\overline{32}$ selects the thirty-two-bit I/O mode; a HIGH selects the sixteen-bit I/O mode. In thirty-two-bit mode, inputs and output buses are 32 bits wide. In sixteen-bit mode, input and output buses are sixteen bits wide, with the least and most significant portions of the thirty-two-bit input and output words being placed on the buses during the HIGH and LOW portions of CLK, respectively.

UNDERFLOW Underflow flag, output. A HIGH indicates that the last operation produced a rounded result that underflowed the floating-point format.

ZERO Zero flag, output. A HIGH indicates that the last operation produced a final result of zero.

ARCHITECTURE

The Am29325 comprises a high-speed, floating-point ALU, a status flag generator, and a 32-bit data path.

Floating-Point ALU

The floating-point ALU performs 32-bit floating-point operations. It also performs floating-point-to-integer conversions, integer-to-floating-point conversions, and conversions between the IEEE and DEC floating-point formats. The ALU has two 32-bit input ports, R and S, and a 32-bit output port, F.

Conceptually, the process performed by the ALU can be divided into three stages – see Figure 1. The operation stage performs the arithmetic operation selected by the user; the output of this section is referred to as the infinitely precise result of the operation. The rounding stage rounds the infinitely precise result to fit in the destination format; the output of this stage is called the rounded result. The last stage checks for exceptional conditions. If no exceptional condition is found, the rounded result is passed through this stage. If some exceptional condition is found, e.g., overflow, underflow, or an invalid operation, this section may replace the rounded result with another output, such as $+\infty$, $-\infty$, a NAN, or a DEC reserved operand. The output of this last stage appears on port F, and is called the final result.

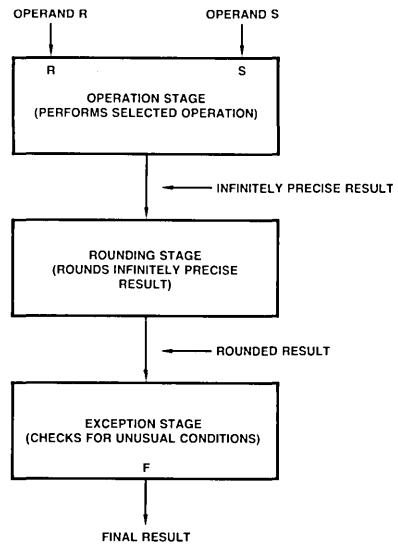
The ALU performs one of eight operations; the operation to be performed is selected by placing the appropriate control code on lines $I_0 - I_2$. The **ALU Operation Select Table** gives the control codes corresponding to each of the eight operations.

The floating-point addition operation (R PLUS S) adds the floating-point numbers on ports R and S, and places the floating-point result on port F. In IEEE mode (IEEE/$\overline{DEC}$ = HIGH) the addition is performed in IEEE floating-point format; in DEC mode (IEEE/$\overline{DEC}$ = LOW) the addition is performed in DEC format.

The floating-point subtraction operation (R MINUS S) subtracts the floating-point number on port S from the floating-point number on port R and places the floating-point result on port F. In IEEE mode (IEEE/$\overline{DEC}$ = HIGH) the subtraction is performed in IEEE floating-point format; in DEC mode (IEEE/$\overline{DEC}$ = LOW) the subtraction is performed in DEC format.

The floating-point multiplication operation (R TIMES S) multiplies the floating-point numbers on ports R and S, and places the floating-point result on port F. In IEEE mode (IEEE/$\overline{DEC}$ = HIGH) the multiplication is performed in IEEE floating-point format; in DEC mode (IEEE/$\overline{DEC}$ = LOW) the multiplication is performed in DEC format.

Figure 1. Conceptual Model of the Process Performed by the Floating-Point ALU



The floating-point constant subtraction (2 MINUS S) operation subtracts the floating-point value on port S from 2, and places the result on port F. The operand on port R is not used in this operation; its value will not affect the operation in any way. In IEEE mode (IEEE/$\overline{DEC}$ = HIGH) the operation is performed in IEEE floating-point format; in DEC mode (IEEE/$\overline{DEC}$ = LOW) the operation is performed in DEC format. This operation is used to support Newton-Raphson floating-point division; a description of its use appears in **Appendix C.**

The integer-to-floating-point conversion (INT-TO-FP) operation takes a 32-bit, two's complement integer on port R and places the equivalent floating-point value on port F. The operand on port S is not used in this operation; its value will not affect the operation in any way. In IEEE mode (IEEE/$\overline{DEC}$ = HIGH) the result is delivered in IEEE format; in DEC mode (IEEE/$\overline{DEC}$ = LOW) the result is delivered in DEC format.

ALU OPERATION SELECT TABLE

I_2	I_1	I_0	Operation	Output Equation
0	0	0	Floating-point addition (R PLUS S)	F = R + S
0	0	1	Floating-point subtraction (R MINUS S)	F = R − S
0	1	0	Floating-point multiplication (R TIMES S)	F = R * S
0	1	1	Floating-point constant subtraction (2 MINUS S)	F = 2 − S
1	0	0	Integer-to-floating-point conversion (INT-TO-FP)	F (floating-point) = R (integer)
1	0	1	Floating-point-to-integer conversion (FP-TO-INT)	F (integer) = R (floating-point)
1	1	0	IEEE-TO-DEC format conversion (IEEE-TO-DEC)	F (DEC format) = R (IEEE format)
1	1	1	DEC-TO-IEEE format conversion (DEC-TO-IEEE)	F (IEEE format) = R (DEC format)

The floating-point-to integer conversion (FP-TO-INT) operation takes a floating-point number on port R and places the equivalent 32-bit, two's complement integer value on port F. The operand on port S is not used in this operation; its value will not affect the operation in any way. In IEEE mode (IEEE/$\overline{\text{DEC}}$ = HIGH) the operand on port R is interpreted using the IEEE floating-point format; in DEC mode (IEEE/$\overline{\text{DEC}}$ = LOW) it is interpreted using the DEC floating-point format.

The IEEE-to-DEC conversion operation (IEEE-TO-DEC) takes an IEEE-format floating-point number on port R and places the equivalent DEC-format floating-point number on port F. The operand on port S is not used in this operation; its value will not affect the operation in any way. The operation can be performed in either IEEE mode (IEEE/$\overline{\text{DEC}}$ = HIGH) or DEC mode (IEEE/$\overline{\text{DEC}}$ = LOW).

The DEC-to-IEEE conversion operation (DEC-TO-IEEE) takes a DEC-format floating-point number on port R and places the equivalent IEEE-format floating-point number on port F. The operand on port S is not used in this operation; its value will not affect the operation in any way. The operation can be performed in either IEEE mode (IEEE/$\overline{\text{DEC}}$ = HIGH) or DEC mode (IEEE/$\overline{\text{DEC}}$ = LOW).

Status Flag Generator

The status flag generator controls the state of six flags that report the status of floating-point ALU operations. The flags indicate when an operation is invalid (e.g., infinity times zero) or when an operation has produced an overflow, an underflow, a non-numerical result (e.g., a NAN or DEC reserved operand), an inexact result, or a result of zero. The flags represent the status of the most-recently-performed operation. Flag status is stored in the flag status register on the LOW-to-HIGH transition of CLK. When the output register feedthrough control FT_1 is HIGH, the flag status register is made transparent.

Data Path

The 32-bit data path consists of the R and S input buses, the F output bus, data registers R, S, and F, the register R input multiplexer, and the ALU port S input multiplexer.

Input operands enter the floating-point processor through the 32-bit R and S input buses, R_0-R_{31} and S_0-S_{31}. Results of operations appear on the 32-bit F bus, F_0-F_{31}. The F bus assumes a high-impedance state when output enable $\overline{\text{OE}}$ is HIGH.

The R and S registers store input operands; the F register stores the final result of the floating-point ALU operation. Each register has an independent clock enable ($\overline{\text{ENR}}$, $\overline{\text{ENS}}$ and $\overline{\text{ENF}}$). When a register's clock enable is LOW, the register stores the data on its input at the LOW-to-HIGH transition of CLK; when the clock enable is HIGH, the register retains its current data. All data registers are fully edge-triggered – both the input data and the register enable need only meet modest setup and hold time requirements. Registers R and S can be made transparent by setting FT_0, the input register feedthrough control, HIGH. Register F can be made transparent by setting FT_1, the output register feedthrough control, HIGH.

The register R input multiplexer selects either the R input bus or the floating-point ALU's F port as the input to register R. Selection is controlled by I_4 – a LOW selects the R input bus; a HIGH selects the ALU F port. The ALU port S input multiplexer selects either register S or register F as the input to the floating-point ALU's S port. Selection is controlled by I_3 – a LOW selects register S; a HIGH selects register F.

Data selected by I_3 and I_4 is described in the **Mux Select Tables**. When registers R and S are transparent (FT_0 = HIGH) multiplexer select I_4 must be kept LOW, so that the register R input multiplexer selects R_0-R_{31}. When register F is transparent (FT_1 = HIGH) multiplexer select I_3 must be kept LOW, so that the ALU port S input multiplexer selects register S.

MUX SELECT TABLES

I_3	Data selected for floating-point ALU S port
0	Register S
1	Register F

I_4	Data selected for register R input
0	R bus
1	Floating-point ALU port F

I/O MODES

The Am29325 data path can be configured in one of three I/O modes: a 32-bit, two-input-bus mode; a 32-bit, single-input-bus mode; and a 16-bit, two-input-bus mode. These modes affect only the manner in which data is delivered to and taken from the Am29325; operation of the floating-point ALU is not altered. The I/O mode is selected with the ONEBUS and S16/$\overline{32}$ controls. The **I/O Mode Selection Table** lists the control codes needed to invoke each I/O mode.

I/O MODE SELECTION TABLE

S16/$\overline{32}$	ONEBUS	I/O Mode
0	0	32-bit, two-input-bus mode
0	1	32-bit, single-input-bus mode(•)
1	0	16-bit, two-input-bus mode(•)
1	1	Illegal I/O mode selection value

(•)FT_0 must be held LOW in this mode (see text).

32-Bit, Two-Input-Bus Mode

In this I/O mode, the R and S buses are configured as independent 32-bit input buses, and the F bus is configured as a 32-bit output bus. Figure 2 is a functional block diagram of the Am29325 in this I/O mode.

R and S operands are taken from their respective input buses and clocked into the R and S registers on the LOW-to-HIGH transition of CLK. Register F is also clocked on the LOW-to-HIGH transition of CLK. Figure 5(a.) depicts typical I/O timing in this mode.

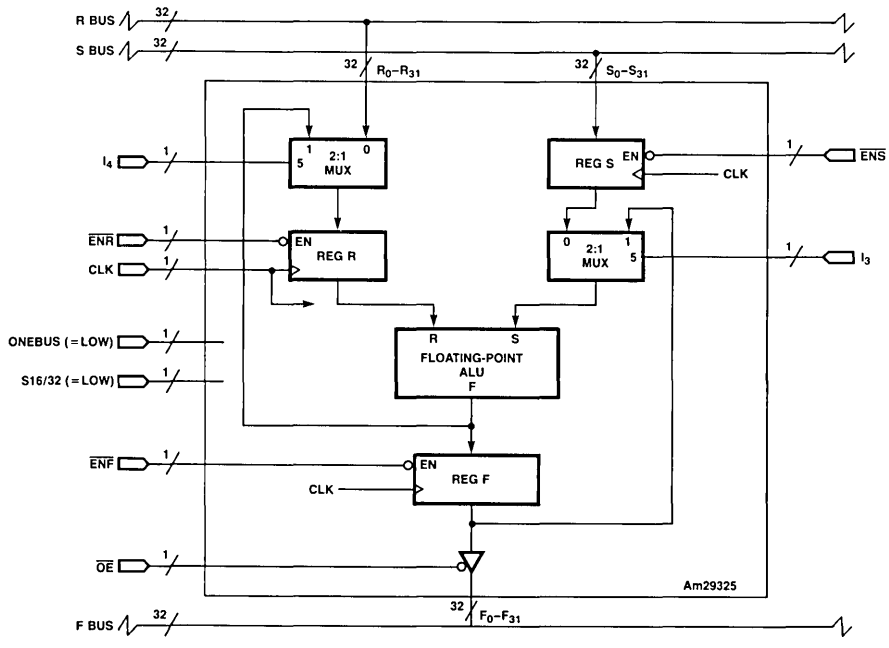
32-Bit, Single-Input-Bus Mode

In this I/O mode, the R and S buses are connected to a single 32-bit multiplexed input data bus; the F bus is configured as an independent 32-bit output bus. Figure 3 is a functional block diagram of the Am29325 in this I/O mode. Note that both the R and S bus lines must be wired to the input bus.

R and S operands are multiplexed onto the input bus by the host system. The S operand is clocked from the input bus into a temporary holding register on the HIGH-to-LOW transition of CLK and is transferred to register S on the LOW-to-HIGH transition of CLK. The R operand is clocked from the input bus into register R on the LOW-to-HIGH transition of CLK. Register F is clocked on the LOW-to-HIGH transition of CLK. Figure 5(b.) depicts typical I/O timing in this mode.

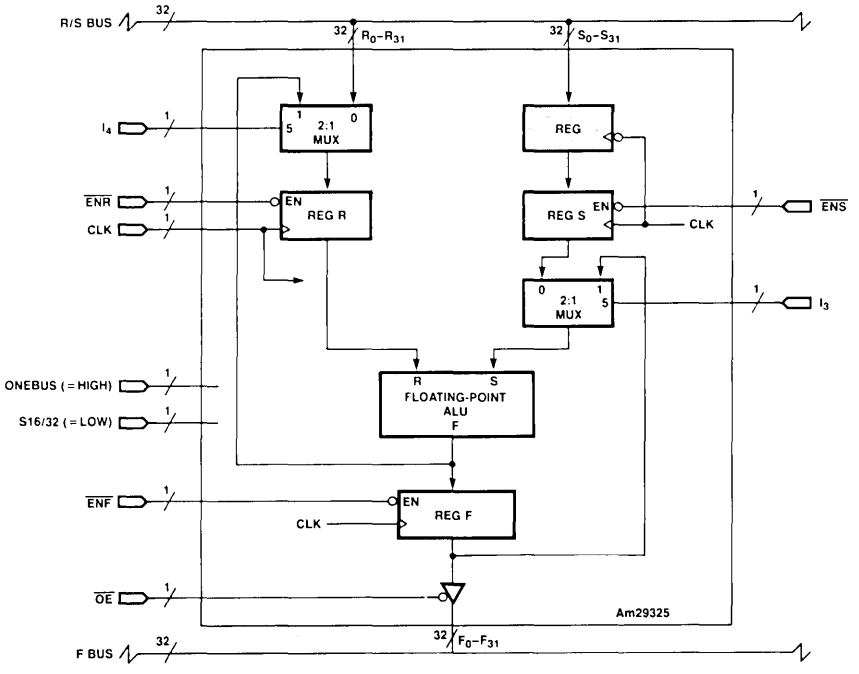
When placed in this I/O mode, the data path will not function properly if the R and S registers are made transparent. Therefore input register feedthrough control FT_0 must be held LOW in this mode.

Figure 2. Functional Block Diagram for the 32-Bit, Two-Input-Bus Mode



05621B-4

Figure 3. Functional Block Diagram for the 32-Bit, Single-Input-Bus Mode



16-Bit, Two-Input-Bus Mode

In this I/O mode, the R and S buses are configured as independent 16-bit input buses, and the F bus is configured as a 16-bit output bus. Figure 4 is a functional block diagram of the Am29325 in this I/O mode. Note that the 16 LSBs and 16 MSBs of the R, S and F buses must be wired to their respective system buses in parallel.

Thirty-two-bit operands are passed along the 16-bit data buses by time-multiplexing the 16 LSBs and 16 MSBs of each 32-bit word. For the R input bus, the host system multiplexes the 16 LSBs and 16 MSBs of the R operand onto the 16-bit R bus. The 16 LSBs of the R operand are stored in a temporary holding register on the HIGH-to-LOW transition of CLK. The 16 MSBs are clocked into register R on the LOW-to-HIGH transition of CLK; at the same time, the 16 LSBs are transferred from the temporary holding register to register R. Transfer of data from the S input bus to the S register takes place in a similar fashion. Register F is clocked on the LOW-to-HIGH transition of CLK. Circuitry internal to the Am29325 multiplexes data from register F onto the 16-bit output bus by enabling the 16 LSBs of the F output bus when CLK is HIGH, and enabling the 16 MSBs of the F output bus when CLK is LOW. Figure 5(c.) depicts typical I/O timing in this mode.

When placed in this I/O mode, the data path will not function properly if the R and S registers are made transparent. Therefore input register feedthrough control FT_0 must be held LOW in this mode. Caution must also be taken in controlling the register R input multiplexer control line, I_4, in this I/O mode. I_4 should be changed only when CLK is HIGH, in addition to meeting the setup and hold time requirements given in the **Switching Characteristics** section.

OPERATION IN IEEE MODE

When input signal IEEE/$\overline{\text{DEC}}$ is HIGH, the IEEE mode of operation is selected. In this mode the Am29325 uses the floating-point format set forth in the IEEE Proposed Standard for Binary Floating-Point Arithmetic, P754. In addition, the IEEE mode complies with most other aspects of single-precision floating-point operation outlined in the proposed standard – differences are discussed in **Appendix A.**

IEEE Floating-Point Format

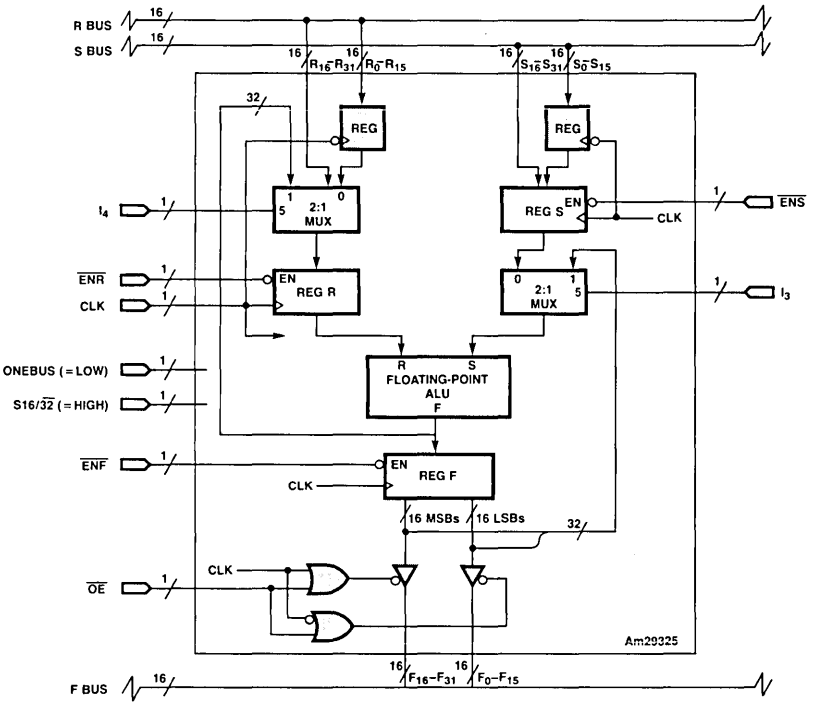
The IEEE single-precision floating-point word is thirty-two bits wide, and is arranged in the format shown in Figure 6. The floating-point word is divided into three fields: a single-bit sign, an eight-bit biased exponent, and a 23-bit fraction.

The sign bit indicates the sign of the floating-point number's value. Non-negative values have a sign of 0; negative values, a sign of 1. The value zero may have either sign.

The biased exponent is an eight-bit unsigned integer field representing a multiplicative factor of some power of two. The bias value is 127. If, for example, the multiplicative factor for a floating-point number is to be 2^a, the value of the biased exponent would be a+127; a is called the true exponent.

The fraction is a 23-bit unsigned fractional field containing the 23 least-significant bits of the floating-point number's 24-bit mantissa. The weight of fraction's most significant bit is 2^{-1}; the weight of the least-significant bit is 2^{-23}.

Figure 4. Functional Block Diagram for the 16-Bit, Two-Input-Bus Mode



05621B-6

A floating-point number is evaluated or interpreted per the following conventions:

let s = sign bit
e = biased exponent
f = fraction

if e = 0 and f = 0 . . . value = $(-1)^{S*}(0)$ (+0, −0)

if e = 0 and f ‡ 0 . . . value = denormalized number

if 0 < e < 255 . . value = $(-1)^{S*}(2^{e-127})*(1.f)$ (normalized number)

if e = 255 and f = 0 . . value = $(-1)^{S*}(\infty)$ ($+\infty$, $-\infty$)

if e = 255 and f ‡ 0 . . . value = not-a-number (NAN)

Zero – The value zero can have either a positive or negative sign. Rules for determining the sign of a zero produced by an operation are given in the **Sign Bit** section on page 12.

Denormalized Number – A denormalized number represents a quantity with magnitude less than 2^{-126} but greater than zero.

Normalized Number – A normalized number represents a quantity with magnitude greater than or equal to 2^{-126} but less than 2^{128}.

Example 1:

The number +3.5 can be represented in floating-point format as follows:

$+3.5 = 11.1_2 \times 2^0$
$= 1.11_2 \times 2^1$

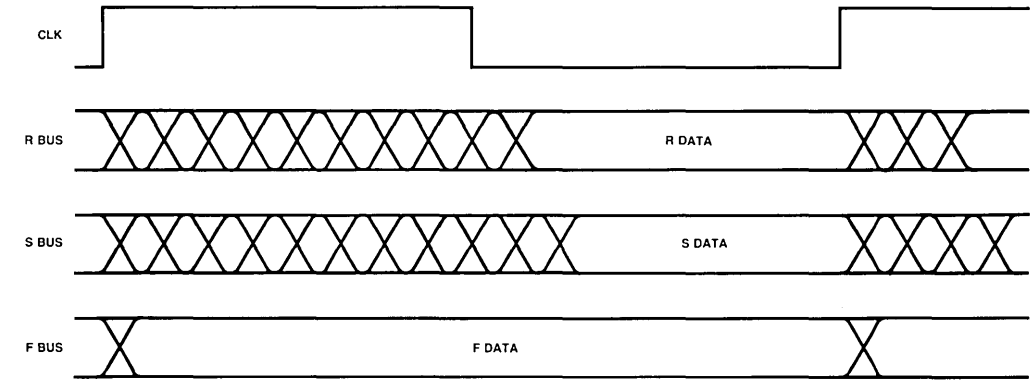
sign = 0

biased exponent = $1_{10}+127_{10} = 128_{10}$
$= 10000000_2$

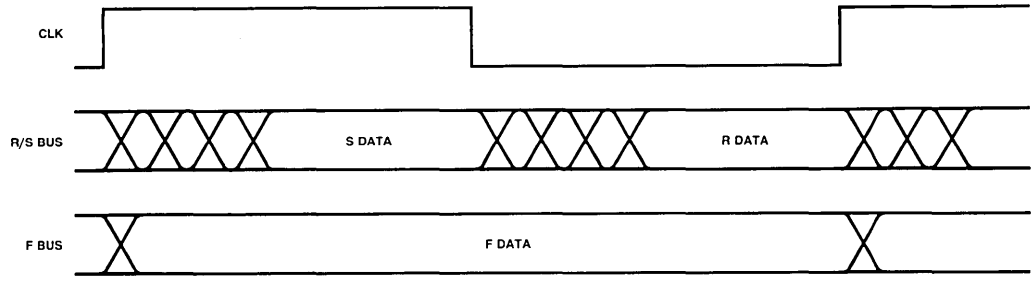
fraction = $11000000000000000000000_2$
(the leading 1 is implied in the format)

Concatenating these fields produces the floating-point word 40600000_{16}.

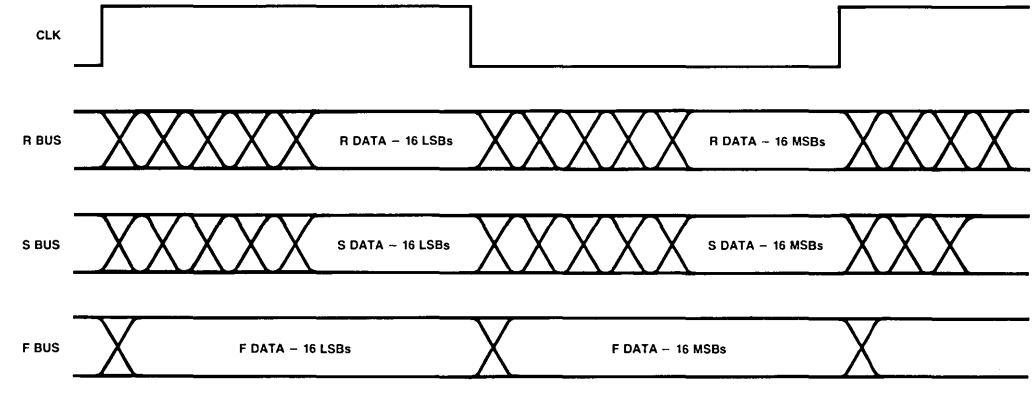
Figure 5. Typical Bus Timing for the I/O Modes, with FT_0 = LOW, FT_1 = LOW



a) 32-Bit, Two-Input-Bus Mode



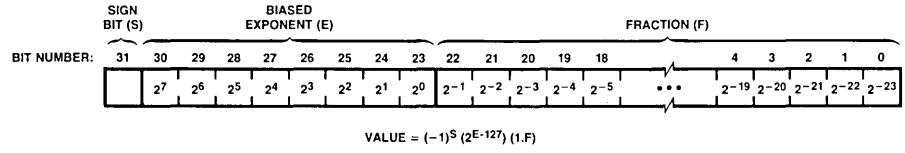
b) 32-Bit, Single-Input-Bus Mode



c) 16-Bit, Two-Input-Bus Mode

05621A-7

Figure 6. IEEE Mode Single-Precision Floating-Point Format



05621A-8

Example 2:

The number −11.375 can be represented in floating-point format as follows:

$-11.375 = -1011.011_2 \times 2^0$
$= -1.011011_2 \times 2^3$

sign = 1

biased exponent = $3_{10}+127_{10} = 130_{10}$
$= 10000010_2$

fraction = $01101100000000000000000_2$
(the leading 1 is implied in the format)

Concatenating these fields produces the floating-point word $C1360000_{16}$.

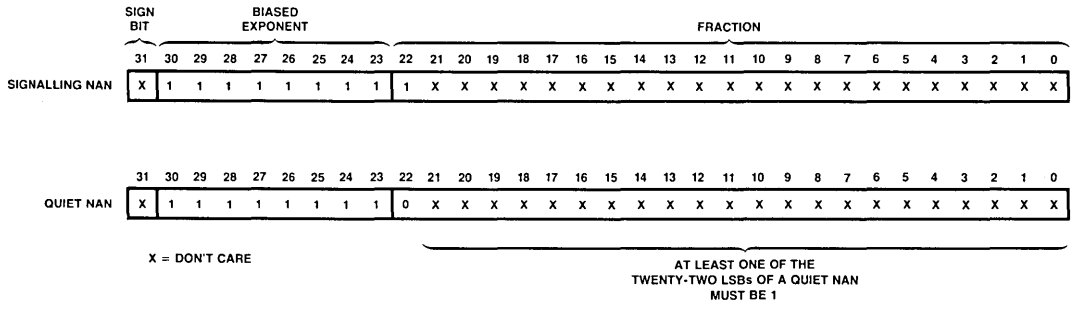
Infinity – Infinity can have either a positive or negative sign. The way in which infinities are interpreted is determined by the state of the projective/affine mode select, PROJ/$\overline{\text{AFF}}$.

Not-a-Number – A not-a-number, or NAN, does not represent a numeric value, but is interpreted as a signal or symbol. NANs are used to indicate invalid operations, and as a means of passing process status information through a series of calculations. NANs arise in two ways: they can be generated by the Am29325 to indicate that an invalid operation has taken place (e.g., infinity times zero), or they can be provided by the user as an input operand. There are two types of NANs: signalling and quiet. These NANs have the formats shown in Figure 7.

IEEE Mode Integer Format

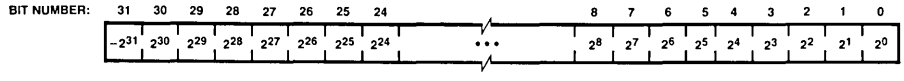
Integer numbers are represented as 32-bit, two's complement words; Figure 8 depicts the integer format. The integer word can represent a range of integer values from -2^{31} to $2^{31}-1$.

Figure 7. Signalling and Quiet NAN Formats



05621A-9

Figure 8. Thirty-Two-Bit Integer Format



05621A-10

Operations

All eight floating-point ALU operations discussed in the Functional Description section above can be performed in IEEE mode. Various exceptional aspects of the R PLUS S, R MINUS S, R TIMES S, 2 MINUS S, INT-TO-FP, and FP-TO-INT operations for this mode are described below. The IEEE-TO-DEC and DEC-TO-IEEE operations are discussed separately in the **IEEE-TO-DEC and DEC-TO-IEEE Operations** section on page 23.

Operations with NANs – NANs arise in two ways: they can be generated by the Am29325 to indicate that an invalid operation has taken place (e.g., infinity times zero), or they can be provided by the user as an input operand. There are two types of NANs: signalling and quiet. These NANs have the formats shown in Figure 7.

Signalling NANs set the invalid operation flag when they appear as an input operand to an operation. They are useful for indicating uninitialized variables, or for implementing user-designed extensions to the operations provided. The ALU never produces a signalling NAN as the final result of an operation.

Quiet NANs are generated for invalid operations. When they appear as an input operand, they are passed through most operations without setting the invalid flag, the floating-point-to-integer conversion operation being the exception.

The sign of any input operand NAN is ignored. All quiet NANs produced as the final result of an operation have a sign of 0.

When a NAN appears as an input operand, the final result of the operation is a quiet NAN that is created by taking the input NAN and forcing bit 22 LOW and bit 21 HIGH. If an operation has two NANs as input operands, the resulting quiet NAN is created using the NAN on the R port.

When a quiet NAN is produced as the final result of an invalid operation whose input operand or operands are not NANs, the resulting NAN will always have the value $7FA00000_{16}$.

The NAN flag will be HIGH whenever an operation produces a NAN as a final result.

Example 1:

Suppose the floating-point addition operation is performed with the following input operands:

R port: $3F800000_{16}$ $(1.0 \cdot 2^0)$
S port: $7FC12345_{16}$ (signalling NAN)

Result: The signalling NAN on the S port is converted to a quiet NAN by forcing bit 22 LOW and bit 21 HIGH. The operation's final result will be $7FA12345_{16}$. Since one of the two input operands is a signalling NAN, the invalid flag will be HIGH; the NAN flag will also be HIGH.

Example 2:

Suppose the floating-point multiplication operation is performed with the following input operands:

R port: $FFF11111_{16}$ (signalling NAN)
S port: $7FC22222_{16}$ (quiet NAN)

Result: Since both input operands are NANs, the NAN on the R port is chosen for output. In addition to forcing bit 22 LOW, the sign bit (bit 31) is set LOW (bit 21 is already HIGH, and need not be changed). The operation's final result will be $7FB11111_{16}$. Since one of the two input operands is a signalling NAN, the invalid flag is HIGH; the NAN flag will also be HIGH.

Example 3:

Suppose the floating-point subtraction operation is performed with the following input operands:

R port: $FF800001_{16}$ (quiet NAN)
S port: $7F800000_{16}$ $(+\infty)$

Result: To create the final result, the quiet NANs sign bit (bit 31) is forced LOW and bit 21 is forced HIGH (bit 22 is already LOW, and need not be changed). The final result will be $7FA00001_{16}$. The NAN flag will be HIGH.

Operations with Denormalized Numbers – The proposed IEEE standard incorporates denormalized numbers to allow a means of gradual underflow for operations that produce non-zero results too small to be expressed as a normalized floating-point number. The Am29325 does not support gradual underflow. If a floating-point operation produces a non-zero rounded result that is not large enough to be expressed as a normalized floating-point number, the final result will be a zero of the same sign; the inexact, underflow, and zero flags will be HIGH. If an input operand is a denormalized number, the floating-point ALU will assume that operand to be a zero of the same sign.

Operations Producing Overflows – If an operation has a finite input operand or operands, and if the operation produces a rounded result that is too large to fit in the destination format, that operation is said to have overflowed.

A floating-point overflow occurs if an R PLUS S, R MINUS S, R TIMES S, or 2 MINUS S operation with finite input operand(s) produces a result which, after rounding, has a magnitude greater than or equal to 2^{128}. Positive or negative infinity will appear as the final result if the rounded result is positive or negative, respectively, and the overflow and inexact flags will be HIGH.

Integer overflow occurs when the fixed-to-floating-point conversion operation attempts to convert a number which, after rounding, is greater than $2^{31}-1$ or less than -2^{31}. The final result will be quiet NAN $7FA00000_{16}$, and the invalid operation and NAN flags will be HIGH. Note that the overflow and inexact flags remain LOW for integer overflow.

Operations Producing Underflows – If an operation produces a floating-point rounded result having a magnitude too small to be expressed as a normalized floating-point number, but greater than zero, that operation is said to have underflowed. Underflow occurs when an R PLUS S, R MINUS S, or R TIMES S operation produces a result which, after rounding, has a magnitude in the range:

$$0 < \text{magnitude} < 2^{-126}.$$

In such cases, the final result will be $+0$ (00000000_{16}) if the rounded result is non-negative, and -0 (80000000_{16}) if the rounded result is negative. The underflow, inexact, and zero flags will be HIGH.

Underflow does not occur if the destination format is integer. If the infinitely precise result of a floating-point-to-integer conversion has a magnitude greater than 0 and less than 1 but the rounded result is 0, the underflow flag remains LOW.

Operations with Infinities – In most cases, positive and negative infinity are valid input arguments for the R PLUS S, R MINUS S, R TIMES S, and 2 MINUS S operations. Those cases for which infinities are not valid inputs for these operations are listed in the **IEEE Mode Invalid Operations Table** (see next page).

Infinities in IEEE mode can be handled either as projective or affine. The projective mode is selected when PROJ/$\overline{\text{AFF}}$ is HIGH;

the affine mode is selected when PROJ/$\overline{AFF}$ is LOW. The only differences between the modes that are relevant to Am29325 operation occur during the addition and subtraction of infinities:

Operation	Affine Mode	Projective Mode
$(+\infty)+(+\infty)$	Output $+\infty$	Output $7FA00000_{16}$ (quiet NAN), set invalid and NAN flags
$(-\infty)+(-\infty)$	Output $-\infty$	Output $7FA00000_{16}$ (quiet NAN), set invalid and NAN flags
$(+\infty)-(-\infty)$	Output $+\infty$	Output $7FA00000_{16}$ (quiet NAN), set invalid and NAN flags
$(-\infty)-(+\infty)$	Output $-\infty$	Output $7FA00000_{16}$ (quiet NAN), set invalid and NAN flags

If an R PLUS S, R MINUS S, R TIMES S, or 2 MINUS S operation has infinity as an input operand or operands, the final result, if valid, is presumed to be exact. For example, adding $+\infty$ and 2.0 will produce a final result of $+\infty$; since the result is considered exact, the inexact flag remains LOW.

Invalid Operations – If an input operand is invalid for the operation to be performed, that operation is considered invalid. When an invalid operation is performed, the floating-point ALU produces a quiet NAN as the final result, and the invalid operation flag goes HIGH. **The IEEE Mode Invalid Operations Table** lists the cases for which the invalid flag is HIGH in IEEE mode, and the final results produced for these operations.

IEEE MODE INVALID OPERATIONS TABLE

Operation	Input Operand	Final Result
R PLUS S	$(+\infty) + (-\infty)$ or $(-\infty) + (+\infty)$	$7FA00000_{16}$ (quiet NAN)
R PLUS S	$(+\infty) + (+\infty)$ or $(-\infty) + (-\infty)$ (Note 1)	$7FA00000_{16}$ (quiet NAN)
R MINUS S	$(+\infty) - (+\infty)$ or $(-\infty) - (-\infty)$	$7FA00000_{16}$ (quiet NAN)
R MINUS S	$(+\infty) - (-\infty)$ or $(-\infty) - (+\infty)$ (Note 1)	$7FA00000_{16}$ (quiet NAN)
R TIMES S	$(+0) \bullet (+\infty)$ or $(+0) \bullet (-\infty)$ or $(-0) \bullet (+\infty)$ or $(-0) \bullet (-\infty)$	$7FA00000_{16}$ (quiet NAN)
R PLUS S R MINUS S R TIMES S	R or S is a signalling NAN	(Note 2)
2 MINUS S	S is a signalling NAN	(Note 2)
FP-TO-INT	R is a signalling or quiet NAN	(Note 2)
FP-TO-INT	$R > 2^{31}-1$ or $R < -(2^{31})$	$7FA00000_{16}$ (quiet NAN)

Notes: 1. These cases are invalid in projective mode only.
2. Results for these operations are described in the **Operations with NANs** section.

The Sign Bit

For most floating-point operations, the sign bit of the final result is unambiguous, i.e., there is only one sign bit value that yields a numerically correct result. Operations that produce an infinitely precise result of zero, however, present a problem, as the IEEE floating-point format allows for representation of both +0 and −0. The following rules can be used to determine the signs of zero produced in such cases:

R PLUS S – The operations $+x + (-x)$ and $-x + (+x)$ produce a final result of zero; the sign of the zero is dependent on the rounding mode:

Rounding Mode	Sign of Final Result
Round to nearest	0
Round toward $-\infty$	1
Round toward $+\infty$	0
Round toward 0	0

The operation $+0 + (+0)$ produces a final result of +0; the operation $-0 + (-0)$ produces a final result of −0.

R MINUS S – The operations $+x - (+x)$ and $-x - (-x)$ produce a final result of zero; the sign of the zero is dependent on the rounding mode:

Rounding Mode	Sign of Result
Round to nearest	0
Round toward $-\infty$	1
Round toward $+\infty$	0
Round toward 0	0

The operation $+0 - (-0)$ produces a final result of +0; the operation $-0 - (+0)$ produces a final result of −0.

R TIMES S – The sign of any multiplication result other than a NAN is the exclusive-OR of the signs of the input operands. Therefore, if x is non-negative,

+0 times +x produces a final result of +0,
+0 times −x produces a final result of −0,
−0 times +x produces a final result of −0,
−0 times −x produces a final result of +0.

2 MINUS S – If S equals 2, the final result is −0 for the round toward $-\infty$ mode, and +0 for all other rounding modes.

Rounding

Rounding is performed whenever an operation produces an infinitely precise result that cannot be represented exactly in the destination format. For example, suppose a floating-point operation produces the infinitely precise result

$1.10101010101010101010101\backslash 01 \times 2^3$.

In this example, the fraction portion of the mantissa has twenty-five bits; the IEEE floating-point format can accommodate only twenty-three. The backslash (\) in the mantissa represents the boundary between the first twenty-three bits of the fraction and any remaining bits. Rounding is the process by which this result is approximated by a representation that fits the destination format.

There are four rounding modes in IEEE mode: round to nearest, round toward $+\infty$, round toward $-\infty$, and round toward 0. The rounding mode is chosen using the rounding mode select lines, RND_0 and RND_1. **The Rounding Mode Select Table** lists the select states needed to obtain the desired rounding mode.

ROUNDING MODE SELECT TABLE

RND_1	RND_0	Rounding Mode
0	0	Round to nearest
0	1	Round toward $-\infty$
1	0	Round toward $+\infty$
1	1	Round toward 0

Round to Nearest – In this rounding mode the infinitely precise result of an operation is rounded to the closest representation that fits in the destination format. If the infinitely precise result is exactly halfway between two representations, it is rounded to the representation having an LSB of zero. Rounding is performed both for floating-point and integer destination formats.

Figure 9 illustrates four examples of the round to nearest process for operations having a floating-point destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be represented exactly in the floating-point format.

Example 1:

In Figure 9(a), the infinitely precise result of an operation is:

$2^{20}+2^{-4}+2^{-5}$ = 1.00000000000000000000000 11 x 2^{20}.

The result is rounded to the closest representable floating-point value,

$2^{20}+2^{-3}$ = 1.00000000000000000000001 x 2^{20}.

Example 2:

In Figure 9(b), the infinitely precise result of an operation is:

$2^{20}-2^{-4}+2^{-8}$ = 1.11111111111111111111111 0001 x 2^{19}.

This result is rounded to the closest representable floating-point value,

$2^{20}-2^{-4}$ = 1.11111111111111111111111 x 2^{19}.

Example 3:

In Figure 9(c), the infinitely precise result of an operation is:

$-(2^{20}+2^{-3}+2^{-4})$
= $-1.00000000000000000000001\backslash 1 \times 2^{20}$.

This result is exactly halfway between two representable floating-point values. Accordingly, it is rounded to the closest representation with an LSB of zero, or

$-(2^{20}+2*2^{-3})$ = $-1.00000000000000000000010 \times 2^{20}$.

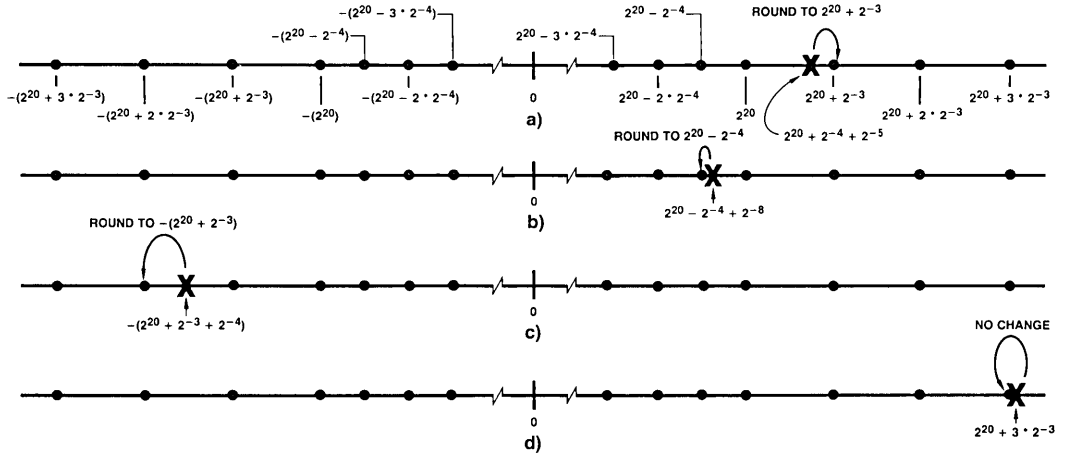
Example 4:

In Figure 9(d), the infinitely precise result of an operation is:

$2^{20}+3*2^{-3}$ = 1.00000000000000000000011 x 2^{20}.

This result can be represented exactly in the floating-point format, and is left unaltered by the rounding process.

Figure 9. Floating-Point Rounding Examples for Round to Nearest Mode



05621A-11

Figure 10 illustrates four examples of the round to nearest process for operations having an integer destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be represented exactly in the integer format.

Example 1:

In Figure 10(a), the infinitely precise result of an operation is:

$2^{10}-2^{-2}$ = 00...001111111111.11.

The result is rounded to the closest representable integer value, 2^{10} = 00...010000000000.

Example 2:

In Figure 10(b), the infinitely precise result of an operation is:

$2^{10}+2^{0}+2^{-3}$ = 00...010000000001.001.

This result is rounded to the closest representable floating-point value,

$2^{10}+2^{0}$ = 00...010000000001.

Example 3:

In Figure 10(c), the infinitely precise result of an operation is:

$-(2^{10}+2^{0}+2^{-1})$ = 11...101111111110.1.

This result is exactly halfway between two representable integer values. Accordingly, it is rounded to the closest representation with an LSB of zero, or

$-(2^{10}+2*2^{0})$ = 11...101111111110.

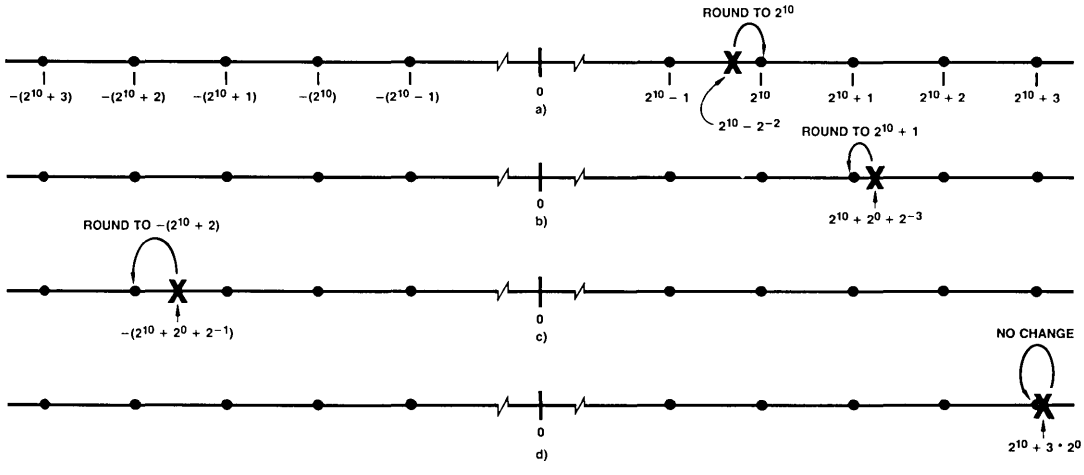
Example 4:

In Figure 10(d), the infinitely precise result of an operation is:

$2^{10}+3*2^{0}$ = 00...010000000011.

This result can be represented exactly in the integer format, and is left unaltered by the rounding process.

Figure 10. Integer Rounding Examples for Round to Nearest Mode



Round Toward $-\infty$ – In this rounding mode the result of an operation is rounded to the closest representation that is less than or equal to the infinitely precise result, and which fits the destination format. Rounding is performed both for floating-point and integer destination formats.

Figure 11 illustrates four examples of the round toward $-\infty$ process for operations having a floating-point destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be represented exactly in the floating-point format.

Example 1:

In Figure 11(a), the infinitely precise result of an operation is:

$2^{20}+2^{-4}+2^{-5} = 1.00000000000000000000000\backslash 11 \times 2^{20}$.

This result cannot be represented exactly in floating-point format, and is rounded to the next-smaller floating-point representation:

$2^{20} = 1.00000000000000000000000 \times 2^{20}$.

Example 2:

In Figure 11(b), the infinitely precise result of an operation is:

$2^{20}-2^{-4}+2^{-8} = 1.11111111111111111111111\backslash 0001 \times 2^{19}$.

This result cannot be represented exactly in floating-point format, and is rounded to the next-smaller floating-point representation:

$2^{20}-2^{-4} = 1.11111111111111111111111 \times 2^{19}$.

Example 3:

In Figure 11(c), the infinitely precise result of an operation is:

$-(2^{20}+2^{-3}+2^{-4})$
$= -1.00000000000000000000001\backslash 1 \times 2^{20}$.

This result cannot be represented exactly in floating-point format, and is rounded to the next-smaller floating-point representation:

$-(2^{20}+2\cdot 2^{-3}) = -1.00000000000000000000010 \times 2^{20}$.

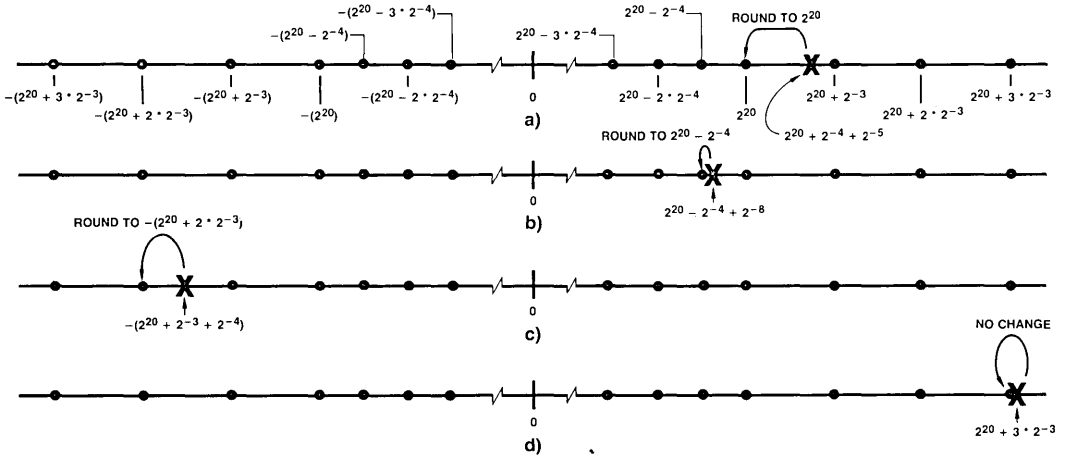
Example 4:

In Figure 11(d), the infinitely precise result of an operation is:

$2^{20}+3\cdot 2^{-3} = 1.00000000000000000000011 \times 2^{20}$.

This result can be represented exactly in the floating-point format, and is left unaltered by the rounding process.

Figure 11. Floating-Point Rounding Examples for Round Toward $-\infty$ Mode



05621A-13

Figure 12 illustrates four examples of the round toward $-\infty$ process for operations having an integer destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be exactly represented in the integer format.

Example 1:

In Figure 12(a), the infinitely precise result of an operation is:

$2^{10}-2^{-2}$ = 00...001111111111.11.

The result is rounded to the next-smaller representable integer value,

$2^{10}-2^{0}$ = 00...001111111111.

Example 2:

In Figure 12(b), the infinitely precise result of an operation is:

$2^{10}+2^{0}+2^{-3}$ = 00...010000000001.001.

This result is rounded to the next-smaller representable integer value,

$2^{10}+2^{0}$ = 00...010000000001.

Example 3:

In Figure 12(c), the infinitely precise result of an operation is:

$-(2^{10}+2^{0}+2^{-1})$ = 11...101111111110.1.

This result is rounded to the next-smaller representable integer value:

$-(2^{10}+2\cdot2^{0})$ = 11...101111111110.

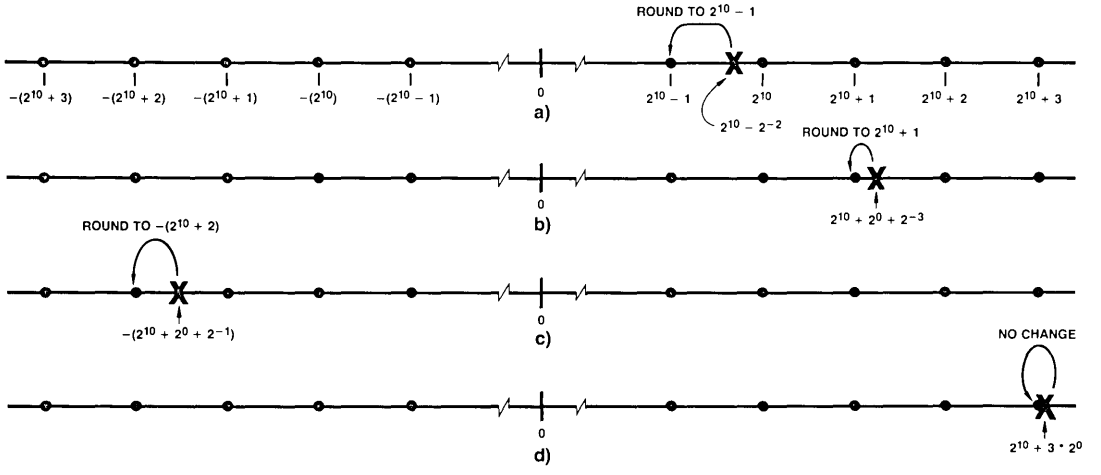
Example 4:

In Figure 12(d), the infinitely precise result of an operation is:

$2^{10}+3\cdot2^{0}$ = 00...010000000011.

This result can be represented exactly in the integer format, and is unaltered by the rounding process.

Figure 12. Integer Rounding Examples for Round Toward $-\infty$ Mode



05621A-14

Round Toward $+\infty$ – In this rounding mode the result of an operation is rounded to the closest representation that is greater than or equal to the infinitely precise result, and which fits the destination format. Rounding is performed both for floating-point and integer destination formats.

Figure 13 illustrates four examples of the round toward $+\infty$ process for operations having a floating-point destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be represented exactly in the floating-point format.

Example 1:

In Figure 13(a), the infinitely precise result of an operation is:

$2^{20}+2^{-4}+2^{-5} = 1.00000000000000000000000\backslash 11 \times 2^{20}$.

This result cannot be represented exactly in floating-point format, and is rounded to the next-larger floating-point representation:

$2^{20}+2^{-3} = 1.00000000000000000000001 \times 2^{20}$.

Example 2:

In Figure 13(b), the infinitely precise result of an operation is:

$2^{20}-2^{-4}+2^{-8} = 1.11111111111111111111111\backslash 0001 \times 2^{19}$.

This result cannot be represented exactly in floating-point format, and is rounded to the next-larger floating-point representation:

$2^{20} = 1.00000000000000000000000 \times 2^{20}$.

Example 3:

In Figure 13(c), the infinitely precise result of an operation is:

$-(2^{20}+2^{-3}+2^{-4})$
$= -1.00000000000000000000001\backslash 1 \times 2^{20}$.

This result cannot be represented exactly in floating-point format, and is rounded to the next-larger floating-point representation:

$-(2^{20}+2^{-3}) = -1.00000000000000000000001 \times 2^{20}$.

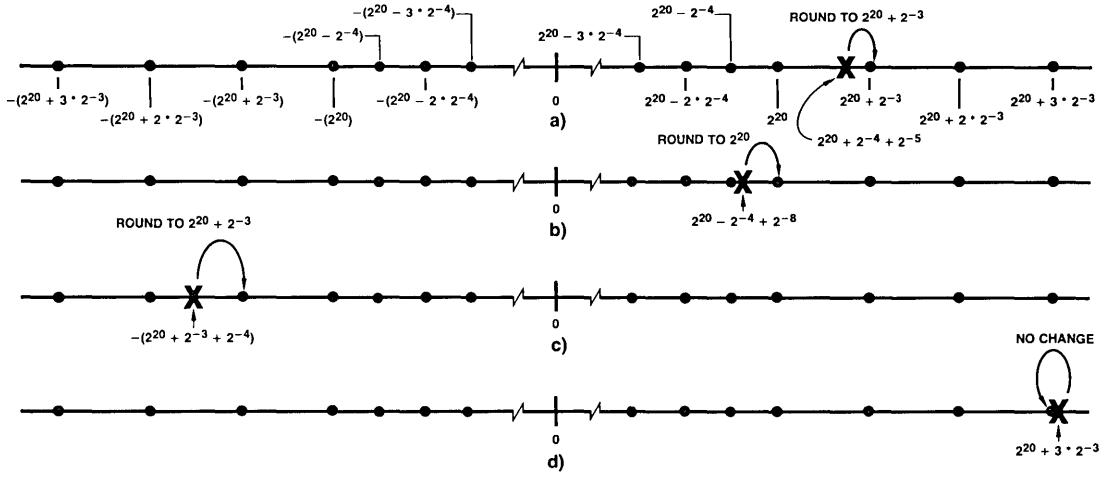
Example 4:

In Figure 13(d), the infinitely precise result of an operation is:

$2^{20}+3\cdot 2^{-3} = 1.00000000000000000000011 \times 2^{20}$.

This result can be represented exactly in the floating-point format – no rounding takes place.

Figure 13. Floating-Point Rounding Examples for Round Toward + ∞ Mode



05621A-15

Figure 14 illustrates four examples of the round toward $+\infty$ process for operations having an integer destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be exactly represented in the integer format.

Example 1:

In Figure 14(a), the infinitely precise result of an operation is:

$2^{10}-2^{-2}$ = 00...001111111111.11.

The result is rounded to the next-larger representable integer value,

2^{10} = 00...010000000000.

Example 2:

In Figure 14(b), the infinitely precise result of an operation is:

$2^{10}+2^{0}+2^{-3}$ = 00...010000000001.001.

This result is rounded to the next-larger representable integer value,

$2^{10}+2*2^{0}$ = 00...010000000010.

Example 3:

In Figure 14(c), the infinitely precise result of an operation is:

$-(2^{10}+2^{0}+2^{-1})$ = 11...101111111110.1

This result is rounded to the next-larger representable integer value:

$-(2^{10}+2^{0})$ = 11...1011111111110.

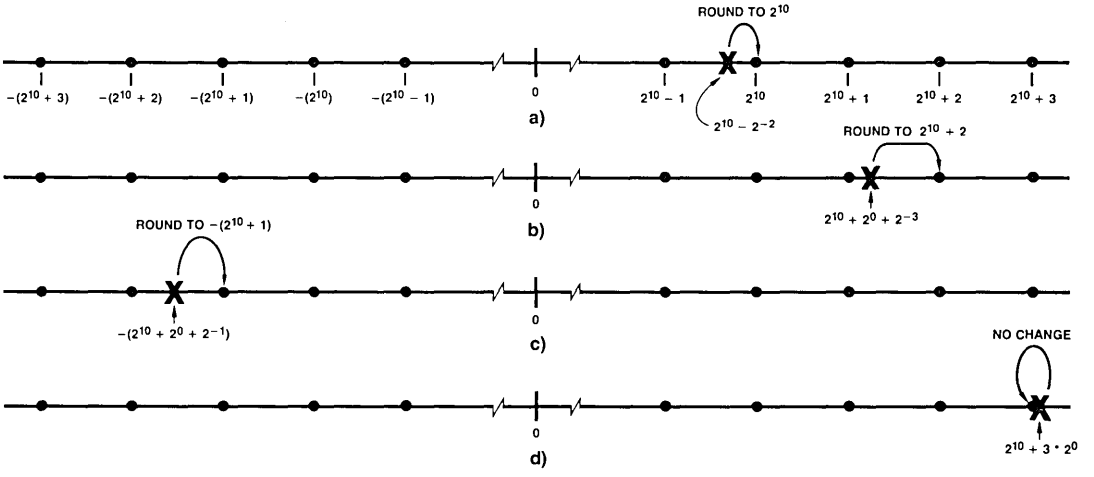
Example 4:

In Figure 14(d), the infinitely precise result of an operation is:

$2^{10}+3*2^{0}$ = 00...010000000011.

This result can be represented exactly in the integer format – no rounding takes place.

Figure 14. Integer Rounding Examples for Round Toward + ∞ Mode



05621A-16

Round Toward 0 – In this rounding mode the result of an operation is rounded to the closest representation whose magnitude is less than or equal to the infinitely precise result, and which fits the destination format. Rounding is performed both for floating-point and integer destination formats.

Figure 15 illustrates four examples of the round toward 0 process for operations having a floating-point destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be represented exactly in the floating-point format.

Example 1:

In Figure 15(a), the infinitely precise result of an operation is:

$2^{20}+2^{-4}+2^{-5} = 1.00000000000000000000000\backslash 11 \times 2^{20}$.

This result cannot be represented exactly in floating-point format, and is rounded to:

$2^{20} = 1.00000000000000000000000 \times 2^{20}$.

Example 2:

In Figure 15(b), the infinitely precise result of an operation is:

$2^{20}-2^{-4}+2^{-8} = 1.11111111111111111111111\backslash 001 \times 2^{19}$.

This result cannot be represented exactly in floating-point format, and is rounded to:

$2^{20}-2^{-4} = 1.11111111111111111111111 \times 2^{19}$.

Example 3:

In Figure 15(c), the infinitely precise result of an operation is:

$-(2^{20}+2^{-3}+2^{-4})$
$= -1.00000000000000000000001\backslash 1 \times 2^{20}$.

This result cannot be represented exactly in floating-point format, and is rounded to:

$-(2^{20}+2^{-3}) = -1.00000000000000000000001 \times 2^{20}$.

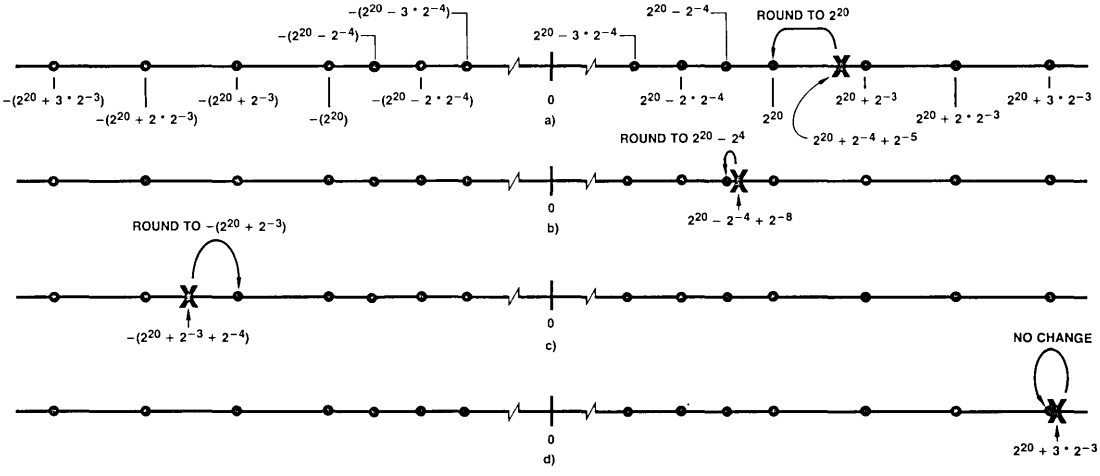
Example 4:

In Figure 15(d), the infinitely precise result of an operation is:

$2^{20}+3*2^{-3} = 1.00000000000000000000011 \times 2^{20}$.

This result can be represented exactly in the floating-point format, and is unaffected by the rounding process.

Figure 15. Floating-Point Rounding Examples for Round Toward 0 Mode



05621A-17

Figure 16 illustrates four examples of the round toward 0 process for operations having an integer destination format. The infinitely precise result of an operation is represented by an X on the number line; the black dots on the number line indicate those values that can be exactly represented in the integer format.

Example 1:

In Figure 16(a), the infinitely precise result of an operation is:

$2^{10}-2^{-2}$ = 00...001111111111.11.

The result is rounded to:

$2^{10}-2^{0}$ = 00...001111111111.

Example 2:

In Figure 16(b), the infinitely precise result of an operation is:

$2^{10}+2^{0}+2^{-3}$ = 00...010000000001.001.

The result is rounded to:

$2^{10}+2^{0}$ = 00...010000000001.

Example 3:

In Figure 16(c), the infinitely precise result of an operation is:

$-(2^{10}+2^{0}+2^{-1})$ = 11...101111111110.1.

This result is rounded to:

$-(2^{10}+2^{0})$ = 11...101111111111.

Example 4:

In Figure 16(d), the infinitely precise result of an operation is:

$2^{10}+3*2^{0}$ = 00...010000000011.

This result can be represented exactly in the integer format, and is unaffected by the rounding process.

Flag Operation

The Am29325 generates six status flags to monitor floating-point processor operation. The following is a summary of flag conventions in IEEE mode:

Invalid Operation Flag – The invalid operation flag is HIGH when an input operand is invalid for the operation to be performed. **The IEEE Mode Invalid Operations Table** on page 12 lists the cases for which the invalid operation flag is HIGH in IEEE mode, and the corresponding final result. In cases where the invalid operation flag is HIGH, the overflow, underflow, zero, and inexact flags are LOW; the NAN flag will be HIGH.

Overflow Flag – The overflow flag is HIGH if an R PLUS S, R MINUS S, R TIMES S, or 2 MINUS S operation with finite input operand(s) produces a result which, after rounding, has a magnitude greater than or equal to 2^{128}. The final result will be $+\infty$ or $-\infty$.

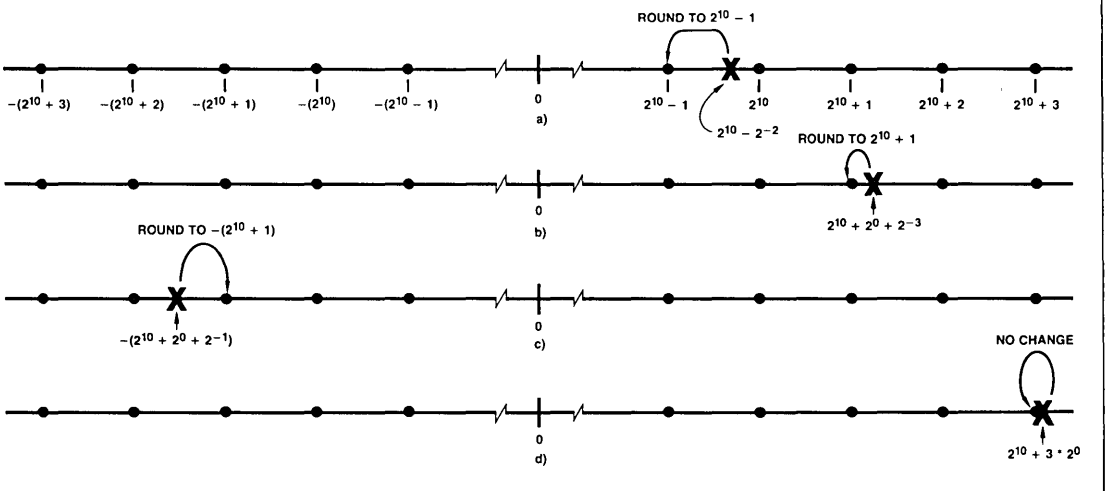
Underflow Flag – The underflow flag is HIGH if an R PLUS S, R MINUS S, or R TIMES S operation produces a result which, after rounding, has a magnitude in the range:

$$0 < \text{magnitude} < 2^{-126}.$$

The final result will be +0 (00000000_{16}) if the rounded result is non-negative, and −0 (80000000_{16}) if the rounded result is negative.

Inexact Flag – The inexact flag is HIGH if the final result of an R PLUS S, R MINUS S, R TIMES S, 2 MINUS S, INT-TO-FP, or FP-TO-INT operation is not equal to the infinitely precise result. Note that if the underflow or overflow flag is HIGH, the inexact flag will also be HIGH.

Figure 16. Integer Rounding Examples for Round Toward 0 Mode



05621A-18

Zero Flag – The zero flag is HIGH if the final result of an operation is zero. For operations producing an IEEE floating-point number, the flag accompanies outputs +0 (00000000_{16}) and −0 (80000000_{16}). For operations producing an integer, the flag accompanies the output 0 (00000000_{16}).

NAN Flag – The NAN flag is HIGH if an R PLUS S, R MINUS S, R TIMES S, 2 MINUS S, or FP-TO-INT operation produces a NAN as a final result.

OPERATION IN DEC MODE

When input signal IEEE/$\overline{\text{DEC}}$ is LOW, the DEC mode of operation is selected. In this mode the Am29325 uses the single-precision floating-point format (floating F) set forth in Digital Equipment Corporation's VAX Architecture Manual. In addition, the DEC mode complies with most other aspects of single-precision floating-point operation outlined in the manual – differences are discussed in **Appendix B.**

DEC Floating-Point Format

The DEC single-precision floating-point word is thirty-two bits wide, and is arranged in the format shown in Figure 17. The floating-point word is divided into three fields: a single-bit sign, an eight-bit biased exponent, and a 23-bit fraction.

The sign bit indicates the sign of the floating-point number's value. Non-negative values have a sign of 0, negative values a sign of 1.

The biased exponent is an eight-bit unsigned integer field representing a multiplicative factor of some power of two. The bias value is 128. If, for example, the multiplicative factor for a floating-point number is to be 2^a, the value of the biased exponent would be a+128; a is called the true exponent.

The fraction is a 23-bit unsigned fractional field containing the 23 least-significant bits of the floating-point number's 24-bit mantissa. The weight of this field's most significant bit is 2^{-2}; the weight of the least-significant bit is 2^{-24}.

A floating-point number is evaluated or interpreted per the following conventions:

let s = sign bit
e = biased exponent
f = fraction

if e = 0 and s = 0 . . . value = 0

if e = 0 and s = 1 . . . value = DEC reserved operand

if $0 < e \leq 255$ . . value = $(-1)^{s} \cdot (2^{e-128}) \cdot (.1f)$ (normalized number)

Zero – The value zero always has a sign of zero.

DEC Reserved Operand – A DEC reserved operand does not represent a numeric value, but is interpreted as a signal or symbol. DEC reserved operands are used to indicate invalid operations and operations whose results have overflowed the destination format. They may also be used to pass symbolic information from one calculation to another.

Normalized Number – A normalized number represents a quantity with magnitude greater than or equal to 2^{-128} but less than 2^{127}.

Example 1:

The number +3.5 can be represented in floating-point format as follows:

$+3.5 = 11.1_2 \times 2^0$
$= .111_2 \times 2^2$

sign = 0

biased exponent = $2_{10} + 128_{10} = 130_{10}$
$= 10000010_2$

fraction = $11000000000000000000000_2$
(the leading 1 is implied in the format)

Concatenating these fields produces the floating-point word 41600000_{16}.

Example 2:

The number −11.375 can be represented in floating-point format as follows:

$-11.375 = -1011.011_2 \times 2^0$
$= -.1011011_2 \times 2^4$

sign = 1

biased exponent = $4_{10} + 128_{10} = 132_{10}$
$= 10000100_2$

fraction = $01101100000000000000000_2$
(the leading 1 is implied in the format)

Concatenating these fields produces the floating-point word $C2360000_{16}$.

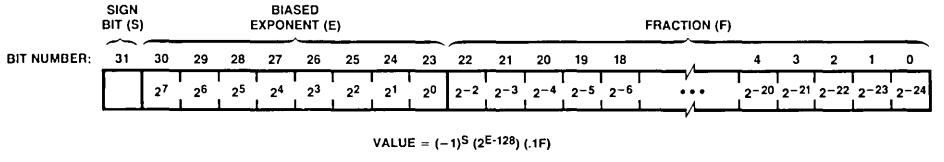
DEC Mode Integer Format

DEC mode integer format is identical to that of the IEEE mode. Integer numbers are represented as 32-bit, two's complement words; Figure 7 depicts the integer format. The integer word can represent a range of integer values from -2^{31} to $2^{31}-1$.

Operations

All eight floating-point ALU operations discussed in the General Description section can be performed in DEC mode.

Figure 17. DEC-Mode Floating-Point Format



05621A-19

Various exceptional aspects of the R PLUS S, R MINUS S, R TIMES S, 2 MINUS S, INT-TO-FP, and FP-TO-INT operations for this mode are described below. The IEEE-TO-DEC and DEC-TO-IEEE operations are discussed separately in the **IEEE-TO-DEC and DEC-TO-IEEE Operations** section on page 23.

Operations with DEC Reserved Operands – DEC reserved operands arise in two ways: they can be generated by the Am29325 to indicate that an invalid operation or floating-point overflow has taken place, or they can be provided by the user as an input operand.

When a DEC reserved operand appears as an input operand, the final result of the operation is the same DEC reserved operand. If an operation has two DEC reserved operands as inputs, the DEC reserved operand on the R port becomes the final result.

The NAN flag will be HIGH whenever an operation produces a DEC reserved operand as a final result.

Example 1:

Suppose the floating-point addition operation is performed with the following input operands:

R port: 40800000_{16} $(0.1 \cdot 2^1)$
S port: 80012345_{16} (DEC reserved operand)

Result: This operation produces the DEC reserved operand on the S port, 80012345_{16}, as the final result. The NAN flag will be HIGH.

Example 2:

Suppose the floating-point multiplication operation is performed with the following input operands:

R port: 80765432_{16} (DEC reserved operand)
S port: 80000001_{16} (DEC reserved operand)

Result: Since both input operands are DEC reserved operands, the operand on the R port, 80765432_{16}, is the final result of the operation. The NAN flag will be HIGH.

Operations Producing Overflows – If an operation produces a rounded result that is too large to fit in the destination format, that operation is said to have overflowed.

A floating-point overflow occurs if a R PLUS S, R MINUS S, R TIMES S, or 2 MINUS S operation with finite input operand(s) produces a result which, after rounding, has a magnitude greater than or equal to 2^{127}. The final result in such cases will be DEC reserved operand 80000000_{16}; the overflow, inexact, and NAN flags will be HIGH.

Integer overflow occurs when the fixed-to-floating-point conversion operation attempts to convert to integer a floating-point number which, after rounding, is greater than $2^{31}-1$ or less than -2^{31}. The final result in such cases will be DEC reserved operand 80000000_{16}; the invalid operation flag will be HIGH. Note that the overflow and inexact flags remain LOW for integer overflow.

Operations Producing Underflows – If an operation produces a floating-point result which, after rounding, has a magnitude too small to be expressed as a normalized floating-point number, but greater than zero, that operation is said to have underflowed. Underflow occurs when an R PLUS S, R MINUS S, or R TIMES S operation produces a result which, after rounding, has magnitude:

$$0 < \text{magnitude} < 2^{-128}.$$

The final result in such cases will be 0 (00000000_{16}). The underflow, inexact, and zero flags will be HIGH.

Underflow does not occur if the destination format is integer. If the infinitely precise result of a floating-point-to-integer conversion has a magnitude greater than 0 and less than 1, but the rounded result is 0, the underflow flag remains LOW.

Invalid Operations – If an input operand is invalid for the operation to be performed, that operation is considered invalid. In DEC mode, there are only two invalid operations:

- Performing a floating-point-to-integer conversion on a value too large to be expressed as a 32-bit integer. In this case the final result will be DEC reserved operand 80000000_{16}, and the invalid operation and NAN flags will be HIGH.
- Performing a floating-point-to-integer conversion on a DEC reserved operand. In this case the final result will be the input DEC reserved operand, and the invalid operation and NAN flags will be HIGH.

Sign Bit

For all operations producing a DEC floating-point result, the sign bit of the final result is unambiguous, i.e., there is only one sign bit value that yields a numerically correct result.

Rounding

There are four rounding modes for DEC operation: round to nearest, round toward $+\infty$, round toward $-\infty$, and round toward 0. The round toward $+\infty$, round toward $-\infty$, and round toward 0 modes are performed in a manner identical to that for IEEE operation; refer to the **Rounding** section under **Operation in IEEE Mode** on page 12. The round to nearest mode is similar to that for IEEE operation, but differs in one respect: for the case in which the infinitely-precise result of an operation is exactly halfway between two representable values, DEC round to nearest mode rounds to the value with the larger magnitude, rather than to the value whose LSB is 0.

Flag Operation

The Am29325 generates six status flags to monitor floating-point processor operation. The following is a summary of flag operation in DEC mode:

Invalid Operation Flag – The invalid operation flag is HIGH if the FP-TO-INT operation is performed on a floating-point number too large to be converted to an integer, or on a DEC reserved operand. If the FP-TO-INT operation is performed on a floating-point number too large to be converted to integer, the final result is the DEC reserved operand 80000000_{16}. If the FP-TO-INT operation is performed on a DEC reserved operand, that operand becomes the final result.

Overflow Flag – The overflow flag is HIGH if an R PLUS S, R MINUS S, R TIMES S, or 2 MINUS S operation produces a result which, after rounding, has a magnitude greater than or equal to 2^{127}. The final result will be the DEC reserved operand 80000000_{16}.

Underflow Flag – The underflow flag is HIGH if an R PLUS S, R MINUS S, or R TIMES S operation produces a result which, after rounding, has a magnitude in the range:

$$0 < \text{magnitude} < 2^{-128}.$$

The final result will be 0 (00000000_{16}) in such cases.

Inexact Flag – The inexact flag is HIGH if the final result of an R PLUS S, R MINUS S, R TIMES S, 2 MINUS S, INT-TO-FP, or FP-TO-INT operation is not equal to the infinitely precise result. Note that if the underflow or overflow flag is HIGH, the inexact flag will also be HIGH.

Zero Flag – The zero flag is HIGH if the final result of an operation is zero. For operations producing an integer or a DEC floating-point number, the flag accompanies the output 0 (00000000_{16}). (It should be noted that any operation producing a floating-point 0 in DEC mode will output 00000000_{16}.)

NAN Flag – The NAN flag is HIGH if an R PLUS S, R MINUS S, R TIMES S, 2 MINUS S, or FP-TO-INT operation produces a DEC reserved operand as the final result.

IEEE-TO-DEC AND DEC-TO-IEEE OPERATIONS

The IEEE-TO-DEC and DEC-TO-IEEE operations are used to convert floating-point numbers between the IEEE and DEC formats. Both operations work in a manner independent of the IEEE/$\overline{\text{DEC}}$ mode control.

IEEE-TO-DEC Conversion

This operation converts an IEEE floating-point number to DEC floating-point format. Most conversions are exact; in no case does the round mode have any effect on the final result. There are, however, a few exceptional cases:

a.) If the IEEE floating-point input has a magnitude greater than or equal to 2^{127}, it is too large to be represented by a DEC floating-point number. The final result will be the DEC reserved operand 80000000_{16}; the overflow, inexact, and NAN flags will be HIGH.
b.) If the IEEE floating-point input is a NAN, the final result will be the DEC reserved operand 80000000_{16}; the invalid and NAN flags will be HIGH.
c.) If the IEEE floating-point input is a denormalized number, the final result will be a DEC 0 (00000000_{16}); the zero flag will be HIGH.
d.) If the IEEE floating-point input is +0 or −0, the final result will be a DEC 0 (00000000_{16}); the zero flag will be HIGH.

DEC-TO-IEEE Conversion

This operation converts a DEC floating-point number to IEEE floating-point format. Most conversions are exact; in no case does the round mode have any effect on the final result. There are, however, a few exceptional cases:

a.) If the DEC floating-point input is not 0, but has a magnitude less than 2^{-126}, it is too small to be expressed as a normalized IEEE floating-point number. The final result will be an IEEE floating-point 0 having the same sign as the input (00000000_{16} for positive inputs and 80000000_{16} for negative inputs); the underflow, inexact, and zero flags will be HIGH.
b.) If the DEC floating-point input is a DEC reserved operand, the final result will be quiet NAN $7FA00000_{16}$; the invalid operation and NAN flags will be HIGH.
c.) If the DEC floating-point input is 0, the final result will be IEEE floating-point +0 (00000000_{16}); the zero flag will be HIGH.

APPENDIX A:

Differences Between the IEEE Proposed Standard for Binary Floating-Point Arithmetic and the Am29325's IEEE Mode

When operated in IEEE mode, the Am29325 High-speed Floating-Point Processor complies with the single-precision portion of the IEEE Proposed Standard for Binary Floating-Point Arithmetic (P754, draft 10.0) in most respects. There are, however, several differences:

Denormalized Numbers

The Am29325 does not handle denormalized numbers. A denormalized input will be converted to a zero of the same sign before the specified operation takes place. The operation proceeds in exactly the same manner as if the input were +0 or −0, producing the same numerical result and flags.

If the result of an operation, after rounding, has a magnitude smaller than 2^{-126}, the result is replaced by a zero of the same sign.

Representation of Overflows

In some rounding modes, the proposed IEEE standard requires that overflows be represented as the format's most positive or most negative finite number. In particular:

- When rounding toward 0, all overflows should produce a result of the largest representable finite number with the sign of the intermediate result.
- When rounding toward $-\infty$, all positive overflows should produce a result of the largest representable positive finite number.
- When rounding toward $+\infty$, all negative overflows should produce a result of the largest representable negative finite number.

The Am29325, however, always represents positive overflows as $+\infty$ and negative overflows as $-\infty$, regardless of rounding mode.

Projective Mode

The proposed IEEE standard provides only for an affine mode to control the handling of infinities. The Am29325 provides both affine and projective modes; the desired mode can be selected by the user.

Traps

The proposed IEEE standard stipulates that the user be able to request a trap on any exception. The Am2935 does not support trapped operation, and behaves as if traps are disabled.

Resetting of Flags

The proposed IEEE standard states that once an exception flag has been set, it is reset only at the user's request. The Am29325's flags, however, reflect the status of the most recent operation.

Generation of the Underflow Flag

The proposed IEEE standard suggests several possible criteria for determining if underflow occurs. These criteria generate underflow flags that differ in subtle ways. The underflow criteria chosen for the Am29325 stipulate that underflow occurs if:

a) the rounded result of an operation has a magnitude in the range:

$$0 < \text{magnitude} < 2^{-126},$$

and

b) the final result is not equal to the infinitely precise result.

Since the Am29325 never produces a denormalized number as the final result of a calculation, condition (b) is true whenever (a) is true. Note, then, that the operation of the Am29325's underflow flag is somewhat different than that of an "IEEE standard" system using the same underflow criteria. For example, if an operation should produce an infinitely precise result that is exactly 2^{-127}, an "IEEE standard" system would produce that value as the final result, expressed as a denormalized number. Since that system's final result is exact, the underflow flag would remain LOW. The Am29325, on the other hand, would output zero; since its final result is not exact, the underflow flag would be HIGH.

APPENDIX B:

Differences Between DEC VAX and Am29325 DEC Mode

Operation in DEC mode complies with most aspects of single-precision floating-point operation outlined in the Digital Equipment Corporation's VAX Architecture Manual. However, there are some differences that should be noted:

Format

The Am29325's DEC format is:

sign – bit 31
exponent – bits 30 – 23
mantissa – 22 – 0

The VAX format is:

sign – bit 15
exponent – 14 – 7
mantissa – bits 6 – 0, bits 31 – 16.

In both cases, fields are listed from MSB to LSB, with bit 31 the MSB of the 32-bit word. The Am29325's DEC format can be converted to VAX format by swapping the 16 LSBs and 16 MSBs of the 32-bit word.

Flags vs. Exceptions

In DEC VAX operation, certain unusual conditions arising during system operation may incur an exception, or an indication to the operating system that special handling is needed.

The VAX recognizes a number of arithmetic exceptions. The following exceptions are relevant to the operations supported by the Am29325:

Integer overflow trap – indicates that the last operation produced an integer overflow. The LSBs of the correct result are stored in the destination operand.

Floating-point overflow trap/fault – indicates that the last operation produced, after normalization and rounding, a floating-point number with magnitude greater than or equal to 2^{127}. A trap replaces the destination operand with the DEC reserved operand 80000000_{16}; a fault leaves the destination operand unchanged.

Floating-point underflow trap/fault – indicates that the last operation produced, after normalization and rounding, a floating-point number with magnitude less than 2^{-128}. A trap replaces the destination operand with zero; a fault leaves the destination operand unchanged.

Reserved operand fault – indicates that the last operation had a reserved operand as an input. The destination operand is unchanged.

The Am29325 does not directly support DEC traps and faults. Rather, it indicates unusual conditions by setting one or more of the six status flags HIGH. Table d2 describes flag operation in DEC mode.

Integer Overflow

In cases of integer overflow, the VAX signals the integer overflow trap and stores the LSBs of the correct result. The Am29325 sets the invalid operation flag and outputs the DEC reserved operand 80000000_{16}.

Floating-Point Underflow/Overflow Operation

The VAX Architecture Manual specifies the action to be taken on the destination operand when floating-point underflow or overflow is encountered. The Am29325 has no immediate control over this destination operand, as it resides somewhere off-chip, either in a register or memory location. This isn't so much a difference between the VAX specification and Am29325 operation as it is a difference in scope.

The Am29325 responds to floating-point underflow by producing a final result of 0 (00000000_{16}); the underflow, inexact, and zero flags will be HIGH. It responds to floating-point overflow by producing the DEC reserved operand 80000000_{16} as the final result; the overflow, inexact, and NAN flags will be HIGH.

Handling of DEC Reserved Operands

If an operation has a DEC reserved operand as an input, the Am29325 will produce that operand as the final result. If an operation has two input arguments and both are DEC reserved operands, the operand on port R becomes the final result. For the VAX, operations with a DEC reserved operand input or inputs do not modify the destination operand. As mentioned above, control of the destination operand is beyond the scope of the Am29325's operation.

Inexact Flag

The Am29325 provides an inexact flag to indicate that the final result produced by an operation is not equal to the infinitely precise result. The VAX does not provide this flag.

APPENDIX C:

Performing Floating-Point Division on the Am29325

While the Am29325 does not have a floating-point division instruction, it can be used to evaluate reciprocals. The division:

$$C = A/B$$

can then be performed by evaluating:

$$C = A \cdot (1/B).$$

Only a modest amount of external hardware is needed to implement the reciprocal function.

The technique for calculating reciprocals is based on the Newton-Raphson method for obtaining the roots of an equation. The roots of equation:

$$F(x) = 0$$

can be found by iteratively evaluating the equation

$$x_{i+1} = x_i - F(x_i)/F'(x_i).$$

The process begins by making a guess as to the value of x_i, and using this guess or "seed" value to perform the first iteration. Iterations are continued until the root is evaluated to the desired accuracy. The number of iterations needed to achieve a given accuracy depends both on the accuracy of the seed value and the nature of F(x).

Now consider the equation

$$F(x) = (1/x) - B.$$

The root of F(x) is 1/B. The reciprocal of B, then, can be found by using the Newton-Raphson method to find the root of F(x). The iterative equation for finding the root is

$$\begin{aligned} x_{i+1} &= x_i - F(x_i)/F'(x_i) \\ &= x_i - (1/x_i - B)/-(x_i)^{-2} \\ &= x_i\,(2 - B \cdot x_i). \end{aligned}$$

It can be shown that, in order for this iterative equation to converge, the seed value x_0 must fall in the range

$$0 < x_0 < 2/B \quad \text{if } B > 0$$

or $$2/B < x_0 < 0 \quad \text{if } B < 0.$$

For example, if the reciprocal of 3 is to be evaluated, the seed value must be between 0 and 2/3.

The error of x_i reduces quadratically; that is, if the error of x_i is e, the error is reduced to order e^2 by the next iteration. The number of bits of accuracy in the result, then, roughly doubles after every iteration. While this is only an approximation of the actual error produced, it is a handy rule-of-thumb for determining the number of iterations needed to produce a result of a certain accuracy, given the accuracy of the seed.

Example 1:

Find the reciprocal of 7.25.

Solution:

The seed value must fall in the range

$$0 < x_0 < 2/7.25$$

or $$0 < x_0 < .275862.$$

Suppose x_0 is chosen to be .1

Iteration 1: $x_1 = x_0\,(2 - B \cdot x_0)$
$= .1(2-(7.25)\,(.1))$
$= .1275$

Iteration 2: $x_2 = x_1\,(2 - B \cdot x_1)$
$= .1275(2-(7.25)\,(.1275))$
$= .1371421875$

Iteration 3: $x_3 = x_2(2 - B \cdot x_2)$
$= .1371421875 \cdot$
$(2-(7.25)\,(.1371421875))$
$= .1379265230$

The actual value of 1/7.25, to ten decimal places, is .1379310345.

The error after each iteration is:

Iteration	x_i	Error to Ten Places
0	.1	−0.0379310345
1	.1275	−0.0104310345
2	.1371421875	−0.0007888470
3	.1379265230	−0.0000045115

Example 2:

Find the reciprocal of −.3.

Solution:

The seed value must fall in the range

$$2/(-.3) < x_0 < 0$$

or $$-6.66 < x_0 < 0.$$

Suppose x_0 is chosen to be −2.0.

Iteration 1: $x_1 = x_0(2 - B \cdot x_0)$
$= -2.0(2-(-.3)\,(-2.0))$
$= -2.8$

Iteration 2: $x_2 = x_1\,(2 - B \cdot x_1)$
$= -2.8(2-(-.3)\,(-2.8))$
$= -3.248$

Iteration 3: $x_3 = x_2(2 - B \cdot x_2)$
$= -3.248(2-(-.3)\,(-3.248))$
$= -3.3311488$

Iteration 4: $x_4 = x_3(2 - B \cdot x_3)$
$= -3.3311488 \cdot$
$(2-(-.3)\,(-3.3311488))$
$= -3.333331902$

The actual value of 1/(−.3), to ten decimal places, is −3.333333333.

The error after each iteration is:

i	x_i	Error to Ten Places
0	−2.0	1.333333333
1	−2.8	0.533333333
2	−3.248	0.085333333
3	−3.3311488	0.002184533
4	−3.333331902	0.000001431

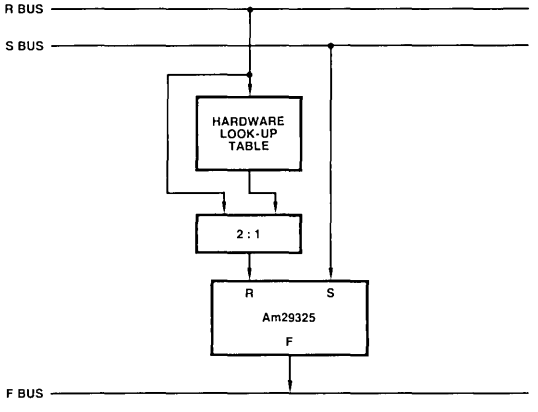
In order to implement the Newton-Raphson method on the Am29325, some means is needed to generate the seed used in the first iteration. One approach is to place a hardware seed look-up table between the R bus and the Am29325; see Table c1. A more detailed diagram of the look-up table appears in Figure c2.

TABLE c1. CONTENTS OF THE SEED EXPONENT PROM

DEC		IEEE	
Address (16)	Data (16)	Address (16)	Data (16)
000	(Note 1)	100	FD
001	(Note 1)	101	FC
002	FF	102	FB
003	FE	103	FA
004	FD	104	F9
005	FC	105	F8
006	FB	106	F7
007	FA	107	F6
008	F9	108	F5
009	F8	109	F4
00A	F7	10A	F3
00B	F6	10B	F2
00C	F5	10C	F1
00D	F4	10D	F0
00E	F3	10E	EF
00F	F2	10F	EE
010	F1	110	ED
011	F0	111	EC
012	EF	112	EB
.	.	.	.
.	.	.	.
.	.	.	.
0EE	13	1EE	0F
0EF	12	1EF	0E
0F0	11	1F0	0D
0F1	10	1F1	0C
0F2	0F	1F2	0B
0F3	0E	1F3	0A
0F4	0D	1F4	09
0F5	0C	1F5	08
0F6	0B	1F6	07
0F7	0A	1F7	06
0F8	09	1F8	05
0F9	08	1F9	04
0FA	07	1FA	03
0FB	06	1FB	02
0FC	05	1FC	01
0FD	04	1FD	(Note 2)
0FE	03	1FE	(Note 2)
0FF	02	1FF	(Note 2)

Notes: 1. The reciprocals of these numbers are too large to be represented in DEC format.
2. The reciprocals of these numbers are too small to be represented in normalized IEEE format.

Figure c1. Adding a Hardware Look-Up Table to the Am29325



05621A-20

The look-up table has two sections: a biased exponent look-up PROM and a fraction look-up PROM. The seed biased exponent look-up table is stored in a 512-by-8-bit PROM. This table consists of two sections – the DEC format section, which occupies addresses $000-0FF_{16}$, and the IEEE section, which occupies addresses $100-1FF_{16}$. The appropriate table will be selected automatically if address line A_8 is wired to the Am29325's IEEE/$\overline{\text{DEC}}$ pin. The equations implemented by these table sections are:

DEC table: seed biased exponent
$= 257_{10}$ – input biased exponent

IEEE table: seed biased exponent
$= 252_{10}$ – input biased exponent

Table c1 lists the contents of this PROM.

The seed fraction look-up table is stored in one or more PROMs, the number of PROMs depending on the desired accuracy of the seed value. The hardware depicted in Figure c2 uses two 4K-by-8-bit PROMs to implement a fraction look-up table whose inputs are the 12 MSBs of the input argument's fraction. These PROMs output the 16 MSBs of the seed's fraction field – the remaining 7 bits of fraction are set to 0. The equation implemented in this table is:

$$\text{seed fraction} = \frac{2}{1 + \text{input fraction}} - 1,$$

where the value of the input fraction falls in the range

$$0 \leq \text{input fraction} < 1.$$

Note that the seed fraction must also be constrained to fall in the range

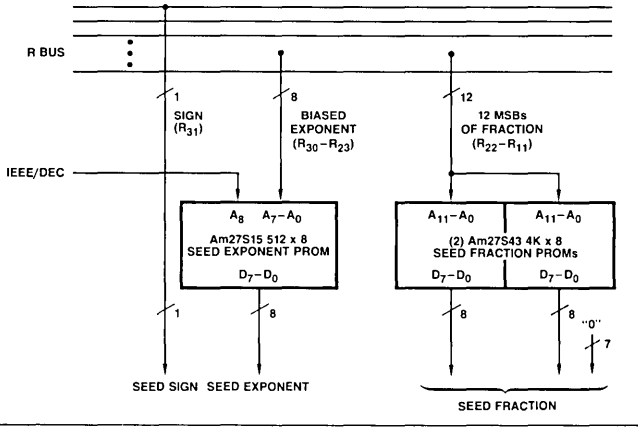
$$0 \leq \text{seed fraction} < 1.$$

Therefore, if the input fraction is 0, the corresponding seed fraction stored in the table must be $.1111...111_2$, not 1.0_2. The same seed fraction look-up table may be used for both IEEE and DEC formats. Table c2 contains a partial listing for the seed fraction look-up table shown in Figure c2.

TABLE c2. CONTENTS OF THE SEED FRACTION PROMs

Address (16)	Value of Input Fraction (10)	Value of Seed Fraction (10)	PROM Outputs (16) $R_{22}-R_{15}$	$R_{14}-R_7$
000	0.0	0.9999999999 (see text)	FF	FF
001	0.0002441406	0.9995118370	FF	E0
002	0.0004882812	0.9990239150	FF	C0
003	0.0007324219	0.9985362280	FF	A0
004	0.0009765625	0.9980487790	FF	80
005	0.0012207031	0.9975615710	FF	60
006	0.0014648438	0.9970745970	FF	40
007	0.0017089844	0.9965878630	FF	20
008	0.0019531250	0.9961013650	FF	00
009	0.0021972656	0.9956151030	FE	E1
00A	0.0024414063	0.9951290800	FE	C0
00B	0.0026855469	0.9946432920	FE	A1
00C	0.0029296875	0.9941577400	FE	81
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
FF6	0.9975585938	0.0012221950	00	50
FF7	0.9978027344	0.0010998410	00	48
FF8	0.9980486750	0.0009775170	00	40
FF9	0.9982910156	0.0008552230	00	38
FFA	0.9985351563	0.0007329590	00	30
FFB	0.9987792969	0.0006107240	00	28
FFC	0.9990234375	0.0004885200	00	20
FFD	0.9992675781	0.0003663450	00	18
FFE	0.9995117188	0.0002442000	00	10
FFF	0.9997558594	0.0001220850	00	08

Figure c2. The Hardware Lookup-Up Table



05621A-21

With the hardware look-up table in place, the reciprocal of value B can be calculated with the following series of operations:

1.) Place B on both the R and S buses. The 2 : 1 multiplexer at the output of the hardware look-up table should select the output of the look-up table. (see Figure c3-a)

2.) Load the seed value x_0 into register R and load B into register S. Select the R TIMES S operation. (see Figure c3-b)

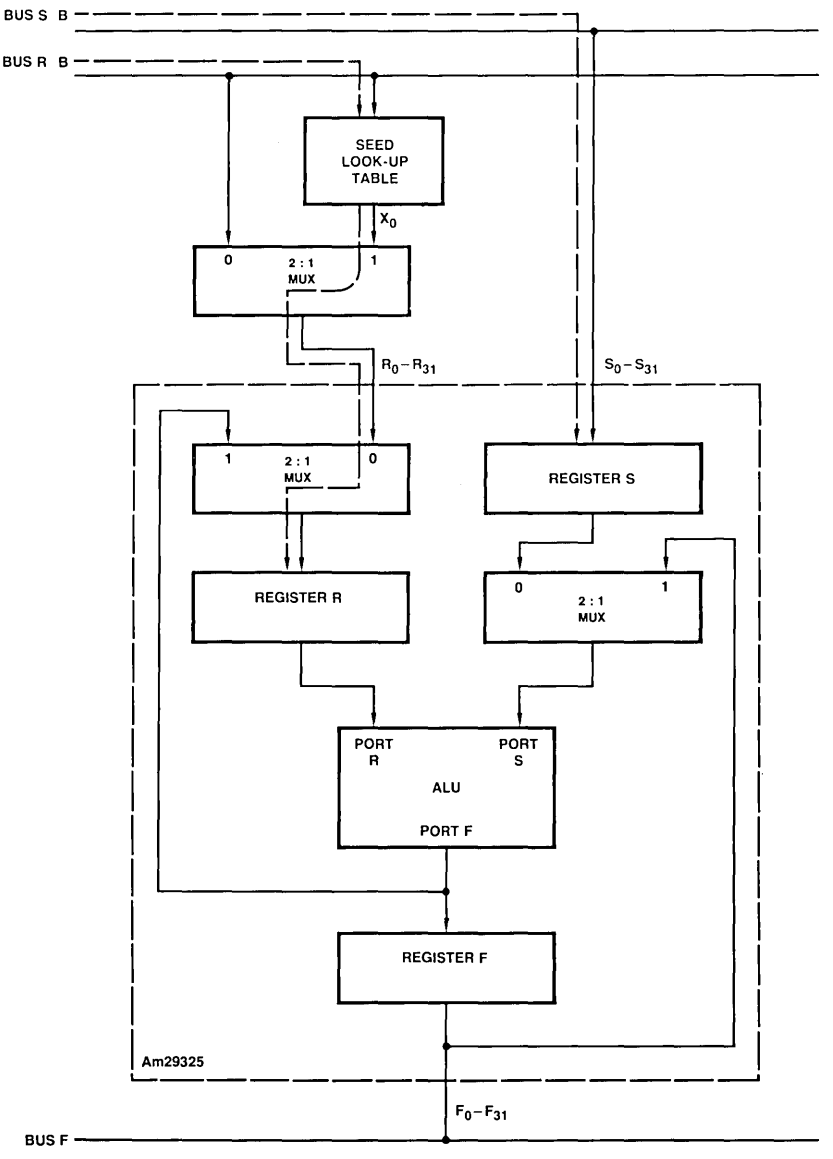
3.) Load product $B*x_0$ into register F. Select the 2 MINUS S operation, and select register F as the input to the ALU S port. (see Figure c3-c)

4.) Load $2 - B*x_0$ into register F. Select the R TIMES S operation and select register F as the input to the ALU S port. (see Figure c3-d)

5.) Load the value x_1 ($x_1 = x_0(2 - B*x_0)$) into registers R and F. Select the R TIMES S operation. (see Figure c3-e)

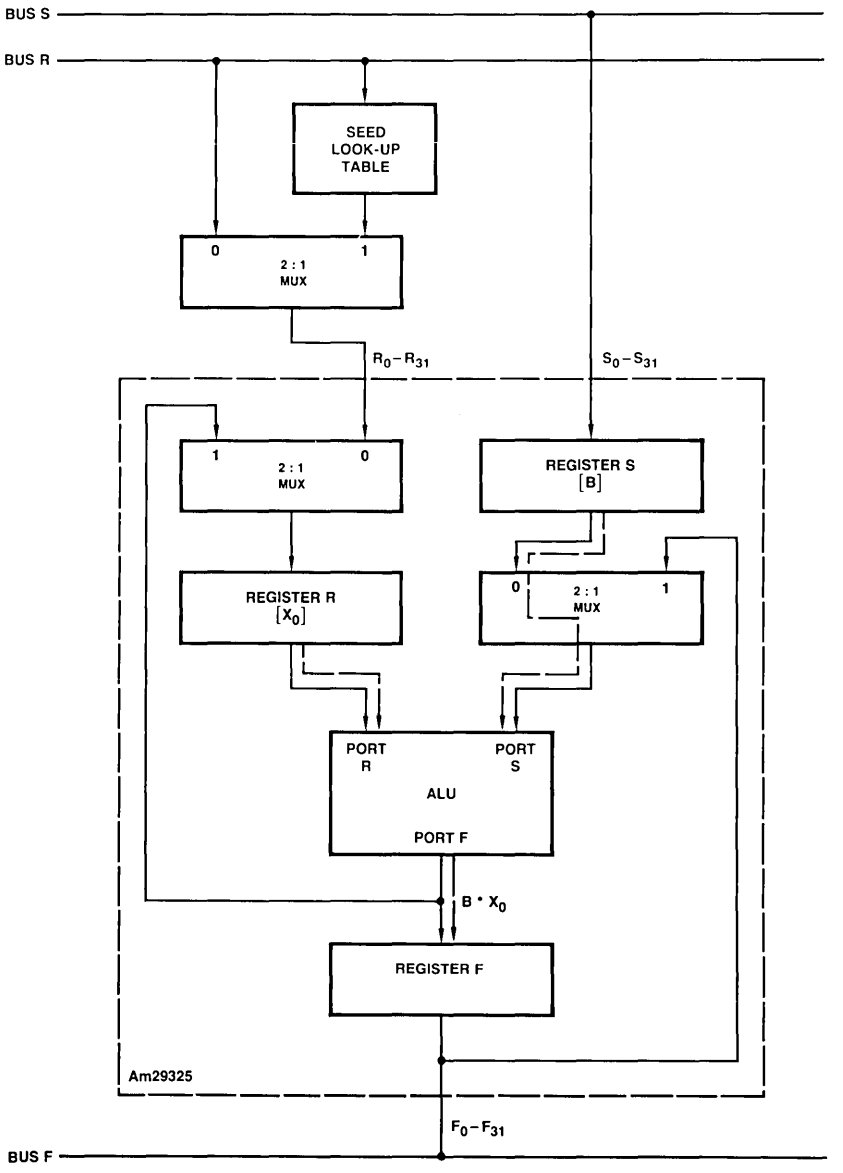
6.) Repeat steps 3 through 5 until the result has the accuracy desired.

Figure c3-a. Data Flow for Step 1 of the Reciprocal Procedure



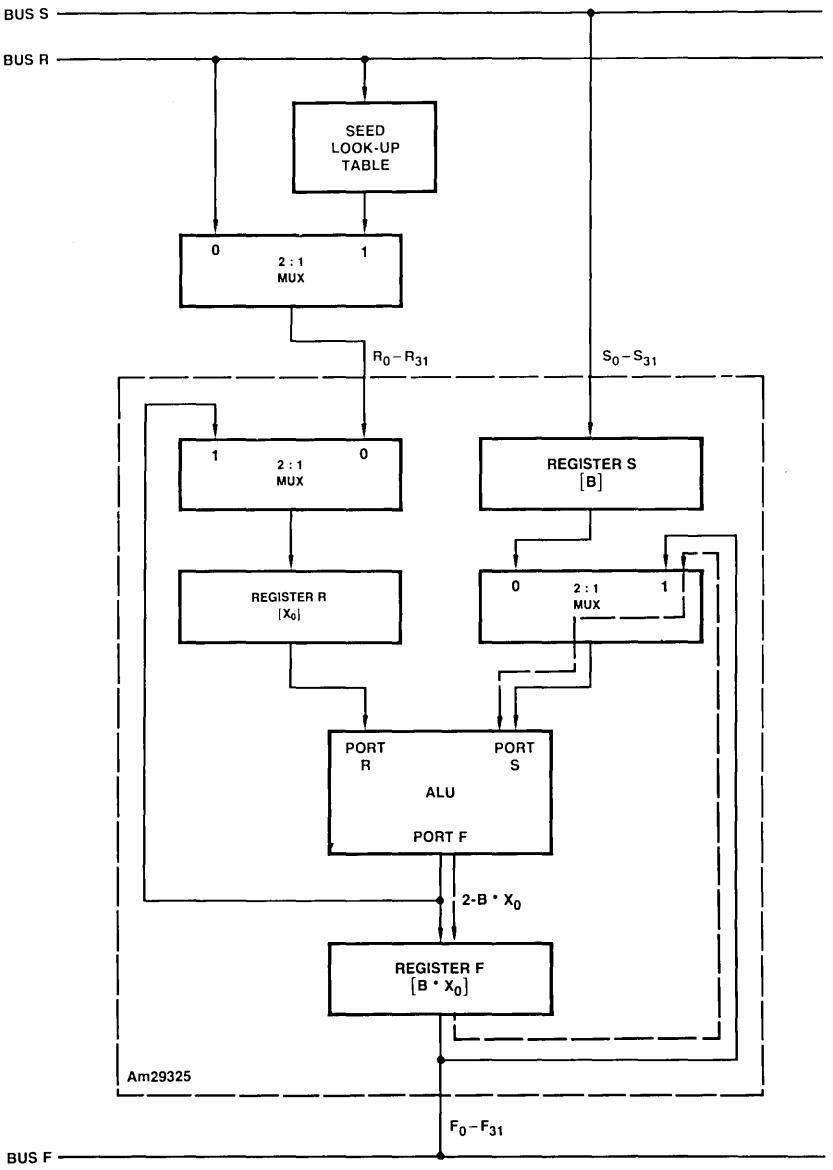
05621B-22

Figure c3-b. Data Flow for Step 2 of the Reciprocal Procedure



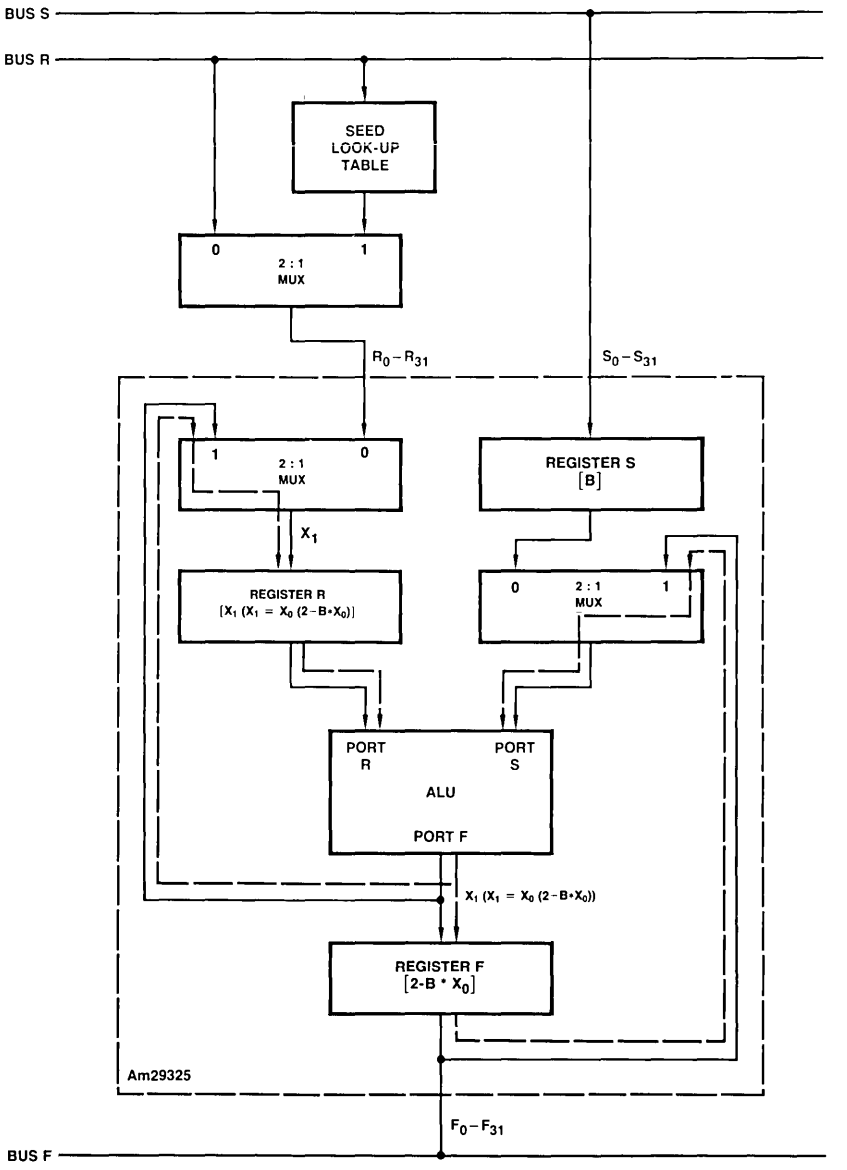
05621B-23

Figure c3-c. Data Flow for Step 3 of the Reciprocal Procedure



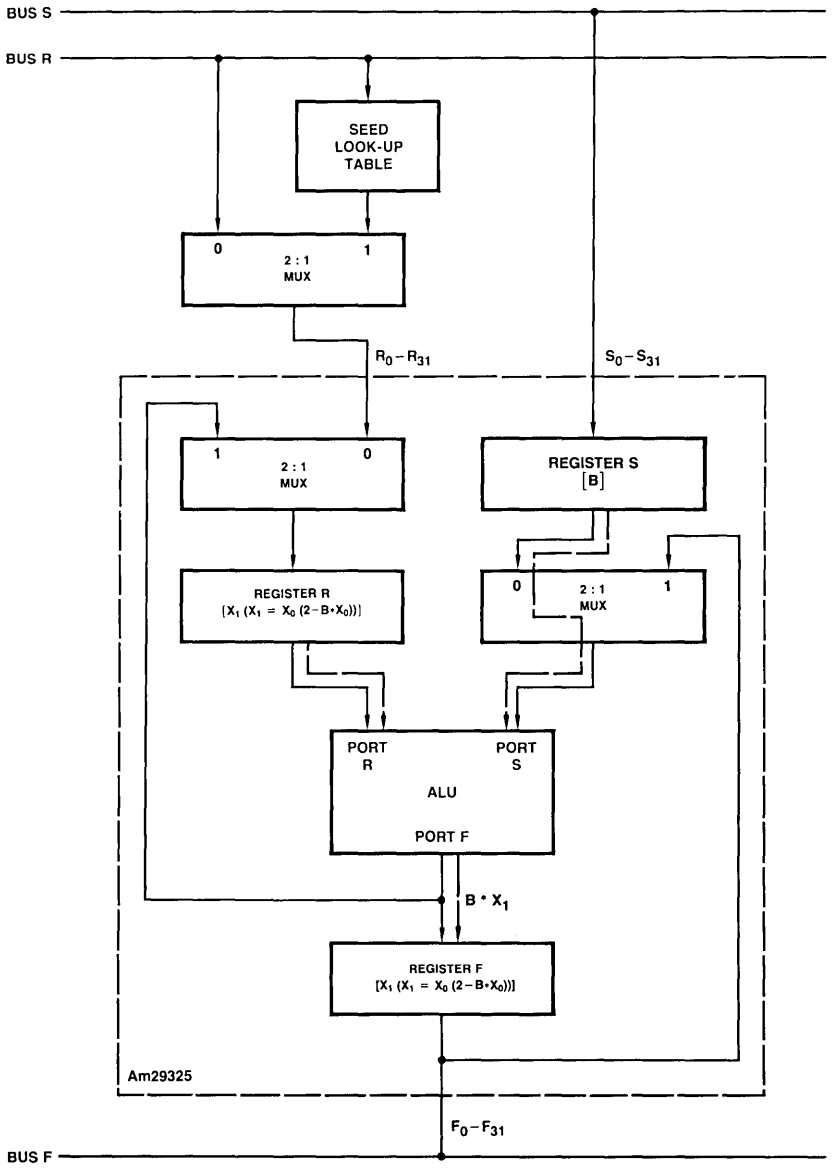
05621A-24

Figure c3-d. Data Flow for Step 4 of the Reciprocal Procedure



05621B-25

Figure c3-e. Data Flow for Step 5 of the Reciprocal Procedure



05621B-26

A tabular description of the operations above is given in Table c3. The following examples, performed in IEEE format, illustrate the process.

Example 1:

Find the reciprocal of 25.3.

Solution: The IEEE floating-point representation for 25.3 is $41CA6666_{16}$. The reciprocal process is begun by feeding this value to both the seed look-up table and port S. The look-up table produces the value $.03952789_{10}$ ($3D21E800_{16}$). The reciprocal is evaluated using the procedure described above; register values for each step are given in Table c4. The expected result, to the precision of the floating-point word, is $.03952569_{10}$ ($3D21E5B1_{16}$). In this case the expected result is produced after the first iteration. All subsequent iterations produce the same result, and are therefore unnecessary.

TABLE c3. SEQUENCE OF EVENTS FOR EVALUATING RECIPROCALS

Clock Cycle	I_0-I_2	I_3	I_4	$\overline{ENR}$	$\overline{ENS}$	$\overline{ENF}$	Register R	Register S	Register F	
1	Y	X	0	0	0	X	–	–	–	
2	R TIMES S	0	X	1	1	0	X_0	B	–	
3	2 MINUS S	1	X	1	1	0	X_0	B	$B \cdot X_0$	First iteration
4	R TIMES S	1	1	0	1	0	X_0	B	$2-B \cdot X_0$	First iteration
5	R TIMES S	0	X	1	1	0	$X_1(= X_0(2-B \cdot X_0))$	B	$X_1(= X_0(2-B \cdot X_0))$	First iteration
6	2 MINUS S	1	X	1	1	0	X_1	B	$B \cdot X_1$	Second iteration
7	R TIMES S	1	1	0	1	0	X_1	B	$2-B \cdot X_1$	Second iteration
8	R TIMES S	0	X	1	1	0	$X_2(= X_1(2-B \cdot X_1))$	B	$X_2(= X_1(2-B \cdot X_1))$	Second iteration

X = DON'T CARE

TABLE c4. INPUT BUS AND REGISTER VALUES FOR EXAMPLE 1

Clock Cycle	R Input	S Input	Register R	Register S	Register F	
1	3D21E800 (.03952789)	$41CA6666_{16}$ (25.3)	–	–	–	
2	–	–	$3D21E800_{16}$ (.03952789)	$41CA6666_{16}$ (25.3)	–	
3	–	–	$3D21E800_{16}$ (.03952789)	$41CA6666_{16}$ (25.3)	$3F8001D3_{16}$ (1.0000556)	
4	–	–	$3D21E800_{16}$ (.03952789)	$41CA6666_{16}$ (25.3)	$3F7FFC5A_{16}$ (.99984419)	
5	–	–	$3D21E5B1_{16}$ (.03952569)	$41CA6666_{16}$ (25.3)	$3D21E5B1_{16}$ (.03952569)	← Result of first iteration
6	–	–	$3D21E5B1_{16}$ (.03952569)	$41CA6666_{16}$ (25.3)	$3F7FFFFF_{16}$ (.99999994)	
7	–	–	$3D21E5B1_{16}$ (.03952569)	$41CA6666_{16}$ (25.3)	$3F800000_{16}$ (1.0)	
8	–	–	$3D21E5B1_{16}$ (.03952569)	$41CA6666_{16}$ (25.3)	$3D21E5B1_{16}$ (.03952569)	← Result of second iteration

Example 2:

Find the reciprocal of −.4725.

Solution: The IEEE floating-point representation for −.4725 is $BEF1EB85_{16}$. The reciprocal process is begun by feeding this value to both the seed look-up table and port S. The look-up table produces the value -2.11621094_{10} ($C0077000_{16}$). The reciprocal is evaluated using the procedure described above; register values for each step are given in Table c5. The expected result, to the precision of the floating-point word, is -2.116402_{10} ($C0077322_{16}$). In this case the expected result is produced after the first iteration. All subsequent iterations produce the same result, and are therefore unnecessary.

TABLE c5. INPUT BUS AND REGISTER VALUES FOR EXAMPLE 2

Clock Cycle	R Input	S Input	Register R	Register S	Register F	
1	$C0077000_{16}$ (−2.1162109)	$BEF1EB85_{16}$ (−0.4725)	–	–	–	
2	–	–	$C0077000_{16}$ (−2.1162109)	$BEF1EB85_{16}$ (−0.4725)	–	
3	–	–	$C0077000_{16}$ (−2.1162109)	$BEF1EB85_{16}$ (−0.4725)	$3F7FFA14_{16}$ (0.99990963)	
4	–	–	$C0077000_{16}$ (−2.1162109)	$BEF1EB85_{16}$ (−0.4725)	$3F8002F6_{16}$ (1.0000904)	
5	–	–	$C0077322_{16}$ (−2.116402)	$BEF1EB85_{16}$ (−0.4725)	$C0077322_{16}$ (−2.116402)	← Result of first iteration
6	–	–	$C0077322_{16}$ (−2.116402)	$BEF1EB85_{16}$ (−0.4725)	$3F800000_{16}$ (1.0)	
7	–	–	$C0077322_{16}$ (−2.116402)	$BEF1EB85_{16}$ (−0.4725)	$3F800000_{16}$ (1.0)	
8	–	–	$C0077322_{16}$ (−2.116402)	$BEF1EB85_{16}$ (−0.4725)	$C0077322_{16}$ (−2.116402)	← Result of second iteration

APPENDIX D:

Summary of Flag Operation

Tables d1, d2, and d3 summarize flag operation for the IEEE mode, the DEC mode, and for the IEEE-TO-DEC and DEC-TO-IEEE operations.

TABLE d1. FLAG SUMMARY FOR IEEE MODE

Operation	Condition(s)	INV	OVF	UNF	INE	ZER	NAN
Any operation listed in the IEEE Invalid Operations Table		H	L	L	L	L	H
R PLUS S R MINUS S R TIMES S 2 MINUS S	Input operands are finite, $\|\text{rounded result}\| \geq 2^{128}$	L	H	L	H	L	L
R PLUS S R MINUS S R TIMES S	$0 < \|\text{rounded result}\| < 2^{-126}$	L	L	H	H	H	L
R PLUS S R MINUS S R TIMES S 2 MINUS S INT-TO-FP FP-TO-INT	Final result does not equal infinitely precise result	L	*	*	H	*	L
R PLUS S R MINUS S R TIMES S 2 MINUS S INT-TO-FP FP-TO-INT	Final result is zero	L	L	*	*	H	L
R PLUS S R MINUS S R TIMES S 2 MINUS S FP-TO-INT	Final result is a NAN	*	L	L	L	L	H

Notes: INV = Invalid operation flag
OVF = Overflow flag
UNF = Underflow flag
INE = Inexact flag
ZER = Zero flag
NAN = NAN flag
L = LOW
H = HIGH
* = State of flag depends on the input operands and the operation performed

TABLE d2. FLAG SUMMARY FOR DEC MODE

Operation	Condition(s)	INV	OVF	UNF	INE	ZER	NAN
FP-TO-INT	Rounded result $> 2^{31}-1$ or rounded result $< -2^{31}$	H	L	L	L	L	H
FP-TO-INT	Input is a DEC reserved operand	H	L	L	L	L	H
R PLUS S R MINUS S R TIMES S 2 MINUS S	$\lvert$Rounded result$\rvert \geq 2^{127}$	L	H	L	H	L	H
R PLUS S R MINUS S R TIMES S	$0 < \lvert$rounded result$\rvert < 2^{-128}$	L	L	H	H	H	L
R PLUS S R MINUS S R TIMES S 2 MINUS S INT-TO-FP FP-TO-INT	Final result does not equal infinitely precise result	L	*	*	H	*	*
R PLUS S R MINUS S R TIMES S 2 MINUS S INT-TO-FP FP-TO-INT	Final result is zero	L	L	*	*	H	L
R PLUS S R MINUS S R TIMES S 2 MINUS S FP-TO-INT	Final result is a DEC reserved operand	*	*	L	L	L	H

Notes: INV = Invalid operation flag
OVF = Overflow flag
UNF = Underflow flag
INE = Inexact flag
ZER = Zero flag
NAN = NAN flag
L = LOW
H = HIGH
* = State of flag depends on the input operands and the operation performed

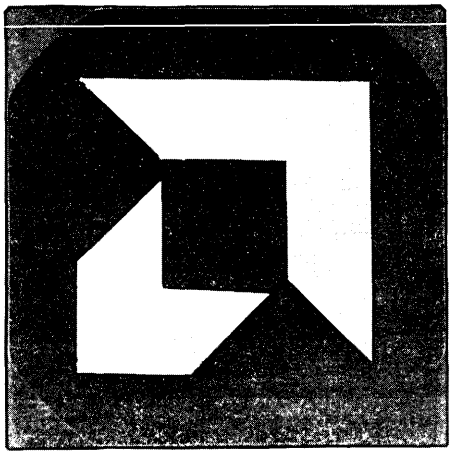
TABLE d3. FLAG SUMMARY FOR IEEE-TO-DEC AND DEC-TO-IEEE CONVERSIONS

Operation	Condition(s)	INV	OVF	UNF	INE	ZER	NAN
IEEE-TO-DEC	Input is a NAN	H	L	L	L	L	H
IEEE-TO-DEC	$\lvert$Input$\rvert \geq 2^{127}$	L	H	L	H	L	H
DEC-TO-IEEE	Input is a DEC reserved operand	H	L	L	L	L	H
DEC-TO-IEEE	$0 < \lvert$rounded result$\rvert < 2^{-126}$	L	L	H	H	H	L
DEC-TO-IEEE IEEE-TO-DEC	Final result is zero	L	L	*	*	H	L

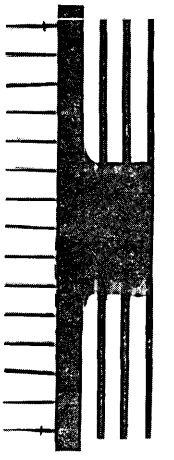
Notes: INV = Invalid operation flag
OVF = Overflow flag
UNF = Underflow flag
INE = Inexact flag
ZER = Zero flag
NAN = NAN flag
L = LOW
H = HIGH
* = State of flag depends on the input operands and the operation performed

PACKAGE INFORMATION

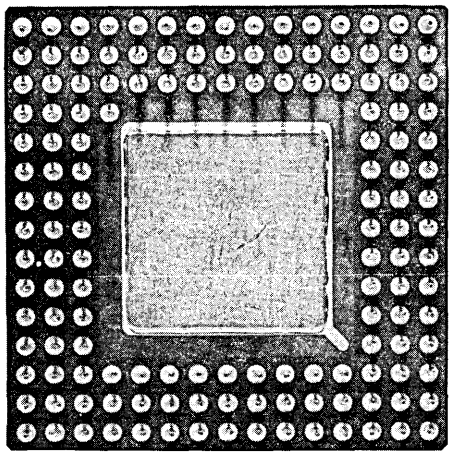
PACKAGE PHOTOGRAPHS



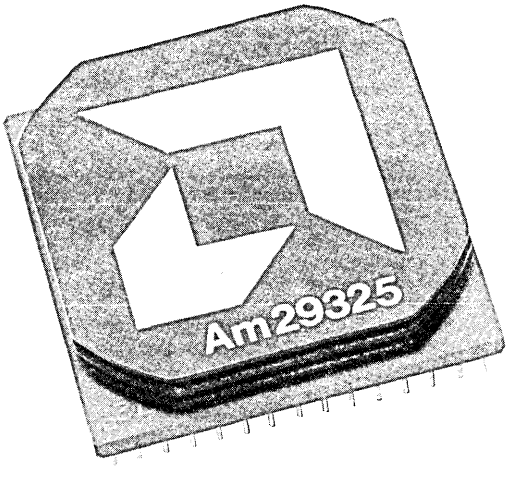
Top View



Lateral View



Bottom View



Isometric View

ABSOLUTE MAXIMUM RATINGS

Storage Temperature ..... −65 to +150°C
Temperature Under Bias − T_C ..... −55 to +125°C
Supply Voltage to Ground Potential
Continuous ..... −0.5 to +7.0V
DC Voltage Applied to Outputs
for High State ..... −0.5V to +V_{CC} Max
DC Input Voltage ..... −0.5 to +5.5V
DC Output Current, into Outputs 30mA
DC Input Current ..... −30 to +5.0mA

Stresses above those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent device failure. Functionality at or above these limits is not implied. Exposure to absolute maximum ratings for extended periods may affect device reliability.

OPERATING RANGES

Commercial (C) Devices
Temperature (T_A) ..... 0 to +70°C
Supply Voltage ..... +4.75 to +5.25V

Military (M) Devices
Temperature (T_C) ..... −55 to +125°C
Supply Voltage ..... +4.5 to +5.5V

Operating ranges define those limits over which the functionality of the device is guaranteed.

DC CHARACTERSITICS OVER OPERATING RANGE unless otherwise specified

Parameter	Description	Test Conditions (Note 1)		Min	Typ (Note 2)	Max	Units
V_{OH}	Output HIGH Voltage	V_{CC} = Min V_{IN} = V_{IL} or V_{IH} I_{OH} = −0.4mA		2.4	2.7		Volts
V_{OL}	Output LOW Voltage	V_{CC} = Min V_{IN} = V_{IL} or V_{IH} I_{OL} = 4.0mA			0.3	0.5	Volts
V_{IH}	Input HIGH Level	Guaranteed Input Logical HIGH Voltage for All Inputs		2.0			Volts
V_{IL}	Input LOW Level	Guaranteed Input Logical LOW Voltage for All Inputs				0.8	Volts
V_I	Input Clamp Voltage	V_{CC} = Min I_{IN} = −18mA				−1.5	Volts
I_{IL}	Input LOW Current	V_{CC} = Max V_{IN} = 0.4V				−0.4	mA
I_{IH}	Input HIGH Current	V_{CC} = Max V_{IN} = 2.4V				75	μA
I_I	Input HIGH Current	V_{CC} = Max V_{IN} = 5.5V				1	mA
I_{OZH} I_{OZL}	F_0−F_{31} Off State (High Impedance) Output Current	V_{CC} = Max	V_O = 2.4V			25	μA
			V_O = .4V			−25	
I_{SC}	Output Short Circuit Current (Note 3)	V_{CC} = Max V_O = 0V	F_0−F_{31} Outputs	−3		−30	mA
			Flag Outputs	−3		−30	
I_{CC}	Power Supply Current (Note 4)	V_{CC} = Max	COM'L, MIL T_A = +25°C				mA
			COM'L Only T_A = 0 to +70°C				
			COM'L Only T_A = +70°C				
			MIL Only T_A = −55 to +125°C				
			MIL Only T_A = +125°C				

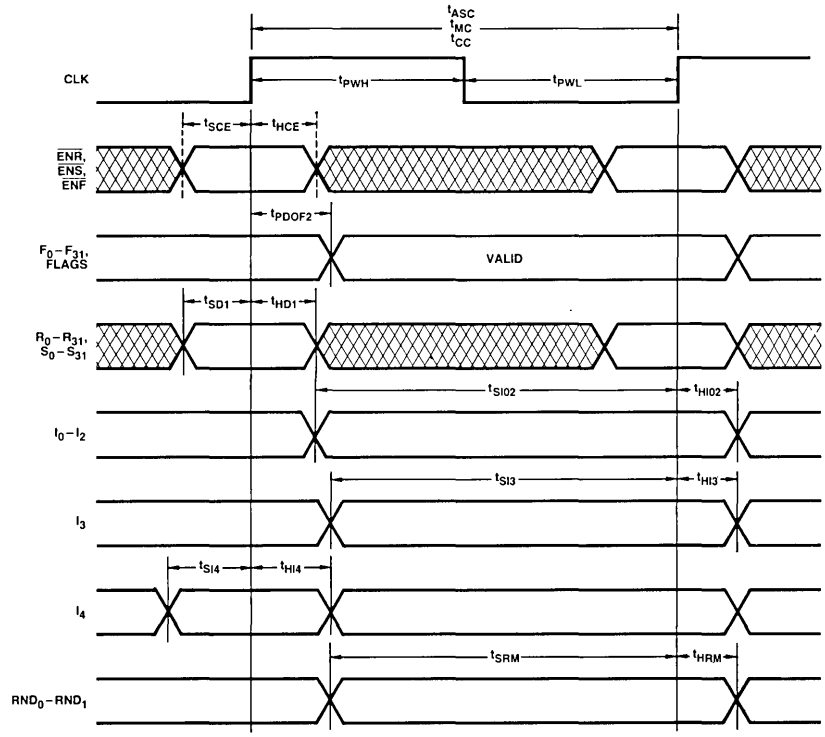
Notes: 1. For conditions shown as Min or Max, use the appropriate value specified under Operating Ranges for the applicable device type.
2. Typical values are for V_{CC} = +25°C ambient and maximum loading.
3. Not more than one output should be shorted at a time. Duration of the short circuit test should not exceed one second.
4. Measured with $\overline{OE}$ LOW, and with all output bits (F_0−F_{31} and flag outputs) LOW.

Am29325
SWITCHING CHARACTERISTICS OVER OPERATING RANGE

Parameters	Description		Test Conditions	COM'L (Note 2) T_A = 25°C, V_{CC} = 5.0V Typ	COM'L (Note 2) T_A = 0 to +70°C, V_{CC} = +5V ±5% Min	COM'L (Note 2) T_A = 0 to +70°C, V_{CC} = +5V ±5% Max	MIL T_C = −55 to 125°C, V_{CC} = +5V ±10% Min	MIL T_C = −55 to 125°C, V_{CC} = +5V ±10% Max	Units
t_{ASC}	Clocked Add, Subtract Time (R PLUS S, R MINUS S, 2 MINUS S)								ns
t_{MC}	Clocked Multiply Time (R TIMES S)								ns
t_{CC}	Clocked Conversion Time (INT-TO-FP, FP-TO-INT, IEEE-TO-DEC, DEC-TO-IEEE)								ns
t_{ASUC}	Unclocked Add, Subtract Time (R, S to F, Flags) for R PLUS S, R MINUS S, and 2 MINUS S Instructions		FT_0 = HIGH, FT_1 = HIGH						ns
t_{MUC}	Unclocked Multiply Time (R, S to F, Flags) for R TIMES S Instruction								ns
t_{CUC}	Unclocked Conversion Time (R, S to F, Flags) for INT-TO-FP, FP-TO-INT, IEEE-TO-DEC and DEC-TO-IEEE Instructions								ns
t_{PWH}	Clock Pulse Width HIGH								ns
t_{PWL}	Clock Pulse Width LOW								ns
t_{PDOF1}	Clock to F_0–F_{31} and Flag Outputs		FT_0 = LOW, FT_1 = HIGH						ns
t_{PDOF2}			FT_1 = LOW						ns
t_{PZL}	$\overline{OE}$ Enable Time	Z to LOW							ns
t_{PZH}		Z to HIGH							ns
t_{PLZ}	$\overline{OE}$ Disable Time	LOW to Z							ns
t_{PHZ}		HIGH to Z							ns
t_{PZL16}	Clock↑ to F_0–F_{15} Enable, 16-Bit I/O Mode	Z to LOW	S16/$\overline{32}$ = HIGH, ONEBUS = LOW						ns
t_{PZH16}		Z to HIGH							ns
t_{PLZ16}	Clock↓ to F_0–F_{15} Disable, 16-Bit I/O Mode	LOW to Z	S16/$\overline{32}$ = HIGH, ONEBUS = LOW						ns
t_{PHZ16}		HIGH to Z							ns
t_{PZL16}	Clock↓ to F_{16}–F_{31} Enable, 16-Bit I/O Mode	Z to LOW							ns
t_{PZH16}		Z to HIGH							ns
t_{PLZ16}	Clock↑ to F_{16}–F_{31} Disable, 16-Bit I/O Mode	LOW to Z							ns
t_{PHZ16}		HIGH to Z							ns
t_{SCE}	Register Clock Enable Setup Time		FT_0 = LOW, FT_1 = LOW						ns
t_{HCE}	Register Clock Enable Hold Time		FT_0 = LOW, FT_1 = LOW						ns
t_{SD1}	R_0–R_{31}, S_0–S_{31} Setup Time (Note 1)		FT_0 = LOW						ns
t_{HD1}	R_0–R_{31}, S_0–S_{31} Hold Time (Note 1)								ns
t_{SD2}	R_0–R_{31}, S_0–S_{31} Setup Time (Note 1)		FT_0 = HIGH, FT_1 = LOW						ns
t_{HD2}	R_0–R_{31}, S_0–S_{31} Hold Time (Note 1)								ns
t_{SI02}	I_0–I_2 Instruction Select Setup Time		FT for Destination Register = LOW						ns
t_{HI02}	I_0–I_2 Instruction Select Hold Time								ns
t_{PDI02}	I_0–I_2 Instruction Select to F_0–F_{31}, Flags		FT_1 = HIGH						ns
t_{SI3}	I_3 Port S Input Select Setup Time		FT_1 = LOW						ns
t_{HI3}	I_3 Port S Input Select Hold Time								ns
t_{SI4}	I_4 Register R Input Select Setup Time (Note 1)		FT_0 = LOW						ns
t_{HI4}	I_4 Register R Input Select Hold Time (Note 1)								ns
t_{SRM}	Round Mode Select Setup Time		FT for Destination Register = LOW						ns
t_{HRM}	Round Mode Select Hold Time								ns
t_{PRF}	Round Mode Select to F_0–F_{31}, Flags		FT_1 = HIGH						ns

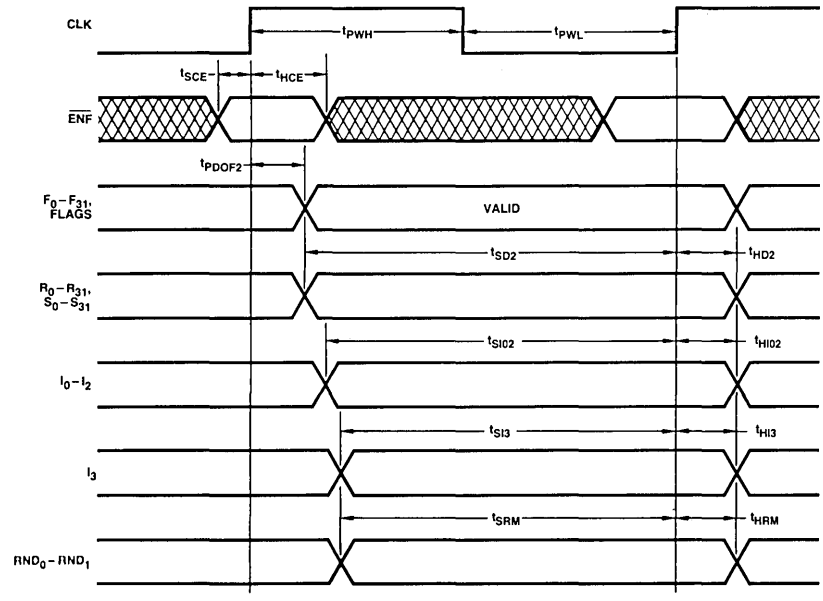
Notes: 1. See timing diagram for desired mode of operation to determine clock edge to which these setup and hold times apply.
2. At air velocity of__linear feet per minute.

CLOCKED OPERATION: FT_0 = LOW, FT_1 = LOW



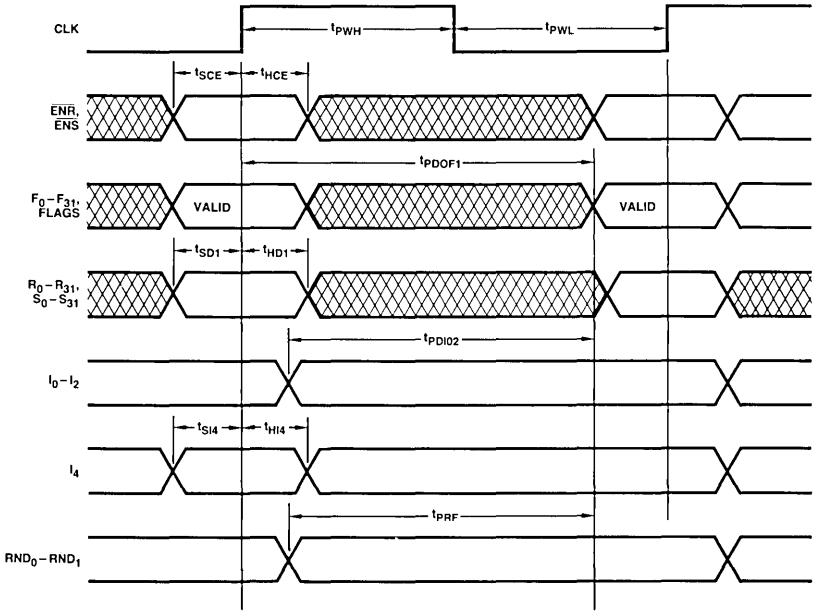
05621A-31

CLOCKED OPERATION: FT_0 = HIGH, FT_1 = LOW



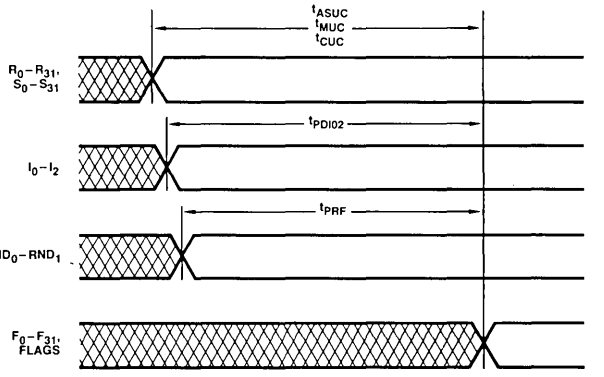
05621A-32

CLOCKED OPERATION: FT_0 = LOW, FT_1 = HIGH



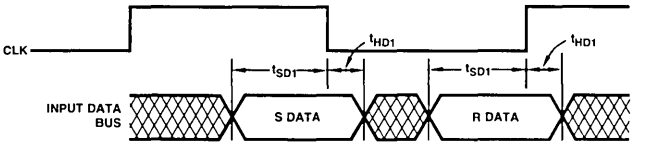
05621A-33

FLOW-THROUGH OPERATION (FT_0 = HIGH, FT_1 = HIGH)



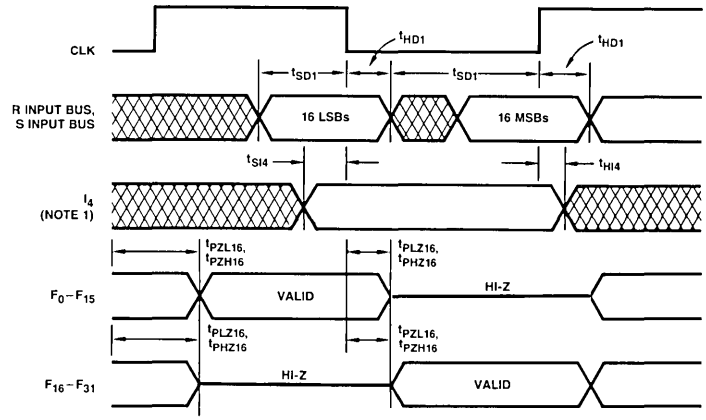
05621A-27

32-BIT, SINGLE-INPUT-BUS MODE



05621A-28

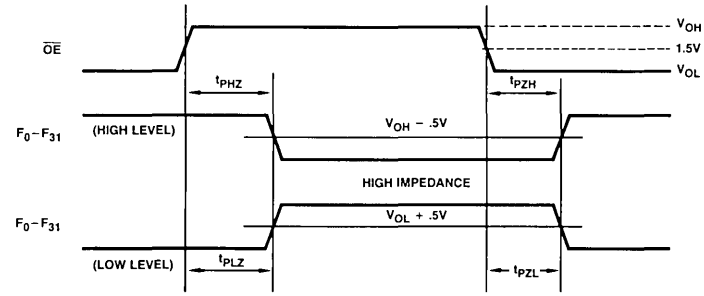
16-BIT, TWO-INPUT-BUS MODE



Note 1. I_4 has special setup and hold time requirements in this mode. All other control signals have timing requirements as shown in the diagram "Clocked operation, FT_0 = LOW, FT_1 = LOW."

05621B-29

OUTPUT ENABLE/DISABLE TIMING



05621A-30

Am29325 PINOUT

SORTED BY PIN NUMBER

Line #	Pin #	Functional Name
1	A1	Inexact
2	A2	Invalid
3	A3	F_{29}
4	A4	F_{30}
5	A5	F_{23}
6	A6	F_{26}
7	A7	F_{21}
8	A8	F_{22}
9	A9	F_{17}
10	A10	F_{18}
11	A11	F_{13}
12	A12	F_{12}
13	A13	F_7
14	A14	F_8
15	A15	F_5
16	B1	I_2
17	B2	NAN
18	B3	Zero
19	B4	F_{31}
20	B5	Overflow
21	B6	F_{27}
22	B7	F_{24}
23	B8	F_{19}
24	B9	F_{20}
25	B10	F_{15}
26	B11	F_{14}
27	B12	F_9
28	B13	F_6
29	B14	F_3
30	B15	F_4
31	C1	I_1
32	C2	I_0
33	C3	GND, TTL
34	C4	GND, TTL
35	C5	Underflow
36	C6	F_{28}
37	C7	F_{25}
38	C8	V_{CC}, TTL
39	C9	V_{CC}, TTL
40	C10	F_{16}
41	C11	F_{11}
42	C12	F_{10}
43	C13	GND, TTL
44	C14	F_2
45	C15	F_1
46	D1	$\overline{ENF}$
47	D2	IEEE/$\overline{DEC}$
48	D3	$\overline{ENR}$
49	D13	GND, TTL
50	D14	GND, TTL
51	D15	GND, TTL
52	E1	I_4
53	E2	$\overline{FT_0}$
54	E3	$\overline{ENS}$
55	E13	GND, TTL
56	E14	F_0
57	E15	PROJ/$\overline{AFF}$
58	F1	ONEBUS
59	F2	FT_1
60	F3	S16/$\overline{32}$

SORTED BY FUNCTIONAL NAME

Functional Name	Pin #
$\overline{CLK}$	J1
$\overline{ENF}$	D1
$\overline{ENR}$	D3
$\overline{ENS}$	E3
F_0	E14
F_1	C15
F_2	C14
F_3	B14
F_4	B15
F_5	A15
F_6	B13
F_7	A13
F_8	A14
F_9	B12
F_{10}	C12
F_{11}	C11
F_{12}	A12
F_{13}	A11
F_{14}	B11
F_{15}	B10
F_{16}	C10
F_{17}	A9
F_{18}	A10
F_{19}	B8
F_{20}	B9
F_{21}	A7
F_{22}	A8
F_{23}	A5
F_{24}	B7
F_{25}	C7
F_{26}	A6
F_{27}	B6
F_{28}	C6
F_{29}	A3
F_{30}	A4
F_{31}	B4
$\overline{FT_0}$	E2
FT_1	F2
GND, ECL	N3
GND, ECL	H14
GND, ECL	G13
GND, ECL	M3
GND, ECL	H13
GND, ECL	J13
GND, TTL	D15
GND, TTL	D14
GND, TTL	E13
GND, TTL	F13
GND, TTL	C4
GND, TTL	C3
GND, TTL	D13
GND, TTL	C13
I_0	C2
I_1	C1
I_2	B1
I_3	P9
I_4	E1
IEEE/$\overline{DEC}$	D2
Inexact	A1
Invalid	A2

Am29325 PINOUT (Cont)

SORTED BY PIN NUMBER

Line #	Pin #	Functional Name
61	F13	GND, TTL
62	F14	S_1
63	F15	$\overline{S_0}$
64	G1	$\overline{OE}$
65	G2	V_{CC}, ECL
66	G3	V_{CC}, ECL
67	G13	GND, ECL
68	G14	S_2
69	G15	S_3
70	H1	V_{CC}, ECL
71	H2	V_{CC}, ECL
72	H3	V_{CC}, ECL
73	H13	GND, ECL
74	H14	GND, ECL
75	H15	S_5
76	J1	CLK
77	J2	RND_0
78	J3	V_{CC}, ECL
79	J13	GND, ECL
80	J14	S_4
81	J15	S_7
82	K1	R_{31}
83	K2	RND_1
84	K3	R_{29}
85	K13	S_8
86	K14	S_9
87	K15	S_6
88	L1	R_{30}
89	L2	R_{27}
90	L3	R_{26}
91	L13	S_{13}
92	L14	S_{10}
93	L15	S_{11}
94	M1	R_{25}
95	M2	R_{28}
96	M3	GND, ECL
97	M13	S_{14}
98	M14	S_{15}
99	M15	S_{12}
100	N1	R_{24}
101	N2	R_{23}
102	N3	GND, ECL
103	N4	R_{15}
104	N5	R_{14}
105	N6	R_9
106	N7	R_8
107	N8	R_3
108	N9	R_0
109	N10	S_{28}
110	N11	S_{27}
111	N12	V_{CC}, ECL
112	N13	V_{CC}, ECL
113	N14	S_{18}
114	N15	S_{17}
115	P1	R_{21}
116	P2	R_{22}
117	P3	R_{19}
118	P4	R_{16}
119	P5	R_{11}
120	P6	R_{10}

SORTED BY FUNCTIONAL NAME

Functional Name	Pin #
$\overline{NAN}$	B2
$\overline{OE}$	G1
ONEBUS	F1
Overflow	B5
PROJ/$\overline{AFF}$	E15
R_0	N9
R_1	R8
R_2	R9
R_3	N8
R_4	P8
R_5	P7
R_6	R7
R_7	R6
R_8	N7
R_9	N6
R_{10}	P6
R_{11}	P5
R_{12}	R5
R_{13}	R4
R_{14}	N5
R_{15}	N4
R_{16}	P4
R_{17}	R2
R_{18}	R3
R_{19}	P3
R_{20}	R1
R_{21}	P1
R_{22}	P2
R_{23}	N2
R_{24}	N1
R_{25}	M1
R_{26}	L3
R_{27}	L2
R_{28}	M2
R_{29}	K3
R_{30}	L1
R_{31}	K1
RND_0	J2
RND_1	K2
S_0	F15
S_1	F14
S_2	G14
S_3	G15
S_4	J14
S_5	H15
S_6	K15
S_7	J15
S_8	K13
S_9	K14
S_{10}	L14
S_{11}	L15
S_{12}	M15
S_{13}	L13
S_{14}	M13
S_{15}	M14
S_{16}	P15
S16/$\overline{32}$	F3
S_{17}	N15
S_{18}	N14
S_{19}	R15

Am29325 PINOUT (Cont)

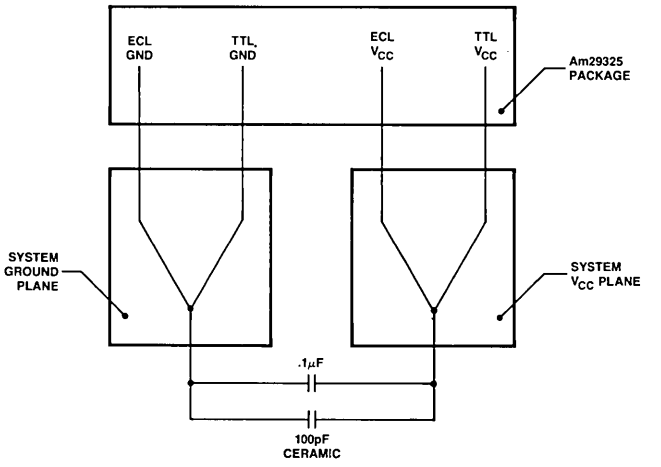
SORTED BY PIN NUMBER

Line #	Pin #	Functional Name
121	P7	R_5
122	P8	R_4
123	P9	I_3
124	P10	S_{31}
125	P11	S_{26}
126	P12	S_{25}
127	P13	S_{22}
128	P14	S_{21}
129	P15	S_{16}
130	R1	R_{20}
131	R2	R_{17}
132	R3	R_{18}
133	R4	R_{13}
134	R5	R_{12}
135	R6	R_7
136	R7	R_6
137	R8	R_1
138	R9	R_2
139	R10	S_{30}
140	R11	S_{29}
141	R12	S_{24}
142	R13	S_{23}
143	R14	S_{20}
144	R15	S_{19}

SORTED BY FUNCTIONAL NAME

Functional Name	Pin #
S_{20}	R14
S_{21}	P14
S_{22}	P13
S_{23}	R13
S_{24}	R12
S_{25}	P12
S_{26}	P11
S_{27}	N11
S_{28}	N10
S_{29}	R11
S_{30}	R10
S_{31}	P10
Underflow	C5
V_{CC}, ECL	J3
V_{CC}, ECL	G2
V_{CC}, ECL	G3
V_{CC}, ECL	H2
V_{CC}, ECL	N13
V_{CC}, ECL	N12
V_{CC}, ECL	H3
V_{CC}, ECL	H1
V_{CC}, TTL	C8
V_{CC}, TTL	C9
Zero	B3

POWER SUPPLY WIRING CONSIDERATIONS

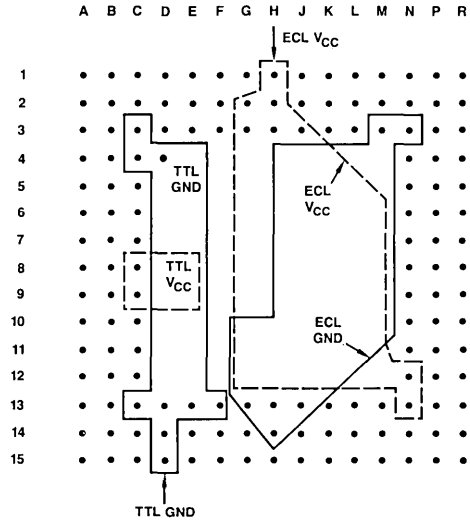


05621A-34

Notes: 1. All power supply pins must be connected.
2. ECL GND and TTL GND should not be connected directly into the main system ground plane. Using signal plane traces as short and wide as possible, ECL GND pins should be connected together, as should TTL GND pins, but without interconnection. These separate ground buses should be connected together and to the system ground plane at a decoupling capacitor close to the package. ECL V_{CC} and TTL V_{CC} should be treated similarly. See diagram above.

SUGGESTED PRINTED CIRCUIT BOARD LAYOUT

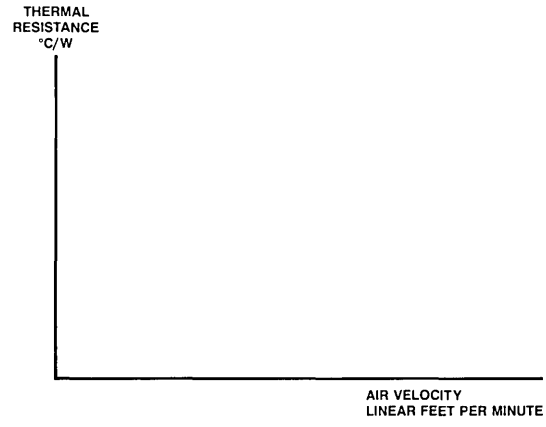
Bottom View



Note: 1. D4 (alignment pin) is not connected internally—may be wired to TTL ground or left unconnected.

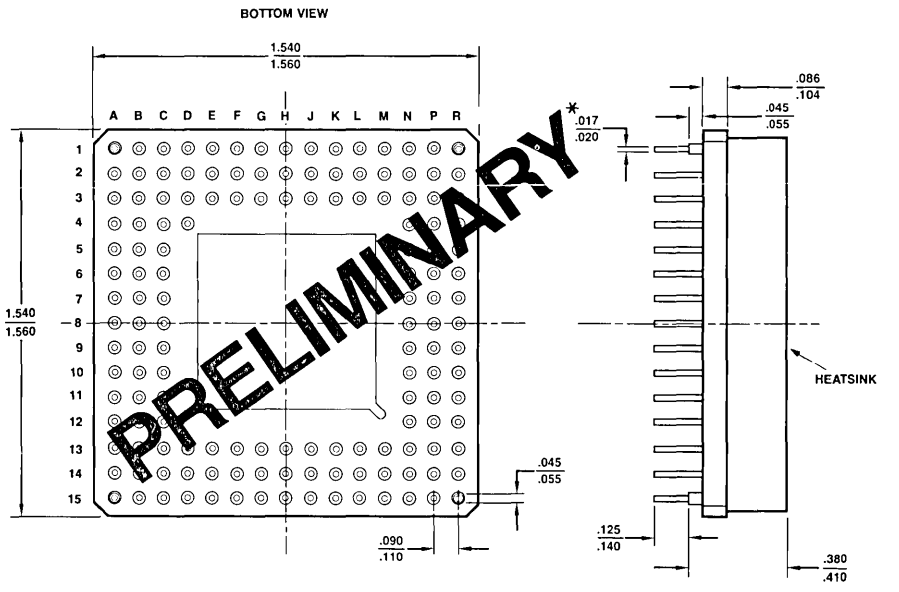
05621B-35

THERMAL CHARACTERISTICS



05621B-36

PHYSICAL DIMENSIONS



*Subject to change.

The International Standard of Quality guarantees the AQL on all electrical parameters, AC and DC, over the entire operating range.

Am29331

16-Bit Microprogram Sequencer

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- **16-Bits Address Up to 64K Words**

  Supports 80–90ns microcycle time for a 32-bit high performance system when used with the other members of the Am29300 Family.

- **Real Time Interrupt Support**

  Micro-TRAP and Interrupts are handled transparently at any microinstruction boundary.

- **Built-In Conditional Test Logic**

  Generates inequality evaluation branch conditions from four ALU status bits. Has eight external tests plus a polarity input.

- **Break-Point Logic**

  Built-in address comparator allows break-points in the microcode for debugging and statistics collection.

- **Master/Slave Error Checking**

  Two sequencers can operate in parallel as a Master and a Slave. The Slave generates a fault flag for unequal results.

- **33-Level Stack**

  Provides support for interrupts, loops and subroutine nesting. It can be accessed through the D-bus to support diagnostics.

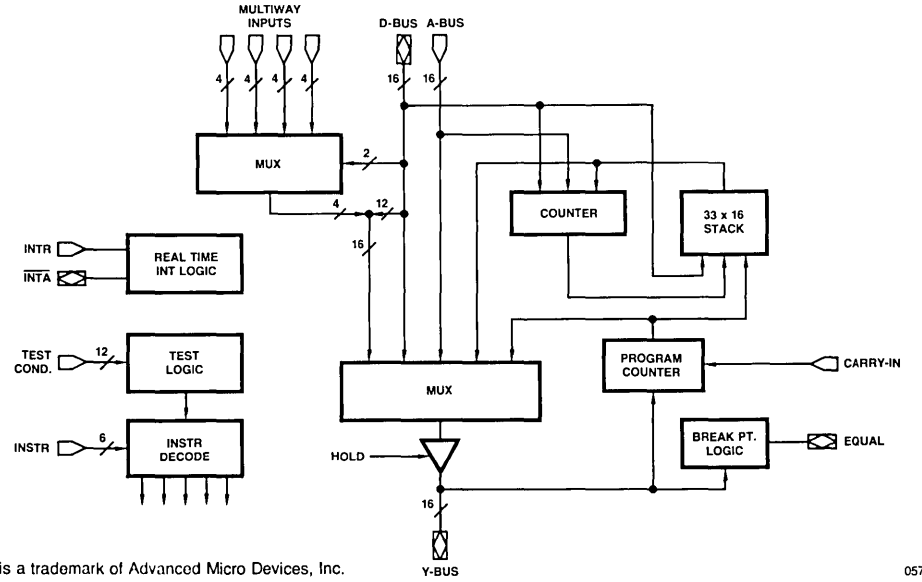
GENERAL DESCRIPTION

The Am29331 is a 16-bit wide high-speed single chip sequencer designed to control the execution sequence of microinstruction stored in the microprogram memory. The instruction set is designed to resemble high-level language constructs, thereby bringing high-level language programming to the micro level.

The Am29331 is interruptible at any microinstruction boundary to support real-time interrupts. Interrupts are handled transparently to the microprogrammer as an unexpected procedure call. Traps are also handled transparently at any microinstruction boundary. This feature allows re-execution of a prior microinstruction. Two separate buses are provided to bring a branch address directly into the chip from two sources to avoid slow turn-on and turn-off times for different sources connected to the data input bus. Four sets of multiway inputs are also provided to avoid slow turn-on and turn-off times for different branch address sources. This feature allows implementation of table look-up or use of external conditions as part of a branch address. The thirty-three deep stack provides the ability to support interrupts, loops and subroutine nesting. The stack can be read through the D-bus to support diagnostics or to implement multi-tasking at the micro-architecture level. The master/slave mode provides a complete function check capability for the device.

The Am29331 is designed with the IMOX™ process which allows internal ECL circuits with TTL-compatible I/O. It is housed in a 120-lead pin-grid-array package.

SIMPLIFIED BLOCK DIAGRAM



IMOX is a trademark of Advanced Micro Devices, Inc.

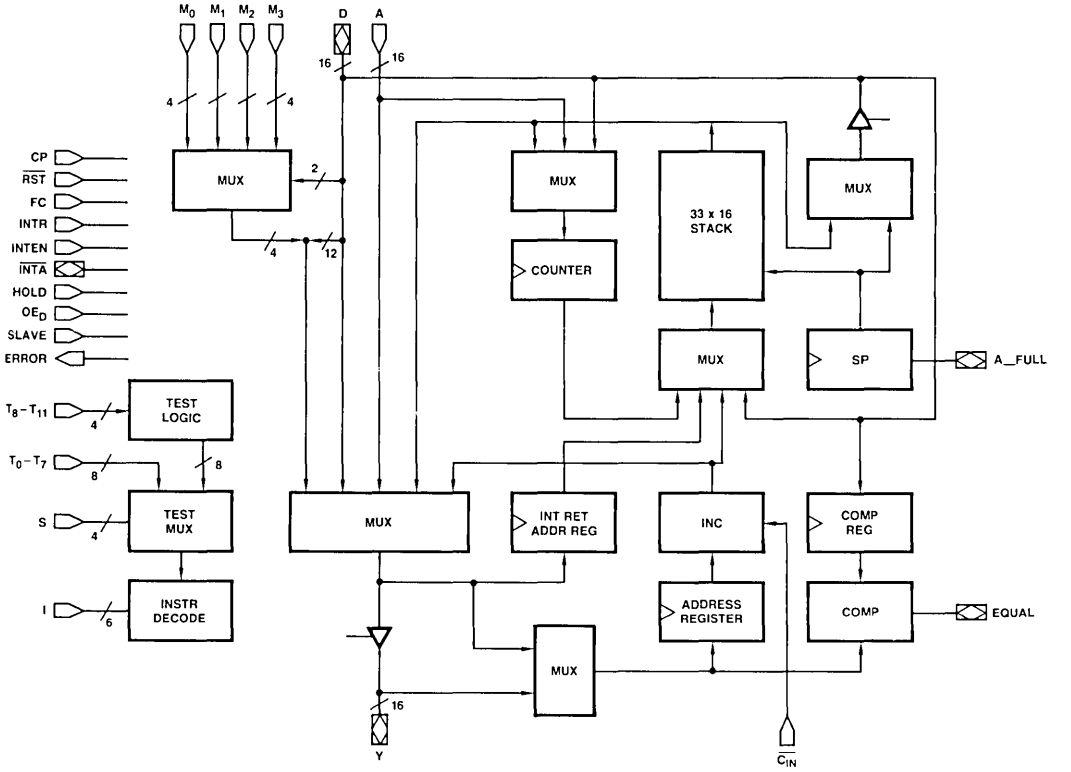
05729B-1

Order # 05729B

RELATED PRODUCTS

Part No.	Description
Am29323	32 x 32 Parallel Multiplier
Am29325	32-Bit Floating Point Processor
Am29332	32-Bit Extended Function ALU
Am29334	64 x 18 Four Port, Dual Access Register File

Figure 1. Am29331 Block Diagram



05729B-2

PIN DESCRIPTION

D_0-D_{15}	**Data, Bidirectional, Three-State** Input to address multiplexer, counter, stack, and comparator register. Output for stack and stack pointer.
A_0-A_{15}	**Alternate Data, Input** Input to address multiplexer and counter.
M_0-3, M_{0-3}	**Multiway, Input** Four sets of multiway inputs providing 16-way branches. The first index refers to the set number.
Y_0-Y_5	**Address, Bidirectional, Three-State** Output of microcode address. Input for interrupt address.
I_0-I_{15}	**Instruction, Input** Selects one of 64 instructions.
T_0-T_{11}	**Test, Input** Provides external test inputs.
S_0-S_3	**Select, Input** Selects one of 16 test conditions.
CP	**Clock Pulse, Input** Clocks sequencer at the low to high transition.
$\overline{RST}$	**Reset, Input** Resets the sequencer.
FC	**Force Continue, Input** Overrides instruction with CONTINUE.
INTR	**Interrupt Request, Input** Requests the sequencer to interrupt execution.
INTEN	**Interrupt Enable, Input** Enables interrupts.
$\overline{INTA}$	**Interrupt Acknowledge, Bidirectional, Three-State, Active Low** Indicates that an interrupt is accepted.
HOLD	**Input** Stops the sequencer and three-states the outputs.
OE_D	**Output Enable D-Bus, Input** Enables the D-bus driver provided the sequencer is not in the hold or slave mode.
SLAVE	**Input** Makes the sequencer a slave.
ERROR	**Output, Three-State** Indicates a Master/Slave error in the slave mode. Indicates a malfunctioning driver or contention in the master mode.
$\overline{C_{IN}}$	**Input, Active Low** Carry-in to incrementer
A-FULL	**Almost Full, Bidirectional, Three-State** Indicates that SP $\geqslant$ 28.
EQUAL	**Bidirectional, Three-State** Indicates that the address comparator is enabled and has found a match.

ARCHITECTURE

The major blocks of the sequencer are the address multiplexer, the microprogram counter (PC), the stack (with the top of stack denoted TOS), the counter (C), the test multiplexer with logic, and the address comparison register (R), (Figure 1). The bidirectional D-bus provides branch addresses and iteration counts; it also allows access to the stack from outside. The A-bus may be used for map addresses. There are four sets of four-bit multiway branch inputs (M). The bidirectional Y-bus either outputs microprogram addresses or inputs interrupt addresses. The buses are all 16 bits wide. Figure 1 shows a block diagram of the sequencer.

ADDRESS MULTIPLEXER

The address multiplexer can select an address from any of five sources:

1) A branch address supplied by the D-bus.
2) A branch address supplied by the A-bus.
3) A multiway branch address.
4) A return or loop address from the top of stack.
5) The next sequential address from the incrementer.

MULTIWAY BRANCH ADDRESS

A multiway branch address is formed by substituting the lower four bits of the address on the D-bus ($D_3D_2D_1D_0$) with one of the four sets (M_0, M_1, M_2 or M_3) of four-bit multiway branch addresses. The multiway branch set is selected by the number D_1D_0, while the bits D_3 and D_2 are don't cares.

ADDRESS REGISTER

The address register contains the current address. It is loaded from the interrupt multiplexer and feeds the incrementer. The incrementer is inhibited if $\overline{C_{IN}}$ is taken HIGH.

STACK

A 33-word deep and 16-bit wide stack provides first-in last-out storage for return addresses, loop addresses, and counter values. Items to be pushed come from the incrementer, the interrupt return address register, the counter, or the D-bus. Items popped go to the address multiplexer, the counter, or the D-bus.

The access to the stack via the D-bus may be used for context switching, stack extension, or diagnostics. As the stack is only accessible from the top, stack extension is done by temporarily storing the whole or some lower part of the stack outside the sequencer. The save and the later restore are done with pop and push operations respectively at balanced points in the microprogram, i.e., points with the same stack depth. The internal D-bus driver must be turned on when popping an item to the D-bus; if the driver is off, the item will be unstacked instead. The driver is normally turned on when the signal Output Enable is asserted and the sequencer is not being reset (OE_D = 1, $\overline{RST}$ = 1).

The stack pointer is a module 64 counter, which is incremented on each push and decremented on each pop. The stack pointer is reset to zero when the sequencer is reset, but the pointer may also be reset by instruction. Thus, the stack pointer indicates the number of items on the stack as long as stack overflow or underflow has not occurred. Overflow happens when an item is pushed onto a full stack, whereby the item at the bottom of the stack is overwritten. Underflow happens when an item is popped from an empty stack, in this case the item is undefined.

The contents of the stack pointer is present on the D-bus for all instructions except POP D, provided the driver is turned on. The output signal A-FULL is defined as SP $\geq$ 28.

COUNTER

The counter may be used as a loop counter. It may be loaded from the D-bus, the A-bus or via a pop from the stack. Its contents may also be pushed onto the stack.

A normal for-loop is set up by a FOR instruction, which loads the counter from the D- or A-bus with the desired number of iterations; the instruction also pushes onto the stack a loop address, that points to the next sequential instruction. The end of the loop is given by an unconditional END FOR instruction, which tests the counter value against the value one and then decrements the counter. If the values differ, the loop is repeated by selecting the address at the stack as the next address. If the values are equal, the loop is terminated by popping the stack, thereby removing the loop address, and selecting the address from the incrementer as the next address. The number of iterations is a 16-bit unsigned number, except that the number zero corresponds to 65536 iterations. By pushing and popping counter values it is possible to handle nested loops.

ADDRESS COMPARISON

The sequencer is able to compare the address from the interrupt multiplexer with the contents of the comparator register. The instruction SET loads the comparator register with the address on the D-bus and enables the comparison, while CLEAR disables it. The comparison is disabled at reset. A HIGH is present at the output EQUAL if the comparison is useful for detection of a breakpoint or counting how often a microinstruction at a specific address is executed.

INSTRUCTION SET

The sequencer has 64 instructions that are divided into four classes of 16 instructions each. The instruction lines I_0-I_5 use I_5 and I_4 to select a class and I_0-I_3 to select an instruction within a class. The classes are:

I_5	I_4	
0	0	Conditional sequence control,
0	1	Conditional sequence control with inverted polarity,
1	0	Unconditional sequence control, and
1	1	Special function with implicit continue.

Note that for the first three classes I_5 forces the condition to be true and I_4 inverts the condition. The basic instructions of the first three classes are shown in Table 1 and the instructions of the fourth class in Table 2.

Structured microprogramming is supported by sequencer instructions that singly or in pairs correspond to high-level language control constructs. Examples are FOR I: = D DOWN TO 1 DO . . . END FOR and CASE N OF . . . END CASE. The instructions have been given high-level language names where appropriate. Figure 2 shows how to microprogram important control constructs; the high-level language is on the left and the microcode on the right.

TEST CONDITIONS

The condition for a conditional instruction is supplied by a test multiplexer, which selects one out of sixteen tests with the select lines S_0-S_3. Twelve of these are supplied directly by the inputs T_0-T_{11}, while the remaining four tests are generated by the test logic from the inputs T_8-T_{11}. The following table shows the assignments.

S	Test	Intended Use
0–7	T_0–T_7	General
8	T_8	C (Carry)
9	T_9	N (Negative)
10	T_{10}	V (Overflow)
11	T_{11}	Z (Zero or equal)
12	$T_8 + T_{11}$	$\overline{C}$ + Z (Unsigned less than or equal, borrow mode)
13	$\overline{T_8} + T_{11}$	C + Z (Unsigned less than or equal)
14	$T_9 \oplus T_{10}$	N ⊕ V (Signed less than)
15	$(T_9 \oplus T_{10}) + T_{11}$	(N ⊕ V) + Z (Signed less than or equal)

FORCE CONTINUE

The sequencer has a force continue (FC) input, which overrides the instruction inputs $I_0 - I_5$ with a CONTINUE instruction. This makes it possible to share the microinstruction field for the sequencer instruction with some other control or to initialize a writable control store.

RESET

In order to start a microprogram properly the sequencer must be reset. The reset works like an instruction overriding both the instruction input and the force continue input. The reset selects the address 0 at the address multiplexer, forces the EQUAL output to LOW, and disregards a potential interrupt request. It synchronously disables the address comparison and initializes the stack pointer to 0.

TABLE 1

I_0–I_3	Instruction	Cond.: False Y	Cond.: False Stack	Cond.: True Y	Cond.: True Stack	Counter	Comp.	D-Mux
0	Goto D	PC	–	D	–	–	–	SP
1	Call D	PC	–	D	Push PC	–	–	SP
2	Exit D	PC	–	D	Pop	–	–	SP
3	End for D, C ≠ 1	PC	–	D	–	C←C – 1	–	SP
	End for D, C = 1	PC	–	PC	–	C←C – 1	–	SP
4	Goto A	PC	–	A	–	–	–	SP
5	Call A	PC	–	A	Push PC	–	–	SP
6	Exit A	PC	–	A	Pop	–	–	SP
7	End for A, C ≠ 1	PC	–	A	–	C←C – 1	–	SP
	End for A, C = 1	PC	–	PC	–	C←C – 1	–	SP
8	Goto M	PC	–	D:M	–	–	–	SP
9	Call M	PC	–	D:M	Push PC	–	–	SP
10	Exit M	PC	–	D:M	Pop	–	–	SP
11	End for M, C ≠ 1	PC	–	D:M	–	C←C – 1	–	SP
	End for M, C = 1	PC	–	PC	–	C←C – 1	–	SP
12	End Loop	PC	Pop	TOS	–	–	–	SP
13	Call Coroutine	PC	–	TOS	TOS←PC	–	–	SP
14	Return	PC	–	TOS	Pop	–	–	SP
15	End for, C ≠ 1	PC	Pop	TOS	–	C←C – 1	–	SP
	End for, C = 1	PC	Pop	PC	Pop	C←C – 1	–	SP

Cond. = (Test[S] or I_5) XOR I_4
: = Concatination
C = Counter

TABLE 2

I_0–I_3	Instruction	Y	Stack	Counter	Comp.	D-Mux
0	Continue	PC	–	–	–	SP
1	For D	PC	Push PC	C←D	–	SP
2	Decrement	PC	–	C←C – 1	–	SP
3	Loop	PC	Push PC	–	–	SP
4	Pop D	PC	Pop	–	–	TOS
5	Push D	PC	Push D	–	–	SP
6	Reset SP	PC	SP←0	–	–	SP
7	For A	PC	Push PC	C←A	–	SP
8	Pop C	PC	Pop	C←TOS	–	SP
9	Push C	PC	Push C	–	–	SP
10	Swap	PC	TOS←C	C←TOS	–	SP
11	Push C Load D	PC	Push C	C←D	–	SP
12	Load D	PC	–	C←D	–	SP
13	Load A	PC	–	C←A	–	SP
14	Set	PC	–	–	R←D, Enable	SP
15	Clear	PC	–	–	Disable	SP

I_4 = I_5 = HIGH; R = Comp. Register

INTERRUPTS

The sequencer may be interrupted at the completion of the current microcycle by asserting the interrupt request input INTR. The return address of the interrupted routine is saved on the stack; nested interrupts are allowed. An interrupt is accepted if interrupts are enabled and the sequencer is not being reset or held (INTEN = HIGH, RESET = LOW, and HOLD = LOW).

When there is no interrupt, addresses go from the address multiplexer to the Y-bus via the driver and to the incrementer and the comparator via the interrupt multiplexer. When there is an interrupt, the driver of the sequencer is turned off, an external driver is turned on, and the interrupt multiplexer is switched. The interrupt address is supplied via the external driver to the Y-bus and the incrementer and the comparator. In order to save the address from the address multiplexer, the address is stored in the interrupt address register, which for simplicity is clocked every cycle. The next microinstruction is the first microinstruction of the interrupt routine.

In this cycle the address in the interrupt return address register is automatically pushed onto the stack. Therefore the microinstruction in this cycle must not use the stack; if a stack operation is programmed, the result is undefined. The instructions that do not use the stack are GOTO D, GOTO A, GOTO M, CONTINUE, DECREMENT, LOAD D, LOAD A, SET and CLEAR. A RETURN instruction terminates the interrupt routine and the interrupted routine is resumed. Interrupts only work with a single-level control path.

TRAPS

A trap is an unexpected situation linked to the current microinstruction, that must be handled before the microinstruction completes and changes the state of the system. An example of such a situation is an attempt to read a word from memory across a word boundary in a single cycle. When a trap occurs, the current microinstruction must be aborted and re-executed after the execution of a trap routine, which in the meantime will take corrective measures. An interrupt, on the other hand, is not linked directly to the current microinstruction that can complete safely before an interrupt routine is executed.

Execution of a trap requires that the sequencer ignores the current microinstruction, selects the trap return address at the address multiplexer, and initiates an interrupt. This will save the trap return address on the stack and issue the trap address from an external source. The address register contains the address of the microinstruction in the pipeline register, thus the address register already contains the trap return address when a trap occurs. This address can be selected by the address multiplexer by disabling the incrementer ($\overline{C_{IN}}$ = 1), and using the force continue mode (FC = 1). In this mode the sequencer ignores the current microinstruction. The remaining part of the trap handling is done by the interrupt. Thus the section on interrupts also applies to traps. There is one exception, however. The interrupt enable cannot be used as a trap enable as it does not control the force continue mode and the carry-in to the incrementer.

HOLD MODE

The sequencer has a hold mode in which operation is suspended.

When the HOLD signal goes active, the incrementer and the outputs (except the D-bus) are disabled and the sequencer enters the hold mode after the current cycle. While the sequencer is in this mode, the internal state is left unchanged and the D-bus is disabled. When the HOLD signal goes inactive the incrementer and the outputs (except the D-bus) are enabled again and the sequencer leaves the hold mode after that cycle.

In a time multiplexed multi-microprocess system there may be one sequencer for all processes with microprogrammed context save and restore, or there may be one sequencer per microprocess permitting fast process switch. In the latter case the Y-buses of the sequencers are tied together and connected to a single microprogram store. A control unit decides on a cycle by cycle basis, what sequencer should be running and activates the HOLD signal to the remaining sequencers. The hold mode has higher priority than interrupts, and works independently of the RESET signal. The hold mode can only be used with a single-level control path.

MASTER/SLAVE CONFIGURATION

In some systems reliability is very important. The master/slave configuration, that consists of two sequencers operated in parallel, is able to detect faults in both the interrconnect and the internal function of the sequencers. One sequencer is the master and operates normally. The other is a slave, i.e., all outputs except the signal ERROR are turned into inputs and connected to the outputs of the master. Since the slave is operated in parallel with the master, it can compare its result with the result of the master and signal an error if they differ. The error signal from the master indicates a malfunctioning driver or contention.

Figure 2A

Loops with unknown number of iterations:

```
REPEAT                LOOP
 –                     –
 –                     –
UNTIL CC              END LOOP NOT CC

WHILE CC DO           LOOP
                      IF NOT CC THEN EXIT L
 –                     –
 –                     –
END WHILE             END LOOP
                      L:

LOOP                  LOOP
 –                     –
IF CC THEN EXIT       IF CC THEN EXIT L
                       –
END LOOP              END LOOP
                      L:
```

Figure 2B

Loop with known number of iterations:

```
FOR CNT: = 10 DOWN TO 1 DO    FOR D 10
 –                             –
 –                             –
END FOR                       END FOR
```

Figure 2C

Case Statment,
with D = $A_{15} \ldots A_4XX00$ and $M_{0,\ 0-3} = A_3I_1I_00$ during the GOTO M instruction. $A_2A_1A_0$ must be 000, and X signifies a don't care.

```
                      PUSH D B
CASE I OF             GOTO M
0: –                  A:     –
   –                         –, RETURN TO B
1: –                  A + 2: –
   –                         –, RETURN TO B
2: –                  A + 4: –
   –                         –, RETURN TO B
3: –                  A + 6: –
   –                         –, RETURN
END CASE              B:
```

Figure 2D

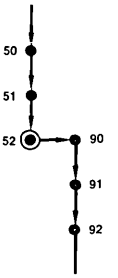
Double nested if-statement:

```
                      PUSH D C
IF X THEN             IF NOT X THEN GOTO A
 IF Y THEN             IF NOT Y THEN GOTO B
  –                     –
  –                     –, RETURN TO C
 ELSE                  B:
  –                     –
  –                     –, RETURN TO C
 END IF
ELSE                  A:
 IF Z THEN             IF NOT Z THEN GOTO D
  –                     –
  –                     –, RETURN TO D
 ELSE                  D:
  –                     –
  –                     –, RETURN TO C
 END IF
END IF                C:
```

INSTRUCTION SET DEFINITION

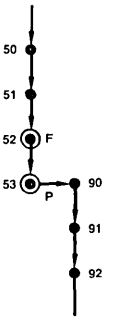
Legend: ● = Other instruction
◉ = Instruction being described
P = Test pass
F = Test fail
◯ = Register in part

Opcode ($I_5 - I_0$)	Mnemonics	Description
32	BRA__D	Go to D. Unconditional branch to the address specified by the D inputs.
36	BRA__A	Go to A. Unconditional branch to the address specified by the A inputs.
40	BRA__M	Go to M. Unconditional branch to the address specified by the D inputs catenated with the multiway M inputs.
44	BRA__S	Go to TOS. Unconditional branch to the address on the top of the stack. Also End Loop when used to terminate WHILE . . . ENDWHILE loops.



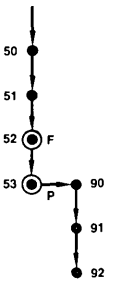
05729B-3

0	BRCC__D	If CC is HIGH then branch to the address specified by the D inputs else continue.
4	PRCC__A	If CC is HIGH then branch to the address specified by the A inputs else continue.
8	BRCC__M	If CC is HIGH then branch to the address specified by the D inputs catenated with the multiway M inputs else continue.
12	BRCC__S	If CC is HIGH then branch to the address on the top of the stack else pop the stack and continue. Also End Loop when used to terminate REPEAT . . . UNTIL loops.



05729B-4

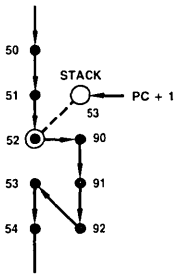
16	BRNC__D	If CC is LOW then branch to the address specified by the D inputs else continue.
20	BRNC__A	If CC is LOW then branch to the address specified by the A inputs else continue.
24	BRNC__M	If CC is LOW then branch to the address specified by the D inputs catenated with the multiway M inputs else continue.
28	BRNC__S	If CC is LOW then branch to the address on the top of the stack else pop the stack and continue. Also End Loop when used to terminate REPEAT . . . UNTIL loops.



Note: Opcode numbers are in decimal notation.

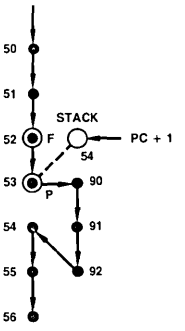
05729B-5

Opcode ($I_5 - I_0$)	Mnemonics	Description
33	CALL_D	Call D. Unconditional branch to the subroutine address specified by the D inputs and push the PC on the stack.
37	CALL_A	Call A. Unconditional branch to the subroutine address specified by the A inputs and push the PC on the stack.
41	CALL_M	Call M. Unconditional branch to the subroutine address specified by the D inputs catenated with the multiway M inputs and push the PC on the stack.
45	CALL_S	Call TOS. Exchange PC and TOS. Also call coroutine.



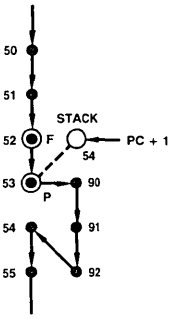
05729B-6

1	CCC_D	If CC is HIGH then call the subroutine address specified by the D inputs else continue.
5	CCC_A	If CC is HIGH then call the subroutine address specified by the A inputs else continue.
9	CCC_M	If CC is HIGH then call the subroutine address specified by the D inputs catenated with the multiway M inputs else continue.
13	CCC_S	If CC is HIGH then call the address on the top of the stack else continue. Also used for conditional coroutine calls.



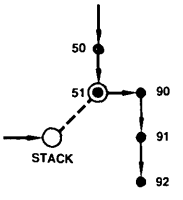
05729B-7

17	CNC_D	If CC is LOW then call the address specified by the D inputs else continue.
21	CNC_A	If CC is LOW then call the address specified by the A inputs else continue.
25	CNC_M	If CC is LOW then call the address specified by the D inputs catenated with the multiway M inputs else continue.
29	CNC_S	If CC is LOW then call the address on the top of the stack else continue. Also a conditional coroutine call.



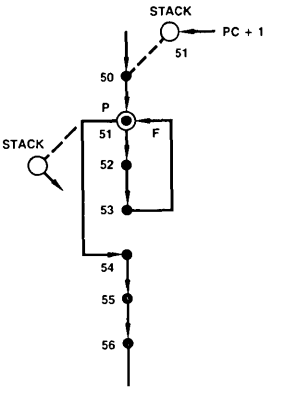
05729B-8

34	EXIT_D	Exit to D. Unconditional branch to the address specified by the D inputs and pop the stack.
38	EXIT_A	Exit to A. Unconditional branch to the address specified by the A inputs and pop the stack.
42	EXIT_M	Exit to M. Unconditional branch to the address specified by the D inputs catenated with the multiway M inputs and pop the stack.



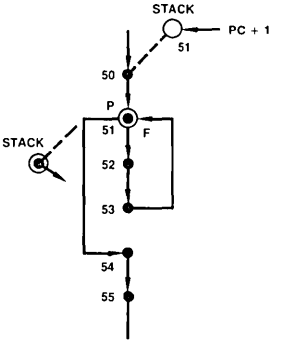
05729B-9

Opcode ($I_5 - I_0$)	Mnemonics	Description
2	XTCC__D	If CC is HIGH then exit to the address specified by the D inputs and pop the stack else continue with no pop.
6	XTCC__A	If CC is HIGH then exit to the address specified by the A inputs and pop the stack else continue with no pop.
10	XTCC__M	If CC is HIGH then exit to the address specified by the D inputs catenated to the multiway M inputs and pop the stack else continue with no pop.



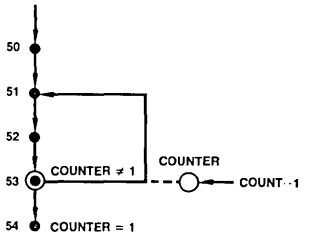
05729B-10

18	XTNC__D	If CC is LOW then exit to the address specified by the D inputs and pop the stack else continue.
22	XTNC__A	If CC is LOW then exit to the address specified by the A inputs and pop the stack else continue.
26	XTNC__M	If CC is LOW then exit to the address specified by the D inputs catenated with the multiway M inputs and pop the stack else continue.



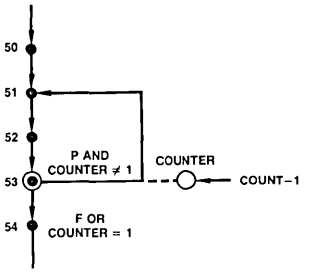
05729B-11

35	DJMP__D	If the counter is not equal to one then decrement the counter and branch to the address specified by the D inputs else continue.
39	DJMP__A	If the counter is not equal to one then decrement the counter and branch to the address specified by the A inputs else continue.
43	DJMP__M	If the counter is not equal to one then decrement the counter and branch to the address specified by the D inputs catenated with the multiway M inputs else continue.
47	DJMP__S	If the counter is not equal to one then decrement the counter and branch to the address on the top of the stack else decrement the counter, pop the stack and continue.



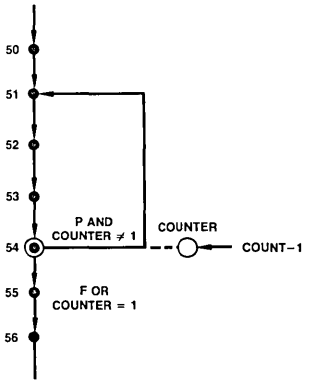
05729B-12

Opcode ($I_5 - I_0$)	Mnemonics	Description
3	DJCC__D	If CC is HIGH and the counter is not equal to one then decrement the counter and branch to the address specified by the D inputs else decrement the counter and continue.
7	DJCC__A	If CC is HIGH and the counter is not equal to one then decrement the counter and branch to the address specified by the A inputs else decrement the counter and continue.
11	DJCC__M	If CC is HIGH and the counter is not equal to one then decrement the counter and branch to the address specified by the D inputs catenated with the multiway M inputs else decrement the counter and continue.
15	DJCC__S	If CC is HIGH and the counter is not equal to one then decrement the counter and branch to the address on the top of the stack else decrement the counter, pop the stack and continue.



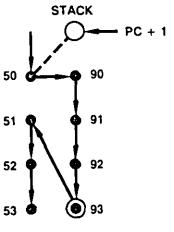
05729B-13

19	DJNCC__D	If CC is LOW and the counter is not equal to one then decrement the counter and branch to the address specified by the D inputs else decrement the counter and continue.
23	DJNCC__A	If CC is LOW and the counter is not equal to one then decrement the counter and branch to the address specified by the A inputs else decrement the counter and continue.
27	DJNCC__M	If CC is LOW and the counter is not equal to one then decrement the counter and branch to the address specified by the D inputs catenated with the multiway M inputs else decrement the counter and continue.
31	DJNCC__S	If CC is LOW and the counter is not equal to one then decrement the counter and branch to the address on the top of the stack else decrement the counter, pop the stack and continue.



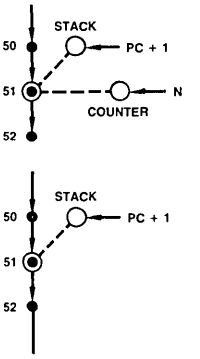
05729B-14

46	RET	Unconditional return from subroutine.
14	RETCC	If CC is HIGH then return from subroutine else continue.
30	RETNC	If CC is LOW then return from subroutine else continue.



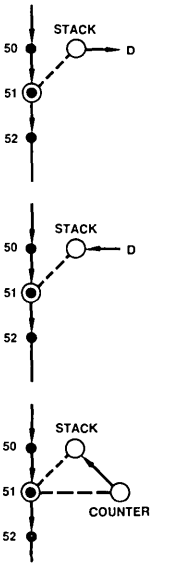
05729B-15

Opcode ($I_5 - I_0$)	Mnemonics	Description
49	FOR_D	Initialize loop. Push the PC on the stack, load the counter with the value of the D inputs and continue. Use with DJMP_S for FOR . . . NEXT loops.
55	FOR_A	Initialize loop. Push the PC on the stack, load the counter with the value of the A inputs and continue. Use with DJMP_S for FOR . . . NEXT loops.
51	LOOP	Initialize loop. Push the PC and continue. Use with BRCC_S for REPEAT . . . UNTIL loops or with XTCC_D and BRA_S for WHILE . . . ENDWHILE loops.



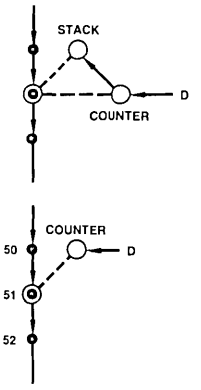
05729B-16

Opcode ($I_5 - I_0$)	Mnemonics	Description
52	POP_D	Pop the stack, output the value on the D outputs and continue.
56	POP_C	Pop the stack, place the value in the counter and continue.
53	PUSH_D	Push the D inputs on the stack and continue.
57	PUSH_C	Push the counter on the stack and continue.
58	SWAP	Exchange the counter and the top of stack and continue.



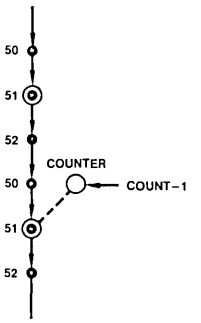
05729B-17

Opcode ($I_5 - I_0$)	Mnemonics	Description
59	STACK_C	Push the counter on the stack, load the counter with the value of the D inputs and continue.
60	LOAD_D	Load the counter with the value of the D inputs and continue.
61	LOAD_A	Load the counter with the value of the A inputs and continue.



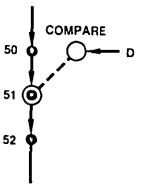
05729B-18

48	CONT	Continue.
50	DECR	Decrement the counter and continue.
54	RESET_SP	Reset the stack pointer and continue.



05729B-19

62	SET	Load the comparison register with the value of the D inputs, enable the comparator and continue.
63	CLEAR	Disable the comparator and continue.



05729B-20

Am29332

32-Bit Arithmetic Logic Unit

ADVANCED INFORMATION

DISTINCTIVE CHARACTERISTICS

- **Single Chip, 32-Bit ALU**
  Supports 80–90ns microcycle time for the 32-bit data path. It is a combinatorial ALU with equal cycle time for all instructions.
- **Flow-through Architecture**
  A combinatorial ALU with two input data ports and one output data port allows implementation of either parallel or pipelined architectures.
- **64-Bit In, 32-Bit Out Funnel Shifter**
  This unique functional block allows n-bit shift-up, shift-down, 32-bit barrel shift or 32-bit field extract.
- **Supports All Data Types**
  It supports one-, two-, three- and four-byte data for all operations and variable-length fields for logical operations.
- **Multiply and Divide Support**
  Built-in hardware to support two-bit-at-a-time modified Booth's algorithm and one-bit-at-a-time division algorithm.
- **Extensive Error Checking**
  Parity check and generate provides data transmission check and master/slave mode provides complete function checking.

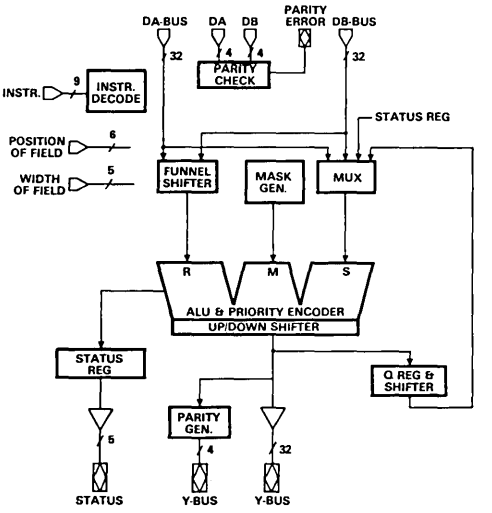
GENERAL DESCRIPTION

The Am29332 is a 32-bit wide non-cascadable Arithmetic Logic Unit (ALU) with integration of functions that normally don't cascade, such as barrel shifters, priority encoders and mask generators. Two input data ports and one output data port provide flow-through architecture and allow the designer to implement his/her architecture with any degree of pipelining and no built-in penalties for branching. Also, the simplicity of a three-bus ALU allows easy implementation of parallel or reconfigurable architectures. The register file is off-chip to allow unlimited expansion and regular addressability.

The Am29332 supports one-, two-, three- and four-byte data for arithmetic and logic operations. It also supports multiprecision arithmetic and shift operations. For logical operations, it can support variable-length fields up to 32 bits. When fewer than four bytes are selected, unselected bits are passed to the destination without modification. The device also supports two-bit-at-a-time modified Booth's algorithm for high-speed multiplication and one-bit-at-a-time division. Both signed and unsigned integers for all byte aligned data types mentioned above are supported.

The Am29332 is designed to support 80–90 ns microcycle time. The device is packaged in a 168-lead pin-grid-array package.

SIMPLIFIED BLOCK DIAGRAM



BD005240

Order # 05730B

RELATED PRODUCTS

Part No.	Description
Am29323	32 x 32 Parallel Multiplier
Am29325	32-Bit Floating Point Processor
Am29331	16-Bit Microprogram Sequencer
Am29334	64 x 18 Four-Port, Dual-Access Register File

PIN DESCRIPTION

$PA_3 - PA_0$ Parity input for operand A on DA-bus (one per byte).

$DA_{31} - DA_0$ Data input lines for operand A.

$PB_3 - PB_0$ Parity input for operand B on DB-bus (one per byte).

$DB_{31} - DB_0$ Data input lines for operand B.

$PY_3 - PY_0$ Parity output for data on Y-bus (one per byte).

$Y_{31} - Y_0$ **Data Input/Output Lines**
When $\overline{OE}$-Y is LOW and the ALU is in the Master mode, the ALU result is enabled on the Y-bus. When $\overline{OE}$-Y is HIGH, the Y-bus is tristated. In Slave mode the Y-bus acts as external data input.

$I_6 - I_0$ **Instruction Inputs**

$I_8 - I_7$ Byte width inputs for byte boundary aligned operand instructions. Selects the sources for width and position inputs for variable field bit operands. If I_7 is LOW it selects the width input from pins $W_4 - W_0$. If I_7 is HIGH the width input is selected from the internal width register. Similarly if I_8 is LOW it selects the position inputs from pins $P_5 - P_0$ and if HIGH it selects input from the internal position register.

$W_4 - W_0$ Width input to select the width of a contiguous bit field.

$P_5 - P_0$ Position input to select the position of the least significant bit of a field. Also indicates the amount by which data is to be shifted up (P_5 = LOW) or down (P_1 = HIGH) or rotated.

C, Z, N, V, L When the Register Status pin is LOW, these pins give the carry, zero, negative, overflow and link outputs of the ALU where applicable to the instruction being executed. When not applicable to the instruction being executed, or when the Register Status pin is HIGH, these pins give the outputs of the carry, zero, negative, overflow and link bits of the internal status register. In SLAVE mode, C, Z, N, V and L become inputs.

Register Status **Register Status Mode Pin**
Selects between ALU status (Register Status = LOW) or register status (Register Status = HIGH) on the C, Z, N, V and L outputs.

HOLD When HIGH it inhibits the update of the status and Q registers.

CP Clocks internal registers (status, Q) at the LOW to HIGH transition, provided HOLD input is LOW.

$\overline{OE}$-Y **Output Enable**
When $\overline{OE}$-Y is HIGH the Y-bus is disabled (tri-stated).

Borrow When HIGH the Carry In and Carry Out are borrows for subtract operations.

Macro Carry **Macro Status Carry Input**

Macro Link **Macro Status Link Input**

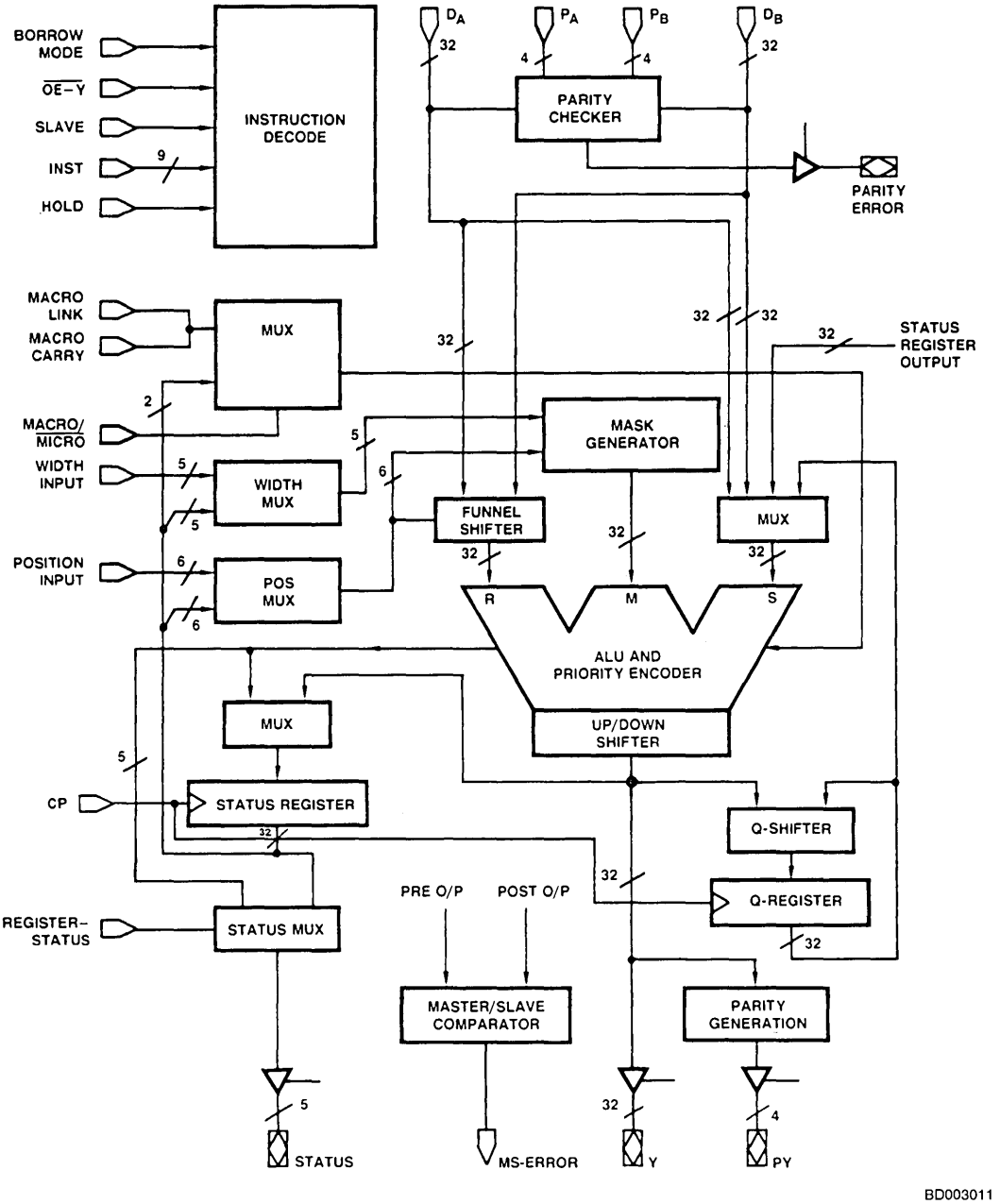
Macro/Micro SEL When HIGH selects macro carry and macro link pins as input instead of micro carry and micro link from the micro-status register.

Slave When HIGH this pin puts the ALU in the slave mode. All output pins become input pins and signals on them are compared with the ALU's internally generated results. When $\overline{OE}$-Y is HIGH, the $Y_0 - Y_{31}$ and $PY_0 - PY_3$ inputs are ignored. When the SLAVE pin is LOW, the ALU is put in master mode where outputs are generated as normal.

MS-Error **Master-Slave Error**
When HIGH this signal indicates that the master's and slave's data were not identical.

Parity-Error When HIGH indicates that a parity error was detected on the DA or DB inputs.

BLOCK DIAGRAM



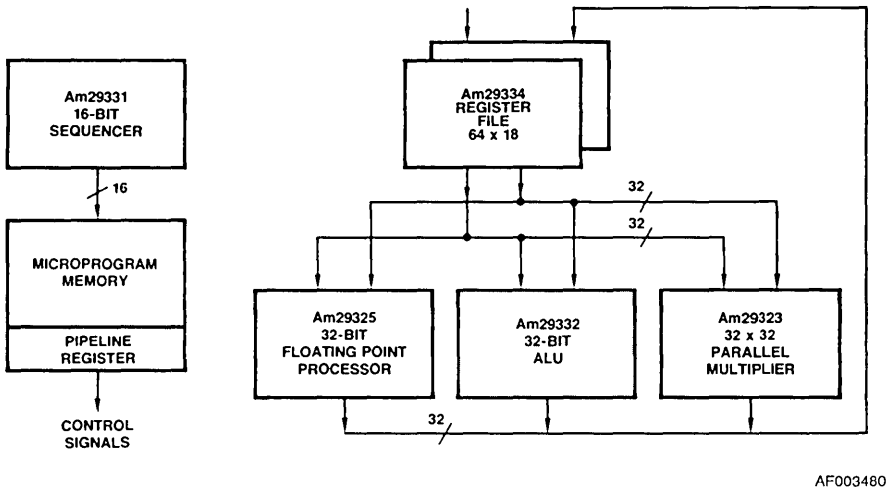


Figure 1. Am29332 Family High Performance System Block Diagram

PRODUCT OVERVIEW

The Am29332 is a 32-bit wide, high performance, non-expandable Arithmetic Logic Unit. It has two 32-bit wide input ports (A and B) and one 32-bit wide output port (Y). These three ports provide flexibility and accessibility for high-performance processor designs. Dedicated input and output ports provide a flow-through architecture and avoid the penalty associated with switching the bus half-way thrcugh the cycle for input and output of data. The chip is designed for use with a dual access RAM (Am29334) as a register file. In addition, the three bus architecture facilitates the connection of other arithmetic units in parallel with the Am29332 for high performance systems.

The Am29332 supports one-, two-, three- and four-byte arithmetic operations. It also supports multiprecision arithmetic and multiple bit shifts. For logical operations, it can handle variable-length fields of up to 32 bits. The chip incorporates dedicated hardware to allow efficient implementation of a two bit-at-a-time (modified Booth) multiply algorithm, supporting signed and unsigned arithmetic data types. Similarly, hardware is provided to support a bit-at-a-time divide algorithm, also supporting signed and unsigned arithmetic data types. An internal 32-bit register (Q) is used by the multiply and divide hardware for double precision operands. For business applications, the Am29332 supports variable-length BCD arithmetic.

Field logical instructions operate on bit-fields taken from the A and B data inputs; they may be of variable width and starting position. A is normally the source input and B the destination input. In general, destination bits not falling within a specified field are passed by the ALU unchanged. Field width and position are specified either by direct inputs to the chip, or by entries in the status register. There are two kinds of field logical instructions – aligned and non-aligned. The first type of instruction assumes that source and destination fields are aligned and the operation is performed only for bits within the specified fields. In the second type of instruction, source and destination fields are normally non-aligned. However, it is always assumed that one field (either source or destination) is least significant bit (LSB) aligned.

If the destination field is LSB aligned then the source field is downshifted in order to make it LSB aligned as well. Downshifting is accomplished by making the 6-bit position input equal to the two's complement of the number of places the field is to be downshifted. If the source field is LSB aligned then it is upshifted in order to align it with the destination. Upshifting is accomplished by making the position inputs equal to the number of places the field is to be upshifted. Any other type of field operation is not allowed. Whenever the field crosses the word boundary, the portion not falling within the word boundary is ignored. This effect is useful when performing operations on fields that overlap two different words. Instructions to perform straightforward multiple-bit shifts (either up or down) are also provided. Additionally, it is possible to extract a bit-field from a word in one instruction, even if that field overlaps a word boundary.

The power and the flexibility of the processor comes partly from its ability to generate a mask to control the width of an operation for each instruction without any overhead. For all byte aligned instructions (three quarters of the instruction set), the mask is either 1, 2, 3 or 4 bytes wide and is generated from the byte width input ($I_8 - I_7$). For all field instructions the mask is of variable width and is generated from the position inputs ($P_5 - P_0$) and the width inputs ($W_4 - W_0$). Whenever the width of the operand is less than 32-bits, all unselected bits from the inputs of the ALU are passed to the output without any modification. Depending upon the instruction type, unselected bits are taken from different sources. For example in all single operand instructions, bits from the source operand (from either A or B input) are passed in unselected bit positions. For two operand instructions, bits from the B input are passed in unselected bit positions. There are some exceptions which are explained in the instruction set section.

The processor has a 32-bit status register to indicate the status of different operations performed. The status register is loaded at the rising edge of the clock with new status unless the HOLD signal is HIGH. The bit position for each status bit is given in the functional description. The least significant byte of the status register holds the six position bits ($P_5 - P_0$). The two most significant bits of this byte may be read or loaded but are otherwise unused by the ALU. The second byte (bits 8 to 15) consists of the five width bits ($W_4 - W_0$) and three read-only bits that are a combinational function of other status bits, and which indicate useful branch conditions. The third byte con-

sists of ALU status bits plus bits for high speed multiply and divide. The most significant byte holds intermediate nibble carries for BCD operations. An extract-status instruction is provided which allows a Boolean value to be formed from any selected bit. This is particularly useful in machines employing a stack architecture. Instructions to save and restore the status register are provided. As the entire status of each instruction is stored in the status register, interrupts at any microinstruction boundary are feasible.

The processor has a 32-bit wide priority encoder to support floating-point and graphics operations. The priority encoder supports all byte aligned data types – the result is dependent upon the byte width specified. The result of a priority encode is also loaded into the position bits of the status register. The result of the prioritize operation can then be used in the following clock cycle, e.g., to normalize a floating-point number or to help detect the edge of a polygon in graphics applications.

To support system diagnostics, the Am29332 has a special "Master-Slave" mode. To use this mode, two chips are connected in parallel, and hence receive the same instructions and data. The master chip is used for the normal data path. However, in the slave chip, all outputs becomes inputs. The slave compares the outputs of the master with its own internally generated result. If the two do not match, the slave will activate an error signal.

As a further diagnostic aid, byte-wise parity checking is performed at both the A and B data inputs. The "parity" signal is activated if an error is detected. Parity bits (one per byte) are generated for the 32-bit output bus.

FUNCTIONAL DESCRIPTION

A detail description of each functional block is given in the following paragraphs.

64-Bit Funnel Shifter

The 64-bit funnel shifter is a combinatorial network. The 64-bit input is formed from a combination of the A and B inputs. This may be left-shifted by up to 31 bits before being used by the ALU. The output of the shifter is the most significant 32 bits of the result. The 64-bit shifter can be used on either the A or B operands to perform barrel shifts (either up or down) or rotates. The operation is controlled by positioning operands properly at the input of the 64-bit up-shifter.

The number "n" by which operand is shifted comes from two sources: the microprogram memory via the $P_5 - P_0$ pins or the internal register (byte 0 of the status register), as selected by an instruction bit.

In general, the 6-bit position input, $P_5 - P_0$, takes a 6-bit two's complement number representing upshifts from 0 to 31 places (positive numbers) or downshifts from 1 to 32 places (negative numbers).

Mask Generator

The mask generator logic provides the ability to generate the appropriate mask for an operand of given width and position. The generation of the mask depends upon two types of instructions. The first type has byte boundary aligned operands (widths of either 1, 2, 3 or 4 bytes) with the least significant bit aligned to bit 0. The width of an operand is specified by the byte width inputs (I_8 and I_7) as shown in Table 1. The second type of instruction has operands of variable width (1 to 32 bits) and position. The operand is specified by the width inputs ($W_4 - W_0$) and the position inputs ($P_5 - P_0$) indicating the least significant bit position of the operand. Thus, in this type of instruction the operand may or may not be least significant bit aligned. Depending upon the type of instruction, the mask generator first generates a fence of all zeros starting from the least significant bit with the width specified either by the byte width or the width input fields. This fence can be upshifted by up to 31 bits by the 32-bit mask shifter. Whenever the mask is moved up over the 32-bit boundary, it does not wrap around. Instead, ONE's are inserted from the least significant end. This configuration provides the ability to operate on a contiguous field located anywhere in a word, or across a word boundary.

The mask generator can be used as a pattern generator by allowing the mask to pass through ALU (by using the PASS-MASK instruction). For example, a single-bit wide mask can be generated and by shifting it up by different amounts can give walking ONE or walking ZERO patterns for memory tests.

TABLE 1.

I_8	I_7	Width in Bytes
0	0	4
0	1	1
1	0	2
1	1	3

Arithmetic and Logical Unit

The ALU is a three input unit which uses the mask as a second or third operand in every instruction. The mask is used to merge two operands. For all selected bits (wherever the mask is 0), the desired operation specified by the instruction input is performed, and for all unselected bits either corresponding destination bits or zeros are passed through. The status of each operation (carry, negative, zero, overflow, link) applies to the result only over the specified width. For all byte aligned arithmetic and logical operations (first three quarters of the instruction set), the status is extracted from the appropriate byte boundary. For all field operations (last quarter of the instruction set), the operand width is assumed to be 32 bits for status generation. The ZERO flag always indicates the status of all bits selected by the mask.

The actual width of the ALU is 34 bits. There are two extra bits used for the high speed signed and unsigned multiplication instructions. These two bits are automatically concatenated to the most-significant end of the ALU depending upon the width specified for the operation. Since the modified Booth algorithm requires a two-bit down-shift each cycle, these ALU bits generate the two most-significant bits of the partial product.

The ALU is capable of shifting data down by two bits for the multiplication algorithm, up by one bit for the divide algorithm and single-bit-up-shifts.

The processor is capable of performing BCD arithmetic on packed BCD numbers. The ALU has separate carry logic for BCD operations. This logic generates nibble carries (BCD digit carry) from propagate and generate signals formed from the A and B operands. In order to simplify the hardware while maintaining throughput, the BCD add and subtract operations are performed in two cycles. In the first cycle, ordinary binary addition or subtraction is performed and BCD nibble carries are generated. These are blocked from affecting the result at this stage, but are saved in the status register to be used later for BCD correction. In the second cycle all BCD numbers are adjusted by examining the previously generated nibble carries. Since all the necessary information is stored in the status register, the processor can be interrupted after the first BCD cycle.

Priority Encoder

The priority encoder is provided to support floating-point arithmetic and some graphics primitives. The priority encoder takes up to 32 bits as input and generates a 5-bit wide binary code to indicate location of the most significant one in the operand. Input to the priority encoder comes from the input multiplexer, which masks all bits that the user does not want to participate in the prioritization. The priority encoder supports 8, 16, 24 and 32-bit operations depending upon the byte width specified. For each data type the priority encoder generates the appropriate binary weighted code. For example, when a byte width of two is specified, the output of the encoder is zero when bit 15 is HIGH. However, if byte width of four is specified ($I_8 - I_7 = 00$), the output of encoder is 16 (decimal) if bit 15 is HIGH and bits 31 – 16 are LOW. Table 2 shows the output for each data type. If none of the inputs are HIGH or the most significant bit of the data type specified is HIGH then the output is zero. The difference between these two cases is indicated by the Z-flag of the status register which is HIGH only if all inputs are zero.

Q-Register

The Q-register holds dividend and quotient bits for division, and multiplier and product bits for multiplication. During division, the contents of the Q-register are shifted left, a bit at a time, with quotient bits inserted into bit 0. During multiplication, the contents of the Q-register are shifted right, two bits at a time, with product bits inserted into the most-significant two bits (according to the selected byte width). The Q-register may be loaded from the A or B inputs and read onto the Y bus.

Master-Slave Comparator

All ALU outputs (except MS-Error) employ tri-state buffers. The master-slave comparator compares the input and output of each buffer. Any difference causes the MS-Error signal to be made true. In SLAVE mode, all output buffers are disabled. Outputs from a second ALU may then be connected to the equivalent pins of the first. The comparator in the slave will then detect any difference in the results generated by the two. When the Y bus is tri-stated by making Output-Enable false, the Y bus master-slave comparators are disabled.

Parity Logic

For each byte of the DA and DB inputs there is an associated parity bit (8 in all). If a parity error is detected on any byte, the PARITY-ERROR signal is made true. Four parity signals (one per byte) are also generated for the Y bus outputs. EVEN parity is employed for the Am29332.

Status Register

All necessary information about operations performed in the ALU is stored in the 32-bit wide status register after every microcycle. Since the register can be saved, an interrupt can occur after any cycle. The status register can be loaded from either the A or B input of the chip and can be read out on the Y bus for saving in an external register file. For loading, the byte width indicates how many bytes are to be updated. The status register is only updated if the HOLD input is inactive.

Each byte of the status register holds different types of information (see Figure 3). The least significant byte (bits 0 to 7) holds six position bits for the data shifter. The two most significant bits are not used. The next most significant byte (bits 8 to 15) holds the 5-bit width field for the mask generator. The three most-significant bits of that byte (bits 13 to 15) are read-only bits that represent three different conditions extracted from the other bits of the status register. They are $\overline{C} + Z$, $N \oplus V$, and $(N \oplus V) + Z$ for bits 13, 14 and 15 respectively. These bits can be read on the Y_0 pin by the extract-status instruction. The next byte contains all the necessary information generated by an ALU operation. The least-significant four bits (bits 16 to 19) hold carry, negative, overflow and zero flags. Bit 20 holds link information for single bit shifts and bits 21 and 22 are used by the multiply and divide instructions. The M flag holds the multiplier bit for the modified Booth algorithm or it holds the sign comparison result for the divide algorithm. The S flag holds the sign of the partial remainder for unsigned division. Both the flags (M and S) are provided as a part of the status register so that multiply and divide instructions can be interrupted at microinstruction boundaries. The most significant byte of the status register holds nibble carries for BCD arithmetic. Since BCD arithmetic is performed in two cycles, the nibble carries are saved in the first cycle and used in the second cycle. Since all the information is stored, BCD instructions are also interruptible at the microinstruction boundary.

TABLE 2.

Highest Priority Active Bit	Encoder Output
Byte Width = 00 (32-bit)	
None	0
31	0
30	1
29	2
28	3
.	.
.	.
.	.
1	30
0	31
Byte Width = 01 (8-bit)	
None	0
7	0
6	1
5	2
.	.
.	.
.	.
1	6
0	7
Byte Width = 10 (16-bit)	
None	0
15	0
14	1
13	2
12	3
.	.
.	.
.	.
1	14
0	15
Byte Width = 11 (24-bit)	
None	0
23	0
22	1
21	2
20	3
.	.
.	.
.	.
1	22
0	23

Status$_{0-7}$: Position Register

PR_7	PR_6	PR_5	PR_4	PR_3	PR_2	PR_1	PR_0
7	6	5	4	3	2	1	0

Status$_{8-12}$: Width Register
Status$_{13}$: $\overline{C} + Z$
Status$_{14}$: $N \oplus V$
Status$_{15}$: $(N \oplus V) + Z$
} Read Only

SIGNED LE	SIGNED LT	UNSIGNED LE	WR_4	WR_3	WR_2	WR_1	WR_0
15	14	13	12	11	10	9	8

Status$_{16}$: Carry
Status$_{17}$: Negative
Status$_{18}$: Overflow
Status$_{19}$: Zero
Status$_{20}$: Link
Status$_{21}$: Multiply (and divide) Bit
Status$_{22}$: Sign Flag
Status$_{23}$: 0

0	S	M	L	Z	V	N	C
23	22	21	20	19	18	17	16

Status$_{24-31}$: Nibble Carries

NC_7	NC_6	NC_5	NC_4	NC_3	NC_2	NC_1	NC_0
31	30	29	28	27	26	25	24

Figure 1. ALU Status Register Bit Assignment

Am29332 INSTRUCTION SET

Data Types

The Am29332 supports the following data types:

1. Integer
2. Binary coded decimal
3. Variable-length bit field

The first two data types fall into the category of byte boundary aligned operands (Figure 2). The size of the operand could be 1 byte, 2 bytes, 3 bytes or 4 bytes. All operands are least significant bit (bit 0) aligned. The byte width is determined by bits I_8 and I_7 of the instruction as shown in Table 3.

TABLE 3.

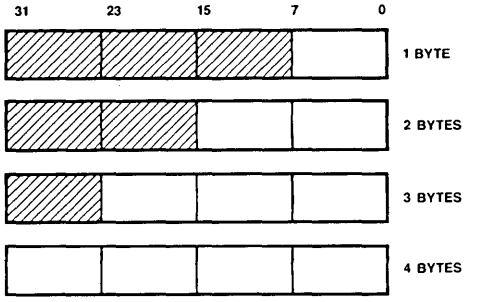
I_8	I_7	Width in Bytes
0	0	4
0	1	1
1	0	2
1	1	3

The third data type has operands of variable width (1 to 32 bits) as shown in Figure 2. The operand is specified by width inputs ($W_4 - W_0$) and position inputs ($P_5 - P_0$). The position inputs indicate the least significant bit position of the operand. Depending on bits I_8 and I_7 of the instruction, the width and position inputs can be selected from either the Status Register or the Width and Position Pins as shown in Table 4. A summary of the data types available is illustrated in Table 5.

TABLE 4.

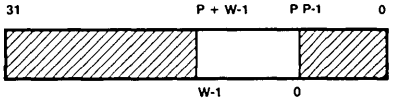
		Position		Width	
I_8	I_7	Pins	Reg	Pins	Reg
0	0	X		X	
0	1	X			X
1	0		X	X	
1	1		X		X

I. Byte Boundary Aligned Operands



TB000096

II. Variable-Length Bit Field



TB000097

P = Bit displacement of the least significant field with respect to bit 0.

W = Width of field in bits.

Figure 2.

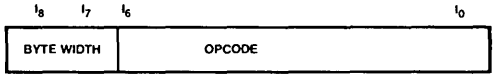
TABLE 5.

Data Type	Size	Range
Integer		Signed Unsigned
1 byte	8 bits	-128 to $+127$ 0 to 255
2 bytes	16 bits	-2^{15} to $+2^{15}-1$ 0 to $2^{16}-1$
3 bytes	24 bits	-2^{23} to $2^{23}-1$ 0 to $2^{24}-1$
4 bytes	32 bits	-2^{31} to $2^{31}-1$ 0 to $2^{32}-1$
BCD	1 to 4 bytes (8 digits)	Numeric, 2 digits per byte. Most-significant digit may be used for sign.
Variable	1 to 32 bits	Dependent on position and width inputs.

INSTRUCTION FORMAT

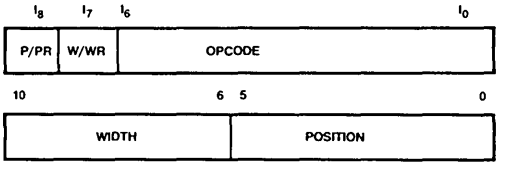
The Am29332 has two types of Instruction Formats:

1. Byte Boundary Aligned Instructions



TB000098

2. Variable-Length Field Bit Instructions



TB000099

For instructions which allow a field to be shifted up or down, $P_5 - P_0$ is a two's complement number in the range −32 to +31 representing the direction and magnitude of the shift. For instructions which assume a fixed field position, $P_4 - P_0$ represent the position of the least-significant bit of the field and P_5 is ignored.

Instruction Classification

ALU instructions can be classified as follows:

A. Byte Boundary Aligned Operand Instructions:

1. Arithmetic
   - Binary, BCD
   - Multiply steps
   - Division steps (single and multiple precision)
2. Prioritize
3. Logical
4. Single-bit shifts
5. Data movement

B. Variable-Length Bit Field Operand Instructions:

1. N-bit shifts and rotates
2. Bit manipulations
3. Field logical operations (aligned, non-aligned, extract)
4. Mask generation

Three-fourths of the ALU instructions apply to operands that are byte boundary aligned. For these instructions, two orthogonal issues are the width of the operand (in bytes) and the contents of the high order unselected bytes on the Y bus. As mentioned earlier, the width of the operand is specified by I_8 and I_7. With the exception of a few instructions, the unselected bytes are assigned values as follows: for single operand instructions, unselected bytes are passed unchanged from the source (A or B). For two operand instructions, unselected bytes are passed unchanged from the destination (B input).

In the last quarter of the instruction set, the width of the operand is from 1 to 32 bits (based on the width input) for field operations, 32 bits for N-bit shift operations and 1-bit for bit-oriented operations. In the case of field-aligned and single-bit operands, the position bits ($P_4 - P_0$) determine the least significant bit of the operand. In the case of N-bit shifts and field non-aligned operands, the position bits $P_5 - P_0$ is a 6-bit signed integer determining the magnitude and direction of the shift.

The operation of each instruction can be explained by the use of a collection of handy functions. The most common of these describes a fundamental property of the ALU:

Merge (X, Y, Mask)

Here the selected bits (determined by the Mask) pass X while the unselected bits pass Y.

Most single byte boundary aligned operand instructions of the ALU can be explained by:

Y ← Merge (f(operand), operand, bytemask)

where bytemask itself is a function of byte width and can be denoted by bytemask = mask (byte width). The function bytemask returns a mask consisting of ones in the least significant bytes (selected by byte width) and zeros in the remaining bytes. In the above operation, the result of the function is returned in the least significant bytes selected by the mask and the operand in the high order unselected bytes.

Similarly two-operand instructions can be explained by:

Y ← Merge (f(A, B), B, bytemask)

The only difference is that here the operation is done on two operands and that the unselected high order bytes always pass the B operand.

The shift operation on byte boundary aligned can be explained by:

Up/Down Shift (operand, fill-bit, byte-width)

where byte-width determines the number of bytes to be shifted and fill-bit is the bit shifted in.

The variable bit field operations can also be explained in the same manner:

Y ← Merge (f(A, B), B, bitmask (position, width))

The mask in this case is a bitmask and is a function of position and width. Position determines the position of the least significant bit of the selected field, and width determines the number of higher order bits selected. Mask bits are HIGH for selected bits in the word and LOW for the remaining bits. The function is done on only the selected bits; a pass of the source operand on unselected bits for single operand instructions and operand B for two operand instructions is performed.

Flags

Byte-Aligned Instructions:

The zero flag always looks only at the selected bytes:

Z ← (Y and bytemask (byte width) = 0)

Similarly, N ← sign bit (Y, byte width), where the function "sign-bit" returns bit 7, 15, 23, or 31 of the first argument for byte widths 01, 10, 11, or 00 respectively.

Also, C ← carry (byte width) returns the carry from the appropriate byte boundary, and:

V ← overflow (byte width)

returns the overflow from the appropriate byte boundary.

The link (L) flag is generally loaded with the bit moved out of the highest selected byte in the case of upshifts, or the bit moved out of the least significant byte for downshifts. Other status flags have specialized uses, explained in the following sections.

Variable-Length Field Instruction:

Generally, only N and Z are affected. N takes the most-significant bit of the 32-bit result (i.e., N ← Y_{31}). Z detects zeros in the selected field of the result (i.e., Z ← (Y and bitmask (position, width) = 0).

Output Select

The Register Status pin may be used to switch the C, Z, N, V, and L output pins between the direct output of the ALU and the outputs of the corresponding bits in the status register. If the direct status output is selected, then for instructions that do not affect a particular flag (e.g., carry for logical arithmetic) that output will reflect the state of its corresponding bit in the status register. Similarly, when the HOLD signal is made HIGH, the C, Z, N, V and L pins will be made equal to the contents of the status register, regardless of the RS input.

INSTRUCTION SET SUMMARY

Operand Size: Variable Byte Width: 1, 2, 3, 4 Bytes

Type	Operation	Data Type
Arithmetic	• Increment by one, two, four • Decrement by one, two, four • Add, addc (carry = macro/micro) • Sub, subr • Subc, subrc (carry/borrow) • BCD sum and difference correct steps	Binary Integer and BCD
	• Negate (two's complement) • Multiply steps (modified Booth) • Divide steps (non-restoring) } (Signed and unsigned)	Binary Integer
Prioritize	• Prioritize	Binary
Logical	• Not, OR, AND, XOR, XNOR, zero, sign	Binary
Single-Bit Shifts	• Upshift with 0, 1, link fill • Downshift with 0, 1, link, sign fill } (Single and double precision)	Binary
Data Movement	• Zero extend • Sign extend • Pass-status, Q-Reg • Load-status, Q-Reg • Merge	Binary

Operand Size: 32 Bits

Type	Operation	Data Type
N-Bit Shifts N-Bit Rotates	• Upshift by 0 to 31 bits with 0 fill • Downshift by 1 to 32 bits with 0, sign fill • Rotate by 0 to 31 bits	Binary

Operand Size: Single Bit

Type	Operation	Data Type
Bit Manipulation	• Extract • Set • Reset	Binary

Operand Size: Variable Length Bitfield: 1 to 32 Bits

Type	Operation	Data Type
Field Logical (aligned and non-aligned)	• Not, OR, XOR, AND, extract, insert	Binary
Mask	• Pass-mask	Binary

Am29334

Four-Port, Dual-Access Register File

ADVANCED INFORMATION

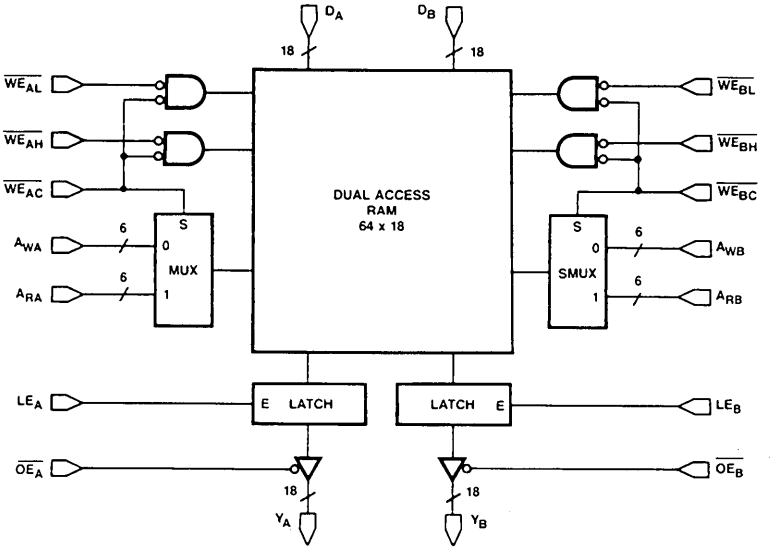
DISTINCTIVE CHARACTERISTICS

- **Fast**
  With an access time of 20ns, the Am29334 supports 80–90ns microcycle time when used with the Am29300 Family for 32-bit systems.
- **64 x 18 Bits Wide Register File**
  The Am29334 is a high-performance, high-speed, dual-access RAM with two READ ports and two WRITE ports.
- **Cascadable**
  The Am29334 is cascadable to support either wider word widths, deeper register files, or both.
- **Simplified Timing Control**
  Control for write enable timing and for on-chip read/write multiplexer are derived from a single-phase clock input.
- **Byte Parity Storage**
  Width of 18 bits facilitates byte parity storage for each port and provides consistency with the Am29332 32-bit ALU.
- **Byte Write Capability**
  Individual byte-write enables allows byte or full word write.

GENERAL DESCRIPTION

The Am29334 is a 64-word deep and 18-bit wide dual-access register file designed to support other members of the Am29300 Family by providing high-speed storage. It has two write and two read ports for data and four 6-bit address ports. Two address ports are associated with each pair of read and write data ports, one to read data and the other to write. The device is capable of performing two reads and two writes in one cycle. The 18-bit wide register file allows storage of byte parity to support parity check and generate in the Am29332 32-bit ALU. Independent control for each read and write data port allows the Am29334 to be used as a high-speed shared memory or as a mailbox for a multiprocessor system. The device is designed with an access time of 20ns. It is housed in a 120 lead-pin-grid-array package.

BLOCK DIAGRAM



BD003022

Order # 05731B

RELATED PRODUCTS

Part No.	Description
Am29323	32 x 32 Parallel Multiplier
Am29325	32-Bit Floating Point Processor
Am29331	16-Bit Microprogram Sequencer
Am29332	32-Bit Extended Function ALU

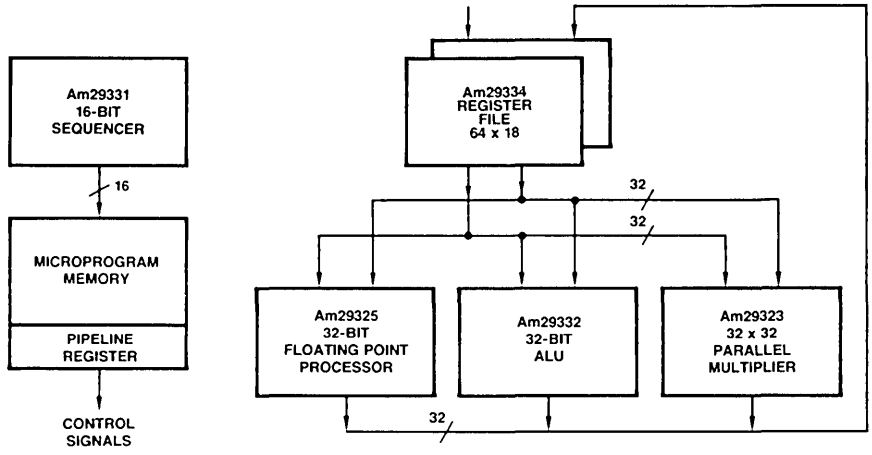


Figure 1. Am29300 Family High Performance System Block Diagram

PIN DESCRIPTION

$A_{RA,0-5}$ **Input**
Read address for Y_A

$A_{RB,0-5}$ **Input**
Read address for Y_B

$Y_{A,0-17}$ **Three-State Output**
Data output A

$Y_{B,0-17}$ **Three-State Output**
Data output B

$A_{WA,0-5}$ **Input**
Write address for D_A

$A_{WB,0-5}$ **Input**
Write address for D_B

$D_{A,0-17}$ **Input**
Data input A

$D_{B,0-17}$ **Input**
Data input B

LE_A **Input**
Latch enable A

LE_B **Input**
Latch enable B

$\overline{OE}_A$ **Input**
Output enable for Y_A

$\overline{OE}_B$ **Input**
Output enable for Y_B

$\overline{WE}_{AC}$ **Input**
Common write enable A

$\overline{WE}_{AL}$ **Input**
Low byte write enable A (bits 0 – 8)

$\overline{WE}_{AH}$ **Input**
High byte write enable A (bits 9 – 17)

$\overline{WE}_{BC}$ **Input**
Common write enable B

$\overline{WE}_{BL}$ **Input**
Low byte write enable B (bits 0 – 8)

$\overline{WE}_{BH}$ **Input**
High byte write enable B (bits 9 – 17)

14 power pins.

FUNCTIONAL DESCRIPTION

The part has two read ports ($Y_{A,0-17}$, $Y_{B,0-17}$), two write ports ($D_{A,0-17}$, $D_{B,0-17}$), four addresses ($A_{RA,0-5}$, $A_{WA,0-5}$, $A_{RB,0-5}$, $A_{WB,0-5}$), two latch enables (LE_A, LE_B), two output enables ($\overline{OE}_A$, $\overline{OE}_B$), and six write enables $\overline{WE}_{AC}$, $\overline{WE}_{AL}$, $\overline{WE}_{AH}$, $\overline{WE}_{BC}$, $\overline{WE}_{BL}$, $\overline{WE}_{BH}$) that allow writing of data into one or both bytes of a word. The separate read and write addresses facilitate creation of three and four-address architectures and allow address set-up and RAM access to overlap.

Since the A and B sides are identical, only operation of the A side is described. The address multiplexer provides the RAM with the address A_{RA} when $\overline{WE}_{AC}$ = HIGH and with the address A_{WA} when $\overline{WE}_{AC}$ = LOW. Internally the part is designed so that there is no race condition between the write address and the write enable. In most cases $\overline{WE}_{AC}$ and LE_A will be connected to the clock as shown in Figure 2 so that reading will take place in the first part of a clock cycle and writing in the last part. The latch at the output of the RAM is transparent when LE_A = HIGH and retains the data when LE_A = LOW. The latch has a three-state output Y_A controlled by $\overline{OE}_A$. Each word is split into two bytes of nine bits that can be individually written. The low byte covers bits 0 through 8 and the high byte covers bits 9 through 17. One or both bytes of the data at D_A are written into the location given by A_{WA} when the common write enable ($\overline{WE}_{AC}$) and the appropriate byte write enables ($\overline{WE}_{AL}$ and $\overline{WE}_{AH}$) are active.

Two special cases arise. First, if a location is written into and read at the same time, the value read is the value being written. Second, if a location is written into from both the A side and the B side, the value written is undefined, but the operation is not harmful.

Extension To Four Read Ports and Two Write Ports

A RAM with four read ports and two write ports can be made by using two dual access RAMs and connecting each of the write ports, write addresses, and write enables in parallel for the two devices. As an example, this RAM may provide data storage for a data ALU and an address adder as shown in Figure 3. A location should not be read before it has been written into for the first time as the contents of the two dual access RAMs are likely to be different upon power-up.

32 Words x 36 Bits Single Access Ram

It is possible to convert the 64 words x 18 bits dual access RAM into a 32 word x 36 bit single access RAM by storing the upper half of the 36 bits in the upper half of the 64 words and address these from the A side and storing the lower half of the 36 bits in the lower half of the 64 words and address these from the B side. This arrangement, which is shown in Figure 4, does not change the capacity of the RAM, but the dual access is lost.

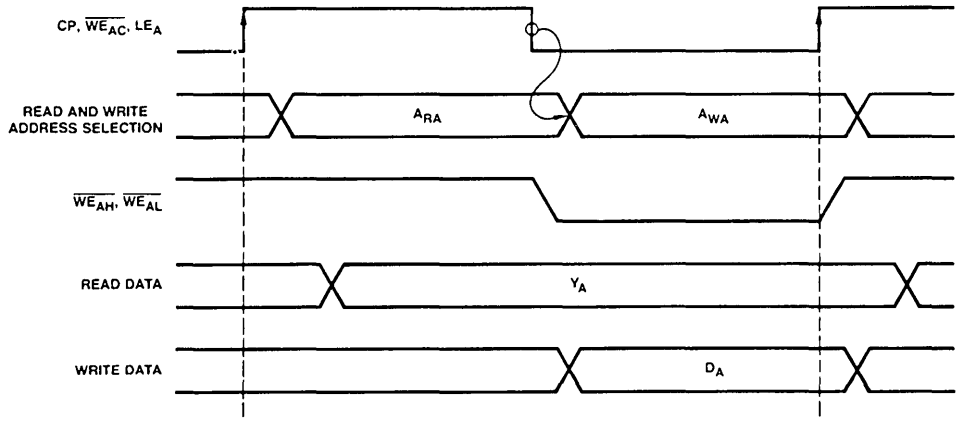
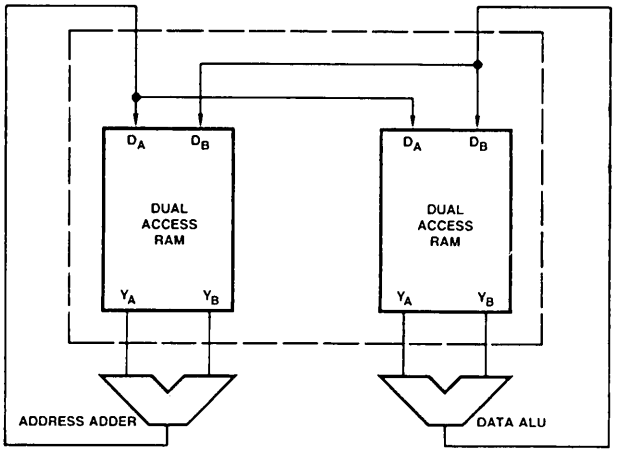
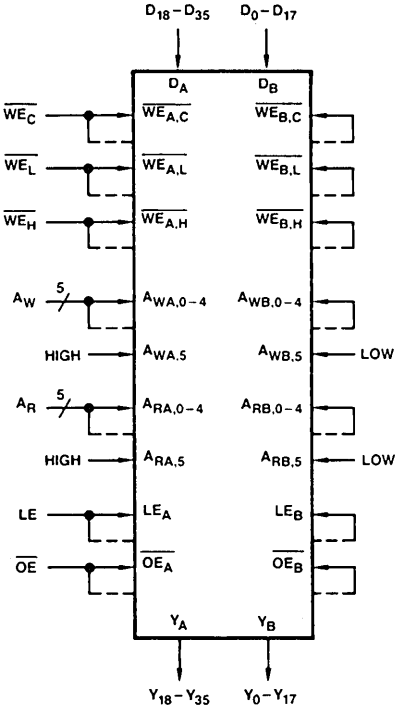


Figure 2. Read through Y_A and Write through D_A in a Single Cycle (Two Bytes)



AF003490

Figure 3. RAM with 4 Read Ports and 2 Write Ports



LS001790

Figure 4. 32 x 36 RAM (Single Access) Using 64 x 18 Dual Access RAM

ADVANCED MICRO DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale,
California 94088
(408) 732-2400
TWX: 910-339-9280
TELEX: 34-6306
TOLL FREE
(800) 538-8450

Order #06584A

Printed in U.S.A. CBM-MU-35M-4/85-0