Multiprocessing 17

17.1 OVERVIEW

Complex signal processing applications may demand higher performance than a single DSP processor can provide. When a single processor falls short, a multiprocessor architecture may boost throughput. However, the law of diminishing returns applies. As more processors are added, additional computation time is spent in interprocessor communication, degrading the overall performance for each processor. Four processors, for example, cannot deliver four times the computation power of one processor. A processor pair almost doubles the speed of a single processor while keeping the architecture and interprocessor coordination as simple as possible. This chapter develops a processor-pair architecture, based on a dual-port RAM. The design is easy to implement and provides a significant computational boost over a single processor.

17.2 SOFTWARE ARCHITECTURE

To complement the hardware design, a hypothetical application is presented. Data is input and low-pass filtered by one processor, then the second processor determines the peak location within a filtered window. Although the software implementation is simplistic, it shows a technique for programming in a multiprocessing environment: alternating buffers and flags.

The alternating buffers in this application are two identical buffers located in dual-port RAM so both processors can access them. The first processor fills buffer 1 with information, while the second processes the information in buffer 2. Each buffer has a flag that indicates completion of operations on that buffer. When processor 1 has finished its operations on the buffer data, it sets the flag, signaling processor 2 to begin operations on that buffer. The sequence of operations is shown in Figure 17.1, on the next page.

The alternating buffer scheme is easier to implement than a single buffer scheme. If only one buffer were used, careful timing analysis or extensive handshaking would be required to ensure that the processors did not use old or invalid data.

17 Multiprocessing

Processor 1 (Filter)	Processor 2 (Peak Locator)
Initialize flags, coefficients delay line, pointers	Initialize pointers
Perform low pass filter operation on data in buffer 1	Check flag 1; wait if not set
Set flag 1	
	Check flag 1; if set, perform peak locating operation on data in buffer 1
Perform low pass filter operation on data in buffer 2	
	Clear flag 1
Set flag 2	
	Check flag 2; if set, perform peak locating operation on data in buffer 2
Perform low pass filter operation on data in buffer 1	
	Clear flag 2
Set flag 1; etc.	
	Check flag 1; etc.

Figure 17.1 Alternating Buffers and Flags

17.3 HARDWARE ARCHITECTURE

This system includes two ADSP-2100s, each with its own private memories. Private memories are accessible to one processor only. Common memory is accessed by both. Figure 17.2 shows a block diagram of the system.

Each processor has a private memory of 32K of 24-bit program memory and 14K of 16-bit data memory. In addition, 2K of 16-bit dual-port RAM is shared by both processors. This area of memory allows inter-processor communication and data transfers.

17.3.1 Using Dual-Port Memory

The 2K x 8-bit dual-port RAMs used in this design are the IDT7132 and the IDT7142 produced by Integrated Device Technology. A useful feature of the IDT7132 is its on-chip arbitration support. The IDT7142 acts as a

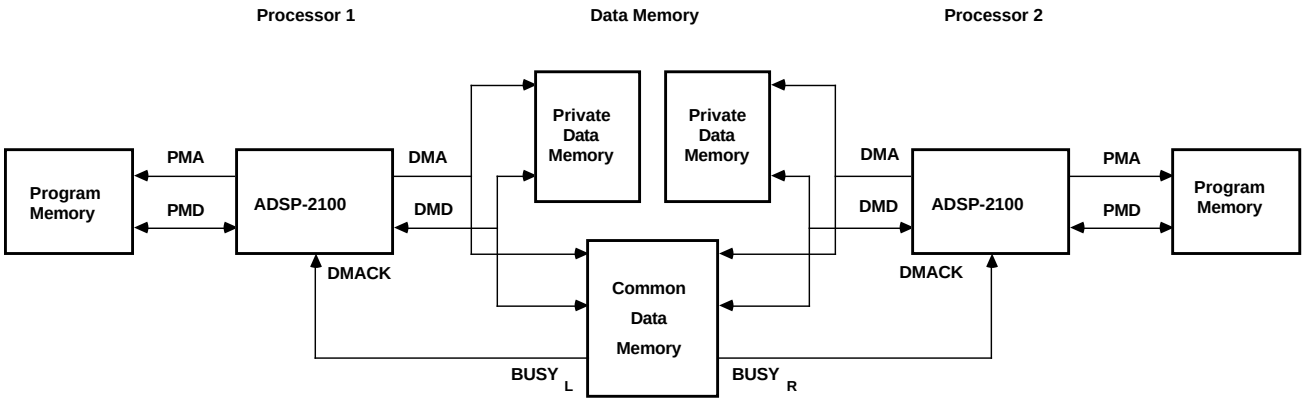


Figure 17.2 Processor Pair Block Diagram

slave chip to the IDT7132 and does not require arbitration circuitry. Most memory accesses can be completed without arbitration, but contention situations require arbitration. Contention occurs if both processors are writing the memory at the same time or if one processor is writing while the other is reading. A simultaneous read by both processors does not cause contention.

Without arbitration, simultaneous writes to a location result in an indeterminate value. The actual value stored is dependent on component timing and other variables.

A simultaneous read/write might occur when one processor is updating variables that the other processor is using in its computations. When one processor is reading a location the other is writing, the result without arbitration depends on the timing of the individual components. The result of the read might be the old value, the new value, or something in between.

The on-chip arbitration of the IDT7132 prohibits simultaneous access of a single memory location. The arbiter circuit consists of two address comparators and two BUSY output signals, one signal for each side of the dual-port memory. If both addresses are equal (and $\overline{CE}$ for that memory is active) the processor whose address arrived last is held with the $\overline{BUSY}$ signal. With the $\overline{BUSY}$ output of the RAM connected to the DMACK input of the corresponding ADSP-2100, the ADSP-2100 will insert wait states while the other ADSP-2100 completes a memory access. In this way, the access for one side is delayed.

When programming in this environment, it is important to be aware of the delay that can occur from contention. Wherever possible, contention should be avoided. One way to avoid contention is to synchronize program flow using the flags in software, so both processors do not access the same buffer concurrently. In the example software shown in this chapter, the alternating buffer scheme prevents the processors from trying to access the same data. Contention can occur only if both processors try to access the same flag.

17.3.2 Dual-Port Memory Interface

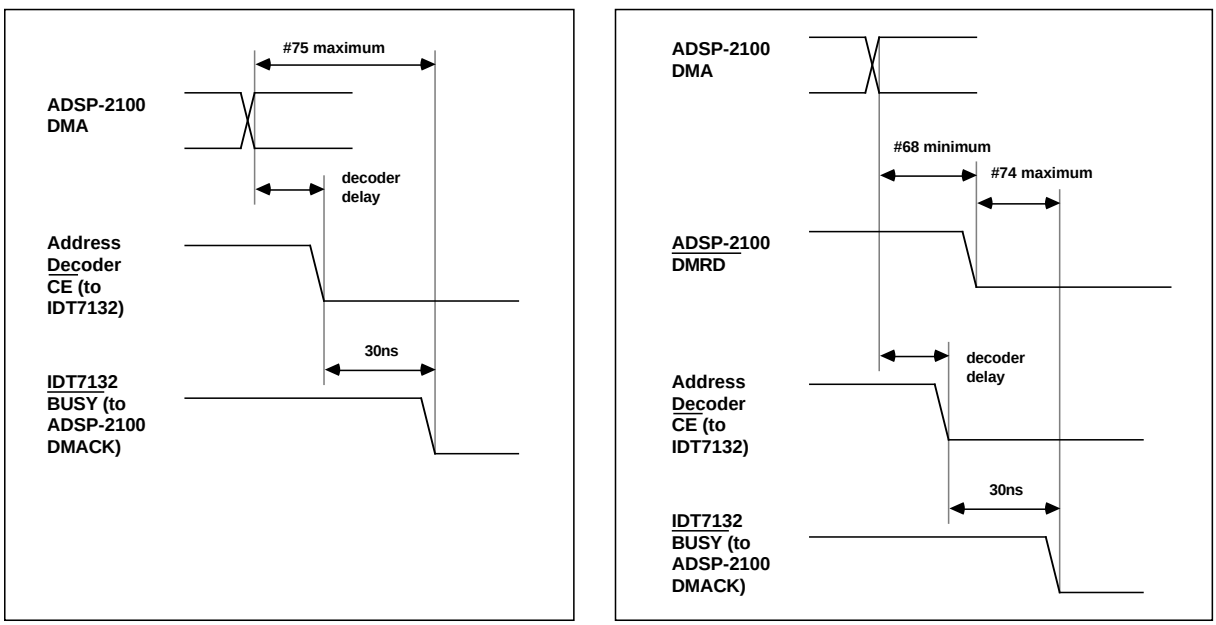
Two 2K x 8-bit dual-port RAM chips are shared between the two ADSP-2100s. This memory is used to transfer information between the two processors. The lowest 2K of data memory space on each processor (locations 0000-2048) is dual-ported. Additional dedicated memory can be added to each processor as needed; decoding is provided for a full 16K of data memory. If your application requires less memory, only include the amount you require. Additional dual-port memory can be used in place of private memory as needed.

The eleven low address bits (DMA0-DMA10) of each processor are connected to either the left-side or right-side address bits of the dual-port memory. The upper three address lines (DMA11-DMA14) are connected to a 1-of-8 decoder to determine which memory bank is enabled. Each BUSY output from the dual-port memory is connected directly to the DMACK input of the corresponding ADSP-2100. When contention occurs, the memory's arbitration circuitry pulls one of the BUSY lines low.

17.3.3 Decoder Timing

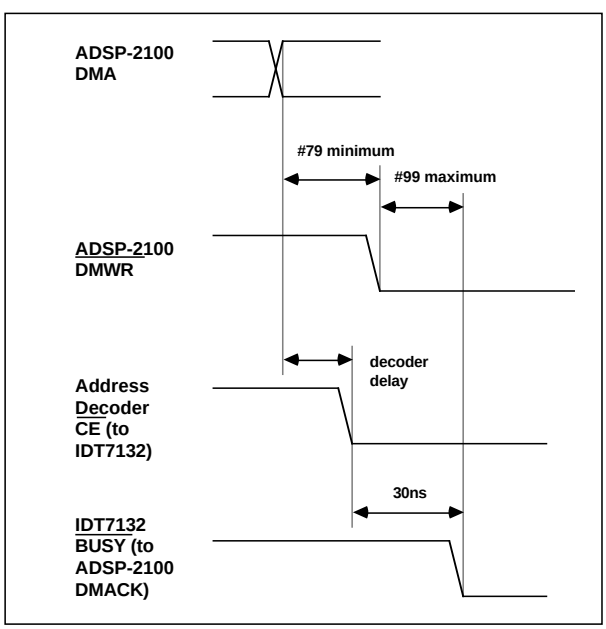
Because the memory consists of multiple 2K blocks of memory, a 1-of-8 decoder is necessary to produce the appropriate chip enable (CE) signal for each memory chip. The delay incurred by decoding the address bits is important because the BUSY output delay from the IDT7132 is relative to CE, and BUSY must be returned to the DMACK input of the ADSP-2100 well before the end of the access cycle.

The CE-to-BUSY delay of the IDT7132 is 30ns (for a 45ns part). The ADSP-2100 has three timing requirements for DMACK. DMACK must be returned a specified time after DMA becomes stable (#75 timing parameter from the *ADSP-2100 Data Sheet*). During a read cycle, DMACK must be returned a specified time after DMRD goes low (#74). During a write cycle, DMACK must be returned a specified time after DMWR goes low (#99). Calculations for each of these three requirements determine the maximum permissable decoder delay, as shown in Figure 17.3. The decoder must be faster than the minimum value of all three.



Requirement 1

Requirement 2



Requirement 3

Figure 17.3 Calculating Decoder Delay Requirement

Requirement 1, for the DMA to DMACK specification (#75), determines the maximum decoder delay as follows:

(DMACK to DMA Valid) – BUSY Delay

or

#75 – 30ns

Requirement 2, for the DMRD to DMACK specification (#74), must include the DMA to DMRD delay (#68). The maximum decoder delay is determined as follows:

(DMA to DMRD) + (DMRD to DMACK) – BUSY delay

or

#68 + #74 – 30ns

Likewise, Requirement 3 for the DMWR to DMACK specification (#99), must include the DMA to DMWR delay (#79).

(DMA to DMWR) + (DMWR to DMACK) – BUSY delay

or

#79 + #99 – 30ns

For an 8MHz ADSP-2100, the minimum value of all three calculations is 7ns. The 74FCT138A 1-of-8 CMOS decoder has a 6ns maximum delay, providing the necessary speed to use the 45ns dual-port memory.

17.4 SYNCHRONIZING MULTIPLE ADSP-2100S

Although processor clock synchronization is not absolutely necessary in a multiprocessor environment, it is advisable. Synchronization ensures that both processors are executing the same internal phase at the same time.

In a system which contains more than one ADSP-2100, synchronization is guaranteed if the processors share a common clock and a common RESET signal. When the RESET line is asserted, both ADSP-2100s become synchronized so that their internal clock phases are the same.

17.5 DEVELOPMENT TOOLS

The ADSP-2100 Development Tools can be used with multiprocessing architectures. Following a few guidelines in writing the software ensures proper operation. Each part of the Development Tools that requires special consideration is described below.

17.5.1 System Builder

The System Architecture file for the system presented in this chapter is shown in Listing 17.1. The program memory space is allocated with 16K of memory for instructions and 16K for data. The data memory space is also divided into two blocks. The lowest 2K is the dual-port memory that shared by both processors. Each processor has an additional 14K of private data memory.

Both processors use the same architecture file in this example. In other applications, each processor might have different memory requirements. The dual-port memory could be mapped into different locations on each processor. For example, one processor could map the dual-port memory in the 0 to 2K range, while the other maps the common memory in the 14K to 16K range.

```
.SYSTEM                        multiprocessor;

{Program Memory Section}
.SEG/ROM/ABS=0/PM/CODE          code_area[H#4000];
.SEG/RAM/ABS=H#4000/PM/DATA     data_area[H#4000];


{Data Memory Section}
.SEG/RAM/ABS=0/DM/DATA          common_memory[H#0800];
.SEG/RAM/ABS=H#800/DM/DATA      private_memory[H#3800];

.ENDSYS;
```

Listing 17.1 System Architecture File

17.5.2 Assembler

Two common arrays and associated flags store the filtered data. They are declared in each processor's main routine (shown in Listings 17.2 and 17.3).

The absolute (ABS) directive causes the variable to be stored in the absolute location specified in the code. Without the ABS directive, the

17 Multiprocessing

Linker is free to place data anywhere in available memory. The Linker places variables in contiguous areas of memory if their declarations are all on the same line.

Because the dual-port memory is defined in both processors' data memory space, the processors must not have conflicting allocations in shared memory. The best way to avoid this is to dedicate sections of the dual-port memory not needed for communication to one processor only. This can be done by allocating a dummy array in one processor's code over the range of memory dedicated to the other processor.

Listing 17.2 shows the variable declarations for the filter processing module. This code executes an FIR filter on the data. The output of the filter is stored in one of the two data arrays. When the array is full, the appropriate full_flag is set, informing the peak processor to start its operations. The filter processor can then filter another window of data into the alternate data array.

The Linker is free to place additional variables anywhere in the unallocated areas. No contention can occur within the common memory, because it is entirely allocated to one processor or the other.

```
.MODULE     data_filter;

.INCLUDE    <const.h>;

{Dual Port Memory Declarations}
.VAR/DM/RAM/ABS=0             full_flag_1, data_1[256];
.VAR/DM/RAM/ABS=256           reserved_for_peak_module[767];

.VAR/DM/RAM/ABS=1024          full_flag_2, data_2[256];

{Private Memory Declarations}
.VAR/PM/RAM/CIRC              coefficient[taps];
.VAR/DM/RAM/CIRC              delay[taps];

.INIT       coefficient : <coeff.dat>;
.INIT       full_flag_1 : 0;
.INIT       full_flag_2 : 0;
```

```
            I0=^delay;        L0=%delay;
            I4=^coefficient;  L4=%coefficient;
            I1=^data_1;       L1=0;
            M0=0;             M1=1;             M4=1;

main:       I1=^data_1;
            CALL zero_delay;
            CALL do_filter;
            AX0=H#FFFF;
            DM(full_flag_1)=AX0;
            I1=^data_2;
            CALL zero_delay;
            CALL do_filter;
            AX0=H#FFFF;
            DM(full_flag_2)=AX0;
            JUMP main;                          {continue loop}

zero_delay: CNTR=%delay;
            AX0=0;
            DO zero_it UNTIL CE:
zero_it:       DM(I0,M1)=AX0;
            RTS;

do_filter:  CNTR=%data_1;
            DO filter_data UNTIL CE;
               CNTR=taps-1;
               SI=DM(I1,M0);
               DM(I0,M1)=SI;
               MR=0, MX0=DM(I0,M1), MY0=PM(I4,M4);
               DO tap_loop UNTIL CE;
tap_loop:         MR=MX0*MY0 (SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
               MR=MX0*MY0 (SS);
               IF MV SAT MR;
filter_data:   DM(I1,M1)=MR1;

            RTS;

.ENDMOD;
```

Listing 17.2 Source Code for Filter Processor

The array called *reserved_for_peak_module* is an array of common memory that used only by the peak processor to avoid conflicting memory allocations. Listing 17.3 shows the variable declarations for the peak processor. The two ranges of memory shared by the processors have identical declarations. The only difference is that the peak processor has the reserved space in the upper 1K of the dual-port area for the filter processor.

```
.MODULE/ABS=0                  peak_processor;

{Dual Port Memory Declarations}
.VAR/DM/RAM/ABS=0              full_flag_1,data_1[256];

.VAR/DM/RAM/ABS=1024           full_flag_2,data_2[256];
.VAR/DM/RAM/ABS=1281           reserved_for_filter_module[767];

            RTI;
            RTI;
            RTI;
            RTI;

            L0=0;
            M0=1;

main:       AX0=DM(full_flag_1);
            AR=PASS AX0;
            IF EQ JUMP main;
            I0=^data_1;
            CALL peak;
                           {Do something with the peak value}
            AX0=0;
            DM(full_flag_1)=AX0;
check_2:    AX0=DM(full_flag_2);
            AR=PASS AX0;
            IF EQ JUMP check_2;
            I0=^data_2;
            CALL peak;         {Do something with the peak value}
            AX0=0;
            DM(full_flag_2)=AX0;
            JUMP main;
```

```
peak:       CNTR=%data_1-1;
            AY0=DM(I0,M0);
            AR=PASS AY0;
            DO find_peak UNTIL CE;
               AF=AR-AY0, AY0=DM(I0,M0);
find_peak:     IF LT AR=PASS AY0;
            AF=AR-AY0;
            IF LT AR=PASS AY0;
            RTS;

.ENDMOD;
```

Listing 17.3 Source Code for Peak Processor

17.5.3 Simulation

The multiprocessing environment can be tested using the Simulator. When simulated, the filter program produces output data and stores it in the common data memory. You can then use the Simulator command to dump from data memory to store the dual-port data memory image on disk. Restart the Simulator, loading the peak processor program, and execute the Simulator command to reload the image of the common memory. Then simulate the peak processor program, which operates on the data generated by the filter program. A batch file that automatically executes this sequence of commands (using ADSP-2100 Cross-Software version 1.5x commands) is shown in Figure 17.4.

Command	*Comment*
`load filter`	Load filter program
`readimage test.dat`	Load memory image of input data
`run`	Execute filter program
`dumpdm full_flag_1 257 hold.dat i`	Write the file *hold.dat* with the flag and buffer data
`load peak`	Load peak program
`readimage hold.dat`	Load memory image of flag and data buffer from *hold.dat*
`run`	Execute peak program

Figure 17.4 Batch File Example

17 Multiprocessing