Technical notes on using Analog Devices DSPs, processors and development tools
Contact our technical support at dsp.support@analog.com and at dsptools.support@analog.com
Or visit our on-line resources http://www.analog.com/ee-notes and http://www.analog.com/processors

Host Communication via the Asynchronous Memory Interface for Blackfin® Processors

Contributed by Prashant Khullar and Jeff Sondermeyer

Rev 2 – March 29, 2004

Introduction

This Engineer-to-Engineer Note discusses the functionality and performance of an asynchronous memory interface developed for ADSP-BF531 / BF532 / BF533 Blackfin® processors. The interface is designed to provide a host port-like interface in applications that require a Blackfin processor to be used in conjunction with a host microcontroller. It can also be used to connect two Blackfin processors with minimal external circuitry. The maximum throughput of this implementation is 14.8 MB/s without concurrent bus activity and 8.3 MB/s while concurrent DMA activity is taking place. (e.g., simultaneous peripheral and memory DMA).

Higher bandwidth interfacing can be achieved with additional external logic or by using other Blackfin peripherals such as the SPORT, PPI, or external Bus Grant/Request.

Hardware Components

The basic architecture of this interface is shown in Figure 1. Two inexpensive 16-bit latches and some combinational logic are necessary for implementation. The host processor only requires an asynchronous memory port to send and receive data. Such an interface can be found on most microcontrollers.

Reads and writes to the data bus on both ends are regulated by the processor's external bus controller. The control signals are routed via the combinational logic elements to the LE and /OE pins on the latches (see Figure 1). Four general-purpose I/O pins are required for the implementation on both the Blackfin and Host processors. These are used to synchronize the interrupt-driven data transfers. Each processor must send and receive two types of interrupts: a Read, and a Read_Ack. The former is issued by the sending processor to indicate that a word of data has been latched into the data bus and may be read by the receiving processor. The latter is issued by the receiving processor to indicate that a word of data has been successfully read in and that the sending processor may send another word.

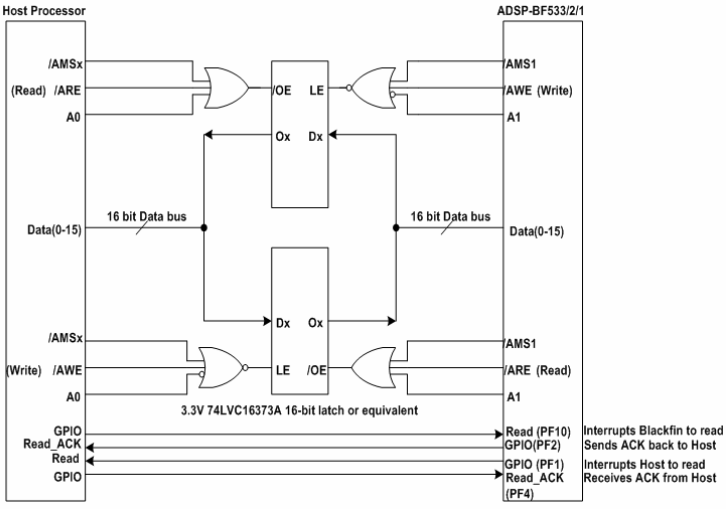

Figure 1. Host Interface Schematic

Software Requirements

Data transfer via the asynchronous interface must be controlled by an interrupt-driven software routine running on both processors. Reads and writes must be processed in interrupt service routines (ISRs). Interrupt requests of both types should be issued within the ISRs as well. The appropriate flag pins can simply be toggled to indicate interrupt requests.

Host-DSP API for Pointer versus Data

Using this interface, how do we distinguish between address and data information? The following paragraph describes one possible method.

Note: In this case, assume all interrupts are rising-edge-sensitive. When the host is sending address data to the DSP, the host Read flag pin can be left high until the host gets back a Read_Ack from the DSP. In this way, when the DSP still sees a high on the Read interrupt pin, it treats the incoming data as an address (a pointer to where the following data is to be placed). At that point, the host can send data, but this time it toggles the Read flag pin (high then low) so that the DSP sees only a low on the Read interrupt pin. In the DSP Read ISR, the code senses the logic level of the Read interrupt pin and sets up a new address pointer or places data at the next pointer location. This should be a clever way of differentiating between address pointers (where to put the following data) and the data itself. The only other consideration is that since the Blackfin external memory interface is only 16 bits wide, you need two 16-bit words to form a 32-bit address. In this way, the pointer/address can place code anywhere within the Blackfin memory map. To accomplish this, you set up and initialize a bit to zero (ADDR_BIT) in the DSP Read ISR so that the first time a new address is encountered, the code sets ADDR_BIT and stores the first 16 bits as the most significant word (MSW). Then, when the second 16 bits of the address arrive and ADDR_BIT = 1, it stores it as the least significant word (LSW). Following the receipt of the MSW, the bit is cleared in preparation for the next address that arrives. The reverse of this method is used for communicating from DSP back to the host.

MIPS Calculation

The worst-case ISR latency for entering and exiting an interrupt is 28 core clock (CCLK) cycles for Blackfin processors with a 10-stage execution pipeline. This latency includes pipeline refills and the return from interrupt (RTI). If a specific minimum transfer rate is required, this should be a high-priority interrupt that is "non-interruptible" to maintain a deterministic number of cycles. Assuming we use the API method discussed in the previous section, the worst case latency from the beginning to the end of an interrupt is 61 CCLK cycles for a single write operation and 75 CCLK cycles for a single read operation. Choosing an arbitrary transfer rate of 2.5 Mwords/s, we consume 75 cycles * 2.5M = 187 MIPS of DSP processing power moving a single 16-bit word of data from the host to the DSP in each interrupt.

It is evident from the above example that for any transfer rate that is on the order of a Mega-word per second, the number of MIPS consumed for the method depicted in Figure 1 is prohibitive. Therefore, under these conditions, adding a multi-word FIFO between the Host and the DSP is recommended. Obviously, the deeper the FIFO, the fewer MIPS are consumed, because more data is transferred in each interrupt and the overhead is minimized. For a Kilo-word per second transfer rate, a one-word FIFO is adequate. An equation has emerged:

If we let

NPI = Number of read/writes Per Interrupt (Note: this is equal to the FIFO depth)

NCS = Number of Context Saves/restores

RTR = Required Transfer Rate (Note: this must be less than the 8.3MB/s for a FIFO depth of 1)

FIL = Fixed Interrupt Latency = 28 CCLK cycles

RLC = Read Latency Constant = 15 CCLK cycles (Note: Wait = 1 system clock (SCLK), Hold = 2 SCLKs and the minimum latency for an asynchronous memory read is 2 SCLKs, CCLK/SCLK = 5/1)

WLC = Write Latency Constant = 0 CCLKs (Note: write operations do not stall the core)

GPR = General-Purpose Flag Read in Interrupt = 30 CCLKs or 0 CCLKs depending on whether you use this method (Note: CCLK/SCLK = 5/1)

DSP MIPs to read host =

[(*FIL*+*RLC*+*NPI*+*GPR*+(2**NCS*))**RTR*]/*NPI*

DSP MIPs to write host =

[(*FIL*+*WLC*+*NPI*+*GPR*+(2**NCS*))**RTR*]/*NPI*

Example: NPI=8, NCS=2, RTR= 2.5Mwords/s

DSP MIPS to read host = [(28+15+8+30+4) *2.5M]/8 = 26.5 MIPs with a GPIO read in the ISR.

Performance Evaluation

The performance of the asynchronous host interface was evaluated on a hardware prototype interconnecting two ADSP-BF533 processors. The software routines used to test the interface are included in Appendix B of this document.

The interface was tested in two real-time scenarios with another ADSP-BF533 processor acting as the host device. In the first, both processors simply idle until an interrupt request is received which is then processed in the appropriate ISR. In the second, one of the processors performs auto-buffered DMAs from external SDRAM to the parallel peripheral interface (PPI) *and* from L1 memory to SDRAM while waiting for interrupts. DMA traffic control is used to optimize bus sharing during these transfers. This second test case emulates the Blackfin processor's behavior in a scenario where a video application is passing encoded data from the host to DSP and vice-versa. The complete source code for these test cases is included in Appendix B.

Additionally, if processor latencies are known beforehand, a further performance enhancement can be made. Specifically, the writing processor can issue a 'Read' interrupt before it has actually completed its data write sequence. Since the reading processor has a known latency (in these tests, an ADSP-BF533 with a 6-cycle latency) involved in reading a flag-pin and triggering an ISR, the correct data is present on the bus once it is ready to read. Figures 2 and 3 illustrate the effect of this modification.

Without interspersed DMA activity *or* the performance enhancement, a complete 16-bit word transfer across the host interface takes 28 system clock (SCLK) cycles including the cycles associated with signaling using general-purpose I/O flags. With the performance enhancement, this can be reduced to 18 SCLK cycles. With interspersed bidirectional DMA, this increases to 32 SCLK cycles. At the maximum SCLK frequency of 133 MHz, this translates to throughputs of 9.5 MB/s, 14.8 MB/s, and 8.3 MB/s, respectively. In all cases, read and write sequences are configured to be 2 SCLK cycles in length. The time taken for a read/write cycle to commence upon receiving an interrupt is 6 SCLK cycles without interspersed DMA and 12 SCLK cycles with interspersed DMA. Issuing a flag-pin interrupt upon completion of a read or write sequence involves a 4-cycle latency in all cases. Figures 2 and 3 show snapshots of the data bus and control logic taken with a logic analyzer during transfer sequences in all test cases.

Conclusion

At a transfer rate of over 8 MB/s with full parallel DMA activity, this scheme can handle complex video algorithms. It is a very low-cost parallel approach that is fully asynchronous and bidirectional.

Appendix A. Timing Plots

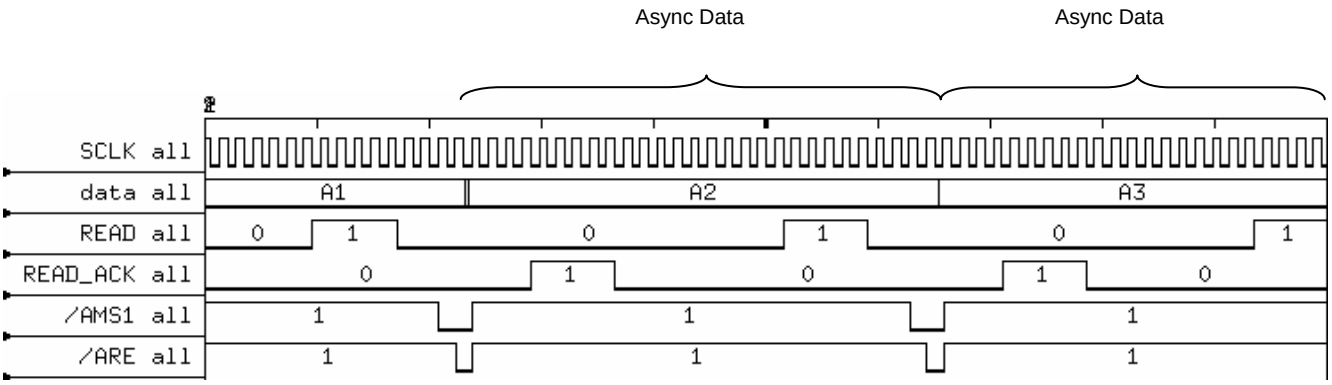

Figure 2. Asynchronous Transfer without Performance Enhancement (Reading Processor's View)

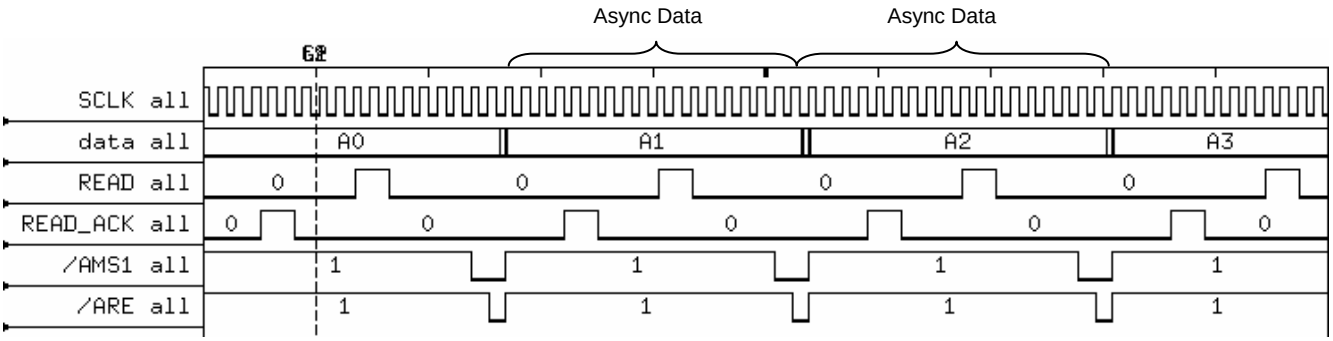

Figure 3. Asynchronous Transfer with Performance Enhancement (Reading Processor's View)

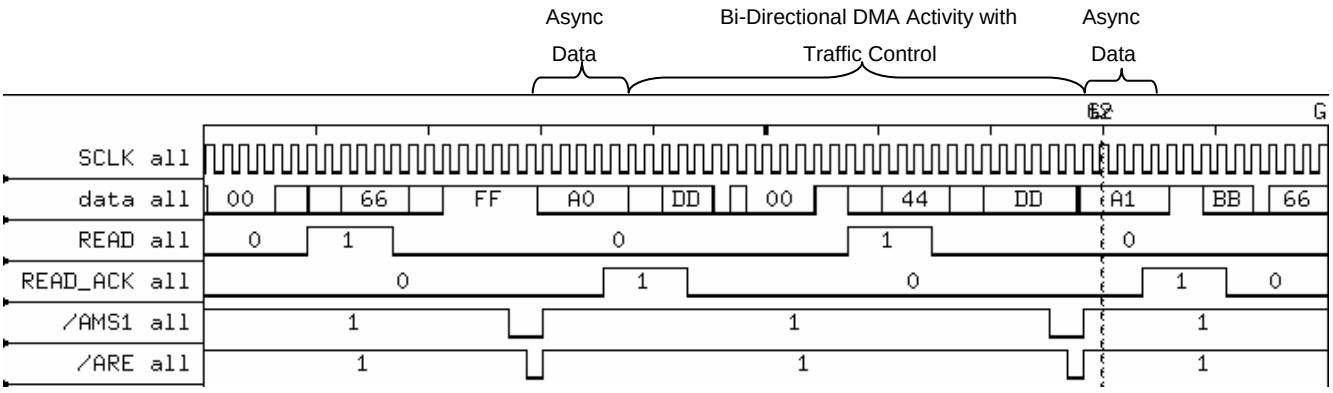

Figure 4. Asynchronous Data Transfer with Background DMA Activity (Reading Processor's View)

Appendix B. Code Listings

asyncreader.asm

```
/*****************************************************************************
 *
 * Copyright (c) 2003 Analog Devices Inc.  All rights reserved.
 *
*****************************************************************************/

// Asynchronous Memory Interface Test Code
//
// READ routine
//
// Last modified: 09/19/2003

#include "defBF533.h"

.section L1_code;
.global _main;

_main:

/* Assign 2 input (PF10, PF4) and 2 output flags (PF1, PF2) */
    P0.L = lo(FIO_DIR);
    P0.H = hi(FIO_DIR);
    R0.L = 0x0006;
    W[P0] = R0.L;

/* Enable input flagpins (PF10, PF4) */

    P0.L = lo(FIO_INEN);
    P0.H = hi(FIO_INEN);
    R0.L = 0x0410;
    W[P0] = R0.L;

/* Enable input flagpin 4 for Interrupt A generation */

    P0.L = lo(FIO_MASKA_S);
    P0.H = hi(FIO_MASKA_S);
    R0.L = 0x0010;
    W[P0] = R0.L;

/* Enable input flagpin 10 for Interrupt B generation */

    P0.L = lo(FIO_MASKB_S);
    P0.H = hi(FIO_MASKB_S);
    R0.L = 0x0400;
    W[P0] = R0.L;

/* Set ISR Address for PF interrupts*/

    P0.L = lo(EVT12);
    P0.H = hi(EVT12);
    R0.H = _ASYNC_READ;
    R0.L = _ASYNC_READ;
    [P0] = R0;
```

```
    /* Initialize EBIU */

    P0.L = lo(EBIU_AMBCTL1);
    P0.H = hi(EBIU_AMBCTL1);

    R0.L = 0x1112;           /* Set Read/Write sequences to be 2 SCLKs in length*/
    R0.H = 0x1112;           /* 1 SCLK setup + 1 SCLK read/write */

    [P0] = R0;
    SSYNC;

    P0.L = lo(EBIU_AMGCTL);
    P0.H = hi(EBIU_AMGCTL);
    R0.L = 0x00f6;
    W[P0] = R0;
    SSYNC;

/* Initialize asynchronous write data and address */

    R3 = 0x0000 (z);
    R2 = 0x0001 (z);
    P2.L = 0x0000;
    P2.H = 0x2020;

/* Initialize Pointer to SIC event register */

    P3.L = lo(SIC_ISR);
    P3.H = hi(SIC_ISR);
    P4.L = lo(FIO_FLAG_D);
    P4.H = hi(FIO_FLAG_D);

/* Initalize SDRAM registers. */

    //SDRAM Refresh Rate Control Register
    P0.L = lo(EBIU_SDRRC);
    P0.H = hi(EBIU_SDRRC);
    R0.L = 0x0817;
    W[P0] = R0.L;

    //SDRAM Memory Bank Control Register
    P0.L = lo(EBIU_SDBCTL);
    P0.H = hi(EBIU_SDBCTL);
    R0.L = 0x0013;
    W[P0] = R0.L;

    //SDRAM Memory Global Control Register
    P0.L = lo(EBIU_SDGCTL);
    P0.H = hi(EBIU_SDGCTL);
    R0.L = 0x998d;
    R0.H = 0x0091;
    [P0] = R0;

    //DMA0_START_ADDR
    R0.L = 0x0;
    R0.H = 0x0;
    P0.L = lo(DMA0_START_ADDR);
    P0.H = hi(DMA0_START_ADDR);
    [P0] = R0;
```

```
    //DMA0_CONFIG
    R0.L = 0x1000;  // Autobuffer mode, no DMA Interrupts
    P0.L = lo(DMA0_CONFIG);
    P0.H = hi(DMA0_CONFIG);
    W[P0] = R0.L;

    //DMA0_X_COUNT
    R0.L = 0xC;
    P0.L = lo(DMA0_X_COUNT);
    P0.H = hi(DMA0_X_COUNT);
    W[P0] = R0.L;

    //DMA0_X_MODIFY
    R0.L = 0x1;
    P0.L = lo(DMA0_X_MODIFY);
    P0.H = hi(DMA0_X_MODIFY);
    W[P0] = R0.L;

    /* PPI Control Register:  Output direction, 656 mode.   */
    P0.L = lo(PPI_CONTROL);
    P0.H = hi(PPI_CONTROL);
    R0.L = 0x2;
    W[P0] = R0.L;

    P1 = 0x0000 (z); // base of SDRAM
    R0.L = 0xBBBB;
    R0.H = 0xAAAA;
    [P1++] = R0;
    R0.L = 0xDDDD;
    R0.H = 0xCCCC;
    [P1++] = R0;
    R0.L = 0xFFFF;
    R0.H = 0xEEEE; // Write known Patterns to SDRAM
    [P1++] = R0;

    P1.L = 0x0000;
    P1.H = 0xFF80; // Base of L1 Data Bank A
    R0.L = 0x1122; // known words
    R0.H = 0x3344;
    [P1++] = R0;
    R0.L = 0x5566;
    R0.H = 0x1122;
    [P1++] = R0;
    R0.L = 0x3344;
    R0.H = 0x5566;
    [P1++] = R0;

/* Set up MemDMA from L1 to SDRAM */

/* MDMA_S0 Start Address */
    R0.L = 0x0000;
    R0.H = 0xFF80;
    P0.L = lo(MDMA_S0_START_ADDR);
    P0.H = hi(MDMA_S0_START_ADDR);
    [P0] = R0;

/* MDMA_S0_X_COUNT */
    R0.L = 0xC;
```

```
    P0.L = lo(MDMA_S0_X_COUNT);
    P0.H = hi(MDMA_S0_X_COUNT);
    W[P0] = R0.L;

/* MDMA_S0_X_MODIFY */
    R0.L = 0x1;
    P0.L = lo(MDMA_S0_X_MODIFY);
    P0.H = hi(MDMA_S0_X_MODIFY);
    W[P0] = R0.L;

/* MDMA_D0 Start Address */
    R0.L = 0x0010;
    R0.H = 0x0000;
    P0.L = lo(MDMA_D0_START_ADDR);
    P0.H = hi(MDMA_D0_START_ADDR);
    [P0] = R0;

/* MDMA_D0_X_COUNT */
    R0.L = 0xC;
    P0.L = lo(MDMA_D0_X_COUNT);
    P0.H = hi(MDMA_D0_X_COUNT);
    W[P0] = R0.L;

/* MDMA_D0_X_MODIFY */
    R0.L = 0x1;
    P0.L = lo(MDMA_D0_X_MODIFY);
    P0.H = hi(MDMA_D0_X_MODIFY);
    W[P0] = R0.L;

/* Enable system PF and PPI DMA interrupts */

    P0.L = lo(SIC_IMASK);
    P0.H = hi(SIC_IMASK);
    R0 = [P0];
    bitset(r0,8);
    bitset(r0,19);
    bitset(r0,20);
    [P0] = R0;

/* Enable core PF and PPI DMA interrupts */

    P0.L = lo(IMASK);
    P0.H = hi(IMASK);
    R0 = [P0];
    bitset (R0,8);
    bitset(R0,12);
    [P0] = R0;    // All inits are complete and interrupts are enabled after
                  // this line. You may insert a software breakpoint here for
                  // testing purposes.

/* Set up DMA Traffic Control */

    R0.L = 0x2222;
    P0.L = 0x0B0C;
    P0.H = 0xFFC0;
    W[P0] = R0.L;

//Enable MemDMA Source
    R0.L = 0x1001;
```

```
    P0.L = lo(MDMA_S0_CONFIG);
    P0.H = hi(MDMA_S0_CONFIG);
    //W[P0] = R0.L;     // Uncomment this line to enable MemDMA from L1
                        // memory to SDRAM
    ssync;

//Enable MemDMA Destination

    R0.L = 0x1003;
    P0.L = lo(MDMA_D0_CONFIG);
    P0.H = hi(MDMA_D0_CONFIG);
    //W[P0] = R0.L;     // Uncomment this line to enable MemDMA from L1
                        // memory to SDRAM
//Enable Peripheral DMA
    P0.L = lo(DMA0_CONFIG);
    P0.H = hi(DMA0_CONFIG);
    R0.L = W[P0];
    bitset(R0,0);
    //W[P0] = R0.L;     // Uncomment this line to enable autobuffered DMA transfers
                        // from SDRAM to PPI
    ssync;

//Enable PPI
    P0.L = lo(PPI_CONTROL);
    P0.H = hi(PPI_CONTROL);
    R0.L = W[P0];
    bitset(R0,0);
    //W[P0] = R0.L; // Uncomment this line to enable autobuffered DMA transfers
                    // from SDRAM to the PPI
    ssync;

    wait:
    jump wait;

/* Perform reads in an ISR */

    _ASYNC_READ:

    /* Read from Latch */

    R3 = W[P2] (z);

    /* Send Read_Ack to Host */

    R0.L = 0x2;
    W[P4] = R0.L;
    ssync;
    R0.L = 0x0;
    W[P4] = R0.L;

    RTI;

_main.end:
```

Listing 1. asyncreader.asm

asyncwriter.asm

```
/*****************************************************************************
 *
 * Copyright (c) 2003 Analog Devices Inc.  All rights reserved.
 *
*****************************************************************************/

// Asynchronous Memory Interface Test Code
//
// WRITE routine
//
// Last modified: 09/19/2003

#include "defBF533.h"

.section L1_code;
.global _main;

_main:

/* Assign 2 input (PF10, PF4) and 2 output flags (PF1, PF2) */
    P0.L = lo(FIO_DIR);
    P0.H = hi(FIO_DIR);
    R0.L = 0x0006;
    W[P0] = R0.L;

/* Enable input flagpins (PF10, PF4) */

    P0.L = lo(FIO_INEN);
    P0.H = hi(FIO_INEN);
    R0.L = 0x0410;
    W[P0] = R0.L;

/* Enable input flagpin 4 for Interrupt A generation */

    P0.L = lo(FIO_MASKA_S);
    P0.H = hi(FIO_MASKA_S);
    R0.L = 0x0010;
    W[P0] = R0.L;

/* Enable input flagpin 10 for Interrupt B generation */

    P0.L = lo(FIO_MASKB_S);
    P0.H = hi(FIO_MASKB_S);
    R0.L = 0x0400;
    W[P0] = R0.L;

/* Set ISR Address for PF interrupts*/

    P0.L = lo(EVT12);
    P0.H = hi(EVT12);
    R0.H = _ASYNC_WRITE;
    R0.L = _ASYNC_WRITE;
    [P0] = R0;

/* Initialize EBIU */

    P0.L = lo(EBIU_AMBCTL1);
```

```
    P0.H = hi(EBIU_AMBCTL1);

    R0.L = 0x1112;          /* Set Read/Write sequences to be 2 SCLKs in length*/
    R0.H = 0x1112;          /* 1 SCLK setup + 1 SCLK read/write */
    [P0] = R0;
    SSYNC;

    P0.L = lo(EBIU_AMGCTL);
    P0.H = hi(EBIU_AMGCTL);
    R0.L = 0x00f6;
    W[P0] = R0;
    SSYNC;

/* Initialize asynchronous write data and address */

    R3 = 0x0000 (z);
    R2 = 0x0001 (z);
    P2.L = 0x0000;
    P2.H = 0x2020;

/* Initialize Pointer to SIC event register */

    P3.L = lo(SIC_ISR);
    P3.H = hi(SIC_ISR);
    P4.L = lo(FIO_FLAG_D);
    P4.H = hi(FIO_FLAG_D);

/* Initalize SDRAM registers. */

    //SDRAM Refresh Rate Control Register
    P0.L = lo(EBIU_SDRRC);
    P0.H = hi(EBIU_SDRRC);
    R0.L = 0x0817;
    W[P0] = R0.L;

    //SDRAM Memory Bank Control Register
    P0.L = lo(EBIU_SDBCTL);
    P0.H = hi(EBIU_SDBCTL);
    R0.L = 0x0013;
    W[P0] = R0.L;

    //SDRAM Memory Global Control Register
    P0.L = lo(EBIU_SDGCTL);
    P0.H = hi(EBIU_SDGCTL);
    R0.L = 0x998d;
    R0.H = 0x0091;
    [P0] = R0;

    //DMA0_START_ADDR
    R0.L = 0x0;
    R0.H = 0x0;
    P0.L = lo(DMA0_START_ADDR);
    P0.H = hi(DMA0_START_ADDR);
    [P0] = R0;

    //DMA0_CONFIG
    R0.L = 0x1000;  // Autobuffer mode, no DMA Interrupts
    P0.L = lo(DMA0_CONFIG);
    P0.H = hi(DMA0_CONFIG);
```

```
    W[P0] = R0.L;

    //DMA0_X_COUNT
    R0.L = 0xC;
    P0.L = lo(DMA0_X_COUNT);
    P0.H = hi(DMA0_X_COUNT);
    W[P0] = R0.L;

    //DMA0_X_MODIFY
    R0.L = 0x1;
    P0.L = lo(DMA0_X_MODIFY);
    P0.H = hi(DMA0_X_MODIFY);
    W[P0] = R0.L;

    /* PPI Control Register:  Output direction, 656 mode.   */

    P0.L = lo(PPI_CONTROL);
    P0.H = hi(PPI_CONTROL);
    R0.L = 0x2;
    W[P0] = R0.L;

    P1 = 0x0000 (z); // base of SDRAM
    R0.L = 0xBBBB;
    R0.H = 0xAAAA;
    [P1++] = R0;
    R0.L = 0xDDDD;
    R0.H = 0xCCCC;
    [P1++] = R0;
    R0.L = 0xFFFF;
    R0.H = 0xEEEE; // Write known Patterns to SDRAM
    [P1++] = R0;

/* Enable system PF and PPI DMA interrupts */

    P0.L = lo(SIC_IMASK);
    P0.H = hi(SIC_IMASK);
    R0 = [P0];
    bitset(r0,8);
    bitset(r0,19);
    bitset(r0,20);
    [P0] = R0;

/* Enable core PF and PPI DMA interrupts */

    P0.L = lo(IMASK);
    P0.H = hi(IMASK);
    R0 = [P0];
    bitset (R0,8);
    bitset(R0,12);
    [P0] = R0;  // All inits are complete and interrupts are enabled after this
                // line
              // You may insert a software breakpoint here for testing purposes.

//Enable DMA
    P0.L = lo(DMA0_CONFIG);
    P0.H = hi(DMA0_CONFIG);
    R0.L = W[P0];
    bitset(R0,0);
    //W[P0] = R0.L;    // Uncomment this line to enable DMA from SDRAM to PPI
```

```
    ssync;

//Enable PPI
    P0.L = lo(PPI_CONTROL);
    P0.H = hi(PPI_CONTROL);
    R0.L = W[P0];
    bitset(R0,0);
    //W[P0] = R0.L; // Uncomment this line to enable DMA from SDRAM to PPI
    ssync;

    wait:
    jump wait;

/* Perform writes in an ISR */

_ASYNC_WRITE:

    /* Write to Latch */

    W[P2] = R2;
    ssync;   // Comment this line to engage performance enhancement
    R2+=1;

    /* Inform Host of available data */

    R0.L = 0x4;
    W[P4] = R0.L;
    ssync;

    R0.L = 0x0;
    W[P4] = R0.L;

    RTI;

_main.end:
```

Listing 2. asyncwriter.asm

References

[1] *ADSP-BF533 Blackfin Processor Hardware Reference.* Revision 1.0, December 2003. Analog Devices, Inc.

Document History

Revision	Description
Rev 2 – March 29, 2004 *by J. Sondermeyer*	Added Host-DSP API method Added MIPS Calculation section
Rev 1 – October 21, 2003 *by P. Khullar*	Initial Release