By Robert Stodieck

The IDT FourPort SRAM

Serving as both a complex four bus interconnect network and fast "parallel" memory, the IDT FourPort™ SRAM can greatly facilitate the creation of multiprocessor and multi-ALU systems to accelerate DSP, graphics, control and other tasks that involve large vector processing tasks.

Memory architectures based on single-port SRAM allow only one device to access a memory array at one time. Hardware designed to accelerate computing processes by utilizing parallelism, or pipelining with single-port memory tend to require architectures that are either complex, specialized, or both. The advent of a fast FourPort™ single chip SRAM greatly simplifies the task of creating generalized small multiprocessor or multi-ALU systems to accelerate a variety of vector algorithms.

Potential applications include dedicated real-time multiprocessor systems for control, graphics, and DSP systems, as well as general purpose vector co-processors to assist general purpose computers. Vector processing means any computing operation with a large number of operations that may be executed in parallel by multiple processors. In these applications the FourPort™ SRAM serves both as a fast static RAM and as the interconnect network between processors working on a common data set.

Imagine a static RAM that allows four processors to randomly and asynchronously read or write four locations at a time in the same SRAM array. For processes that can be executed in parallel, four processors can be programmed to operate simultaneously on different parts of a data set stored in the FourPort SRAM. If data is being generated at different rates than it is being used, software controlled buffers can be created at will, temporarily storing data passing from one processor to the next. The buffering minimizes the time lost in handshaking between processors. Four-way fully random accessibility avoids hardware imposed algorithmic constraints.

The IDT FourPort™ SRAM has precisely these characteristics. There are only two constraints on the access patterns allowed in the FourPort™. Two devices cannot write to the same address location in the SRAM at the same time, since simultaneous multiple writes to any one multiport memory location may corrupt the data in that SRAM location. Also, a device cannot read an address location that is being written, to avoid having the read occur when the output data is changing. There are no other restrictions on access patterns.

As it turns out, address collisions are usually prohibited by the logical sequencing requirements of software, and the time lost in avoiding address collisions is often minimal. Most of the time all processors have essentially free read and write access to the memory.

FourPort™ SRAM-Based Multiprocessor Arrays for Vector Operations

The FourPort™ SRAM is both a storage and communications media. As a communications media it has little, and in some cases, zero handshaking or arbitration overhead. A processing device may be able to store results in a multipart memory and spend little or no time signaling the next device to receive the results. As a communications media it also has very high bandwidth. These characteristics make the IDT7054 and the IDT7052
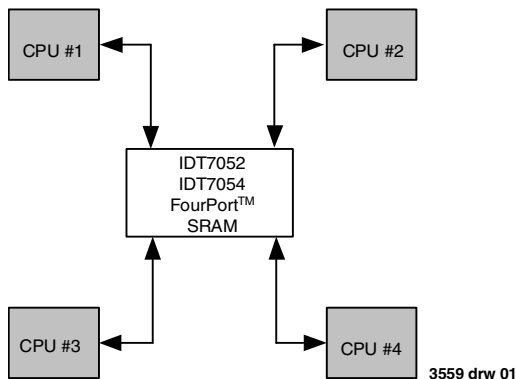

Figure 1. The IDT7052/4 FourPort RAM allows four simultaneous memory accesses to independent addresses a 2K or 4K x 8-bit memory array. It serves both as an interconnect network and as a fast static RAM.
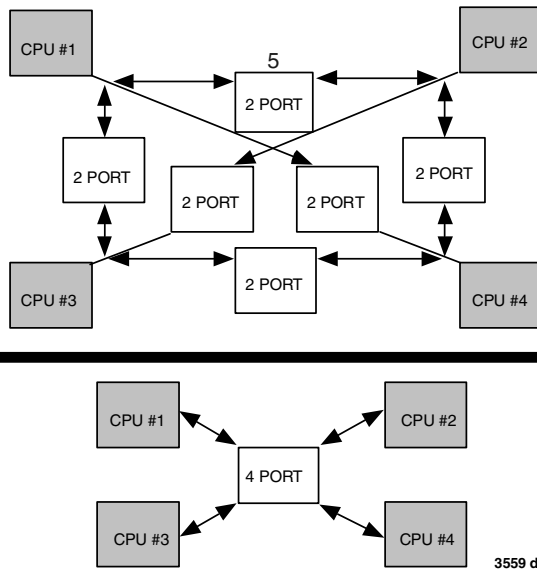
3559/3

Figure 2.   FourPort™ SRAM interconnection advantages over Dual-Port SRAM. The processors in both figures are interconnected with a latency of one memory access. This efficiency requires 6 seperate Dual-Port SRAMs but only 1 FourPort™ SRAM.

an ideal memory for connecting multi-element and multiprocessor computer architectures (see Figure 2).

Since the hardware interface of the FourPort™ SRAM to the processors is that of a simple static RAM, it can be connected transparently to almost any existing system. Control signals as well as data can be handled via the SRAM. Thus, microprocessor boards that were designed for entirely different applications can be used in a multiprocessor array.

An Overview of the Operation of a Multiport RAM Based MULTI-PRO-CESSOR WITH A MULTIPORT RAM BASED CONTROL SYSTEM

Processors sharing multiport memory must avoid writing into memory locations that are simultaneously being read or written from another port. This is usually accomplished by address range segregation. That is, at any one moment processor "A" is prevented from writing to multiport memory locations that processor "B" is accessing from another port. Hardware interrupts, hardware semaphores and stalling processors with hardware busy logic are hardware based methods of controlling the accesses of processors to multiport SRAM. It is also possible to control the processors in a multiprocessor array via the common SRAM interface. This results in an essentially software-only control system. The control algorithms for a multiport SRAM based multiprocessor array are different than those for a multiprocessor array based on single port SRAM. This section describes an example of a control protocol for a multiport SRAM based multiprocessor array.

Access coordination in multiport SRAM based multiprocessors, over-laps with the more familiar task of process coordination in a multiprocessor and uses the same control schemes. In a single master system, the master determines the address ranges being used by all processors. This avoids the problems of arbitrating for resources. In small embedded systems, running algorithms of limited complexity, the software can be tuned so that software-only control approaches have little or no detrimental effect on overall performance. Such systems are more easily debugged if a simple single master control arrangement is used. In this section of this application note we will discuss a single master example.

In a master/slave array, the master controls all the actions of the slaves. The slaves must either have local program stored in SRAM or ROM or be operating out of the FourPort™ SRAM. Each processor must have a unique ID code to be able to identify the unique command location where it is to receive its commands from the master. This can be achieved, for example, by supplying a unique firmware ID code via individual PROMs, PALs or readable DIP switches for each processor. A number of other approaches are possible.

Each slave command has a corresponding op-code. The slaves poll their command locations looking for new command opcodes. For example, finding a "0" in a command location may imply no operation is requested from the slave etc. The commands can be anything that the slave processors have been programmed to do. Appropriate commands might be, multiply data values at locations 000H to 7FFH with the corresponding coefficients at locations 800H to FFFH, or multiply data values at locations 000H to 7FFH with the value at location 800H, etc. Thus, with a few memory accesses, the master processor can trigger and control lengthy slave processor operations.

Master/Slave Control Protocol for a Multiport RAM Based Processor Array

A command protocol is the set of rules for passing commands from the master to the slaves. In a software-only control system, all processors must be aware that writes to certain command locations are forbidden, or
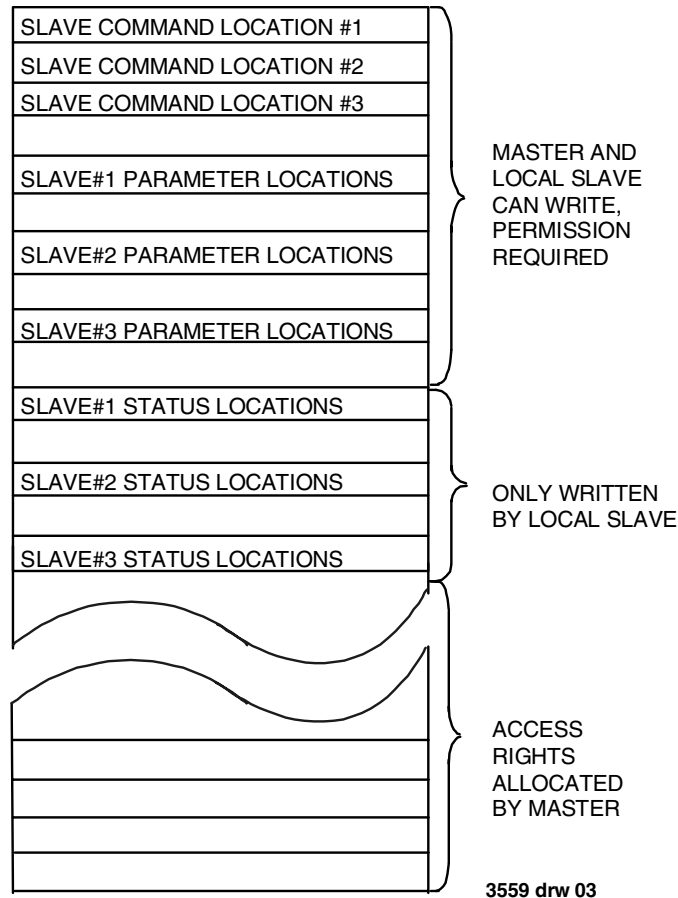
```
┌─────────────────────────────────────┐
│ SLAVE COMMAND LOCATION #1            │
├─────────────────────────────────────┤
│ SLAVE COMMAND LOCATION #2            │
├─────────────────────────────────────┤
│ SLAVE COMMAND LOCATION #3            │
├─────────────────────────────────────┤
│                                     │   MASTER AND
├─────────────────────────────────────┤   LOCAL SLAVE
│ SLAVE#1 PARAMETER LOCATIONS          │   CAN WRITE,
├─────────────────────────────────────┤   PERMISSION
│                                     │   REQUIRED
├─────────────────────────────────────┤
│ SLAVE#2 PARAMETER LOCATIONS          │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│ SLAVE#3 PARAMETER LOCATIONS          │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│ SLAVE#1 STATUS LOCATIONS             │
├─────────────────────────────────────┤
│                                     │   ONLY WRITTEN
├─────────────────────────────────────┤   BY LOCAL SLAVE
│ SLAVE#2 STATUS LOCATIONS             │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│ SLAVE#3 STATUS LOCATIONS             │
├─────────────────────────────────────┤
│                                     │   ACCESS
│                                     │   RIGHTS
│                                     │   ALLOCATED
│                                     │   BY MASTER
└─────────────────────────────────────┘
```

3559 drw 03

Figure 3. Write Access Allocations

forbidden without "permission" from the current owner (see Figure 3). In general, a process is given a variable address range to operate in. The command protocol on the other hand, uses fixed address locations.

The master of an array of processors can tell slave processor #1 to execute a command "n", by writing the command opcode corresponding to command "n" to the slave processor's command location. Parameters for the process, such as constants, or the assigned address range, are placed in reserved locations prior to starting the process that will use them.

There are four problems that a multiport SRAM based command protocol must solve:

1. Write-write conflicts must be avoided in the control locations.

2. Read-write synchronization problems must be avoided in the control locations.

3. The master must not issue a new command out of sequence, i.e. the slave has to acknowledge readiness to execute a new command.

4. A slave must execute each command only one time.

Figure 4 shows flow charts for a protocol that allows a master to control slaves and slaves to receive commands without risk of violating these four rules.

All slaves have unique command locations in SRAM. If the reads and writes to the command locations are asynchronous, command locations must always be read at least twice. The two read results are then compared and discarded if they do not match. In this way, command data that may have been changing during the read operation, and therefore may have been read incorrectly, is discarded.

Before issuing any command, the master first reads a slave's "command" location. If the value read indicates that the slave is ready, the master places the slave's command op-code in that same command location. The slave must signal readiness for new commands by placing a "no-op/ready" value in the command location. The "no-op/ready" flag value is interpreted as a "ready-for new-command" flag by the master, and a "no-operation" command by the slave.

Having to wait for a "ready" signal from the slave prevents the master from issuing new commands out of sequence. Conversely, by signaling "ready" in this way, the slave is also clearing the old commands from the command location. This prevents the slave from later accidentally re-reading and re-executing an old command. The command locations are written alternately by the master and the designated slave, but the protocol prevents simultaneous writes that might destroy the data in the SRAM location. By using the same location for both the master's command and
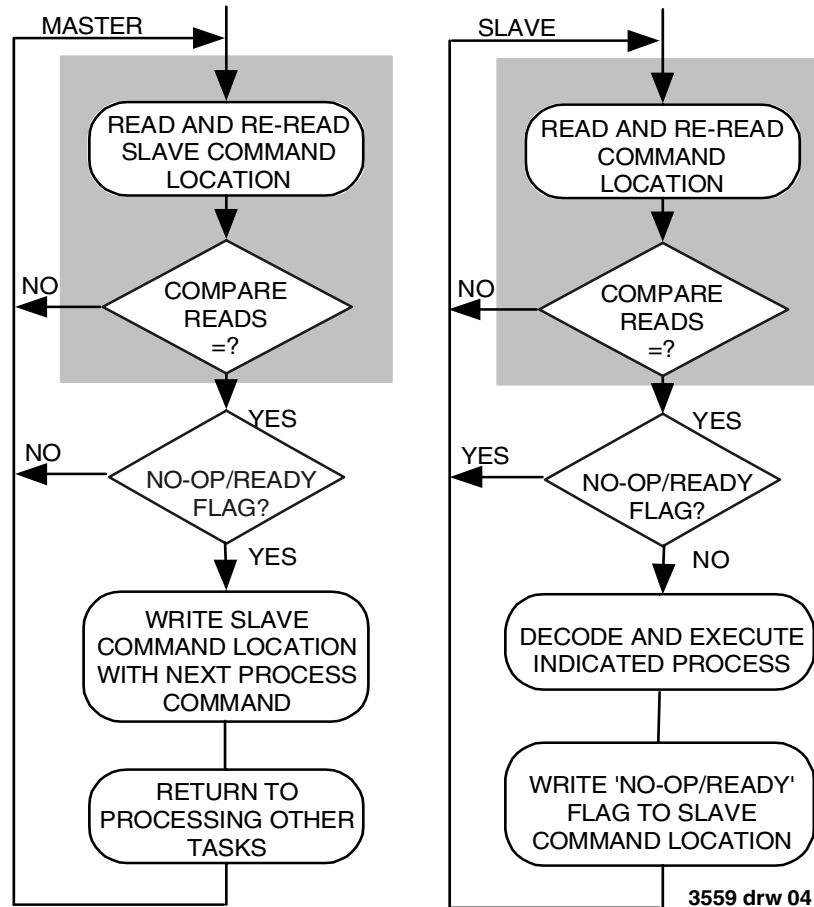
Figure 4. Flow charts for a master-slave software-only command protocol for a multiport SRAM based multiprocessor. In unsynchronized systems (see shaded boxes) all commands must be read at least twice with the same result before the command keyword can be assumed to be valid.

the slave's ready indication, synchronization problems caused by differences in the memory cycle rates of different processors can also be avoided.

All slaves should also have unique slave status locations in SRAM where the master looks for slave status information. The status locations are writable by the slave only. Copious use of reserved slave status locations is essential for the benefit of the programmer trying to debug untested software.

The slave may also signal "done" by writing a "done" flag to a slave status location. The meaning of "done" is that the results of the last operation are ready for use. Keep in mind that "done" is a different signal than "ready". "Ready" implies that the master can post the next command and return to executing other tasks. Depending on the overall algorithm the master is controlling, the master may write a new command as soon as "ready" is signaled, or it may need to wait until "done" is signaled also. It cannot merely check for a "done" signal before issuing a new command. To do so would make it possible to issue new commands out of sequence, based on stale "done" signals.

Initialization

Since multiport SRAM is the control interface, the SRAM command locations must be initialized prior to starting the execution of the slaves. One way this can be handled is by delaying the reset pulses to the slaves while the master initializes SRAM. Alternatively, after reset, the master can issue a known sequence of commands that frees the slaves from a special start up routine.

Outline a Digital Signal Processing Example

Basic DSP algorithms such as the FFT can utilize high degrees of parallelism and provide good examples of vector algorithms. The access patterns of the processor doing such an algorithm are complex and the data sets are usually small enough to fit comfortably in a multiport SRAM array. Analyzing how the FFT will be processed provides a good example of the advantages of a multiport SRAM based multiprocessing environment.

The objective of our example task is to translate a time series of data
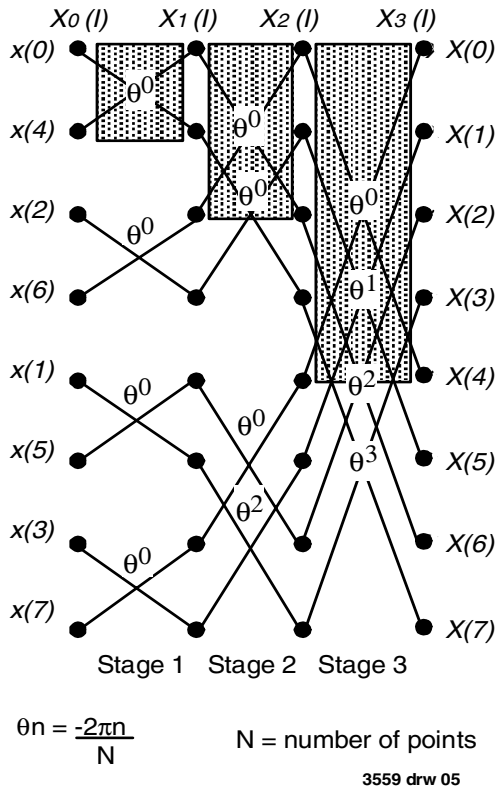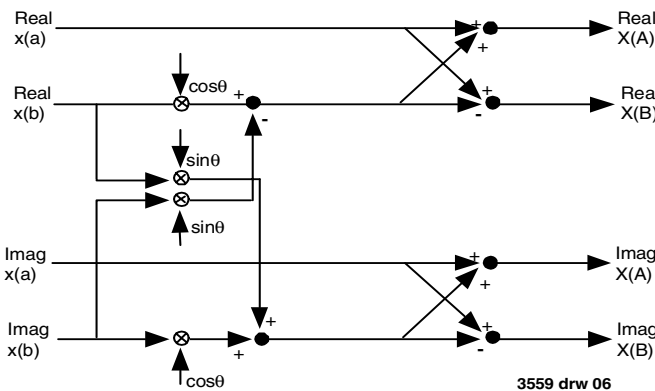
$$\theta n = \frac{-2\pi n}{N}$$

N = number of points

3559 drw 05

Figure 5   Overview of the "Butterfly" calculations for an 8 Point FFT. To complete this 8 Point FFT requires 3 stages of Butterflies ($2^3 = 8$). A 1K FFT has 10 stages or levels ($2^{10} = 1K$). The indexes of x(n), the input sequence, are shown to be out of sequence for graphical clarity. Each stage in this figure has 4 Butterflies.

3559 drw 06

Real X(A) = Real x(a) + (Real x(b) • cosθ - Imag x(b) • sinθ)

Real X(B) = Real x(a) - (Real x(b) • cosθ - Imag x(b) • sinθ)

Imag X(A) = Imag x(a) - (Imag x(b) • cosθ - Real x(b) • sinθ)

Imag X(B) = Imag x(a) - (Imag x(b) • cosθ - Real x(b) • sinθ)

Figure 6. Flow Diagram for Calculating One FFT Butterfly. Each "X" pattern shown in Figure 5 represents one such "Butterfly". Each end point in Figure 5 represents a complex pair of numbers input or output to or from a "Butterfly". The sine and cosine factors are sometimes called "Twiddle Factors". The angles used for calculating the Twiddle Factors for each Butterfly are shown in Figure 5.

values into their frequency domain representation: i.e. execute a fast Fourier transform, as quickly as possible. This is a common process stop in a number of systems for interpreting data from things as diverse as military radar to medical CAT scans. It is also a relatively well known algorithm among many contemporary electrical engineers, and so makes a good example for our system.

Our objective algorithm could be run on a single processor. The object of the FourPort™ SRAM based multiprocessor arrangement is to multiply the speed of our computational process without resorting to a specialized and more expensive architecture.

The generality of this architecture implies that it can be applied to a variety of computationally involved tasks. The generality of this architecture also means that there are often a number of ways a programmer can attack a specific problem. The intent of this example is merely to illustrate one approach, not to fully optimize an algorithm.

Load Balancing

The FFT calculations can be flow graphed. The resulting set of four equations is called a "butterfly" for the appearance of its flow graph (Figure 5). When the FFT calculations are flow graphed they appear as a repetitive array of calculations (Figure 6) of a particular set of four equations.

A common bench mark of processor performance is a 1K FFT. A quick glance at the equations to be calculated shows why multiple processors are desirable for such a task. If we assume that 1024 real and 1024 imaginary data values have been loaded in the four port memory, there are now 2048 multiplications to be done as a first step. All these multiplications could be done simultaneously. Next there are 1024 additions followed by another 2048 additions to complete the first stage of FFT butterflies. Again, all of operations at any one of these three steps could be done simultaneously. For a 1K FFT there are 10 stages of butterflies.

Processing on one stage of the FFT must be completed before processing on the next stage can begin. Each processor is given an address range of FFT butterfly input data to process for each stage of FFT butterflies. A sine table is required for calculation of the FFT "twiddle" factors. This can be stored in the four port memory and, therefore, will always be available to all processors. Calculation of the "twiddle" factors is a matter of calculating the addresses used in the sine look-up table. (See Figure 6 for the angle calculations).

For efficiency, the computational load between processors must be balanced. Since there are hundreds or thousands of operations that may be done in parallel at each stage of the FFT, task partitioning is a matter of assigning each processor an appropriate number of "butterflies" to work on to achieve an equity of loading.

Since the minimal FFT tasks are easily divided between the processors, and the FourPort™ SRAM all but prevents inter-processor data transfer conflicts, the four processors in this example can be kept busy most of time.

Since there are so many tasks that can be done in parallel, other types of tasks can be included without seriously upsetting the balance. For example, if one processor is being used to handle I/O and input conditioning tasks, then it can be assigned to do fewer butterfly calculations than the other processors. If the work load of all the processors can be balanced, the net speed advantage of this four processor array, can then in fact be close to 4 times that of a single processor.
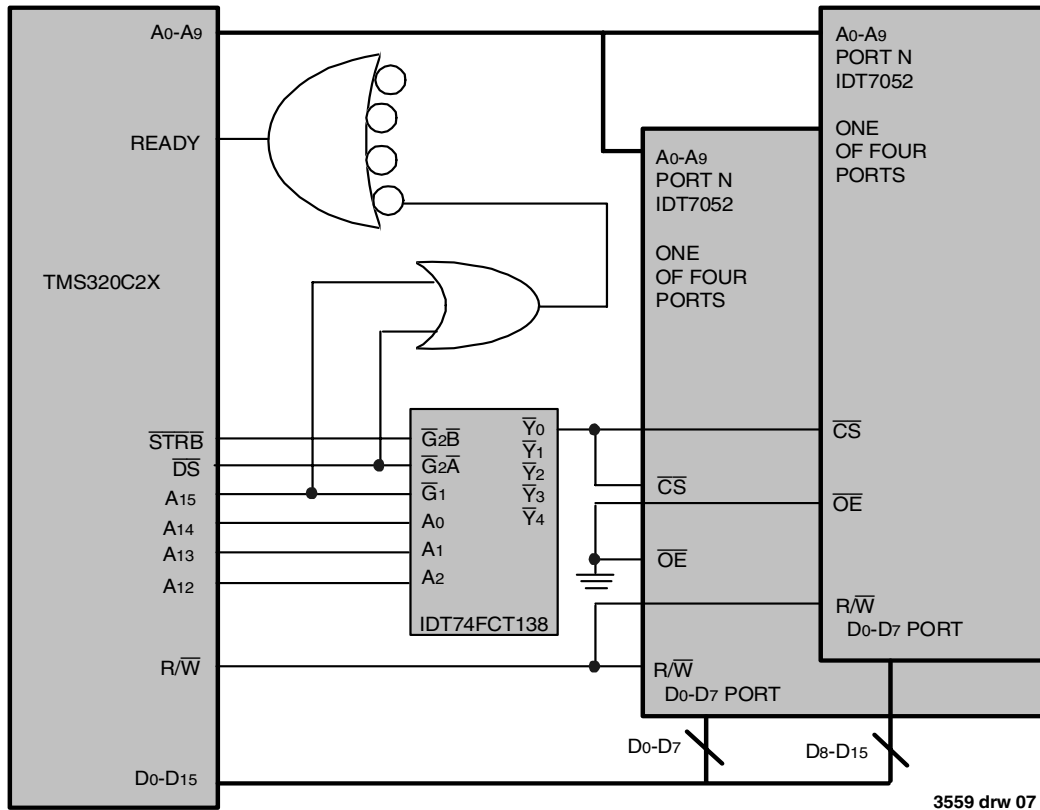
Figure 7.   A 16-bit TMS320C2X to IDT7052 Interface Example

A TMS320C2x Hardware Interface Example

TI's TMS320 single chip DSP processors are particularly well suited for embedded numerical processing. An example interface is shown in Figure 7. The internal SRAM and ROM of the TMS320 can be used for temporary data storage and program memory, making the FourPort™ SRAM the only external SRAM required in a processor array. The FourPort™ SRAM also cascades in depth and width as easily as a standard single port SRAM. The interface shown in Figure 7 would typically require 30ns SRAMs for a 20MHz TMS320C2x type processor.

The TMS320C20 and TMS320C25 also include a "sync" pin that facilitates synchronizing the internal clock phases of multiple processors at reset. In synchronous TMS320C2x arrays, this guarantees that memory accesses are in phase with each other and there are no partial clock phase memory access collisions. This form of processor synchronization is accommodated entirely in hardware.

Summary

The FourPort™ SRAM combines features of fast static memory and a complex multiple bus interconnect network. The FourPort™ SRAM all but eliminates stalls when transferring data between processors or to memory. This prevents bus conflicts from being a bottleneck in multiprocessor systems.

The flexibility of the FourPort™ SRAM allows multiprocessor designs to remain generalized while achieving high speeds on critical vector processes. The fact that the SRAM itself is also the interconnect network between processors eliminates the complexity of a conventional multiprocessor bus system. The straightforward static RAM interface of the FourPort™ SRAM, allows almost any processor to be used in an array for embedded systems. These factors conspire to make practical a variety of new vector processing architectures centered around the world's first "large" truly four ported single-chip SRAM.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES ("RENESAS") PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01 Jan 2024)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.