Technical Notes on using Analog Devices' DSP components and development tools
Contact our technical support by phone: (800) ANALOG-D or e-mail: dsp.support@analog.com
Or visit our on-line resources http://www.analog.com/dsp and http://www.analog.com/dsp/EZAnswers

Interfacing the ADSP-BF535 Blackfin® Processor to High-Speed Converters (like those on the AD9860/2) over the External Memory Bus

Contributed by Jeff Sondermeyer, Jeritt Kent, Martin Kessler, and Rick Gentile *June 5, 2003*

Introduction

In the 1970's and 80's high speed mixed signal designs were often constrained by digital circuitry limitations, not analog. High-speed parallel converters (>10MSPS), for example, have been available from industry leaders like Analog Devices Inc. (NYSE: ADI), since the 1970's. More and more applications are demanding intensive real-time algorithms. In addition, higher sample rates (14 bits at greater than 50MSPS) are available from both analog-to-digital converters (ADCs) and digital-to-analog converters (DACs). These factors mandate faster programmable general-purpose (GP) digital signal processors (DSPs) to handle the challenges presented by these high-speed designs. Until recently, most designers were forced to interface high-speed parallel converters to Application Specific ICs (ASICs) or fast Field Programmable Gate Arrays (FPGAs). Devices like these are capable of resolving the many required simultaneous parallel digital operations but are often inflexible and can be prohibitively expensive.

Now, with the new Blackfin® processor architecture, ADI has a family of programmable GP processors that combines 16-bit fixed-point DSP and 32-bit RISC processing with a unified instruction set model. The first instantiation of this family, the ADSP-BF535 Blackfin processor can process the sustained input/output (I/O) and core throughputs required to process data from many of these converters. In particular, the ADSP-BF535 Blackfin processor's core can be clocked at 300MHz. Depending on the core clock frequency, a maximum I/O or system clock (SCLK) of 133MHz can be achieved. This SCLK should not to be confused with the serial clock for the Serial Peripheral Interface (SPI).

Advantages of using a GP DSP

One of the biggest advantages of GP programmable processors is that these solutions are typically much lower in cost than their closest digital processing counterparts, FPGAs and ASICs. Additionally, GP DSP design cycles are much shorter which allows for a faster time to market. Some companies must hire or consult professionals with specialized skills to design FPGAs/ASICs. Companies may even be forced to send their intellectual property (IP) out-of-house involving certain risks in confidentiality (hardware, firmware, and software). On the other hand, GP DSP code can be converted to Read-Only-Memory-based (ROM) or be masked into a DSP, like the ADSP-BF53x, which further protects IP. Finally, GP processors are fully programmable, unlike an ASIC implementation, where every change requires a costly redesign (time and money). These factors, alone, are driving many engineers to reconsider GP DSP as the solution of choice, especially as GP DSPs approach "Pentium class" core rates.

Generally speaking, the Blackfin processor needs to be clocked minimally an order of magnitude (10X) faster than the converter's sample rate to guarantee sufficient data processing bandwidth. Obviously, the amount of processing bandwidth needed is dependent upon the processors interface capabilities which is, in turn, influenced by several other factors including: block processing versus sample processing, the existence of a Direct Memory Access (DMA) controller, multi-ported memory, and whether external FIFOs are used. Fortunately, the first instantiation of ADI's Blackfin processor family, the ADSP-BF535, has a full DMA controller that operates independent of the core with multi-ported Level 1 (L1) and Level 2 (L2) memories. The combination of core speed, an independent DMA controller, and a large multi-ported on-board memory (308K bytes) allows the ADSP-BF535 to perform efficient block processing at high data rates. For example, if the Revision 2.2-compliant, 33MHz, 32-bit Peripheral Component Interconnect (PCI) interface is used (not shown in this application), transfer bandwidths can be achieved that approach 132MB/sec.

Copyright 2003, Analog Devices, Inc. All rights reserved. Analog Devices assumes no responsibility for customer product design or the use or application of customers' products or for any infringements of patents or rights of others which may result from Analog Devices assistance. All trademarks and logos are property of their respective holders. Information furnished by Analog Devices Applications and Development Tools Engineers is believed to be accurate and reliable, however no responsibility is assumed by Analog Devices regarding technical accuracy and topicality of the content provided in Analog Devices' Engineer-to-Engineer Notes.

Unlike the ADSP-BF531/BF532/BF533 Blackfin processors, which have a dedicated Parallel Peripheral Interface (PPI) to connect directly to high-speed converters, the ADSP-BF535 does not. However, the External Bus Interface Unit (EBIU) of the ADSP-BF535 provides interfaces to asynchronous (ASYNC) external memories. If the PCI bus must be used for other system communications, the EBIU is the only available parallel interface to connect the ADSP-BF535 to a high-speed converter. Combining the DSP-mastered, asynchronous control of this port with the synchronous, continuous data stream of converters provides a challenge for system designers.

This application note will cover one particular hardware implementation utilizing a low pin count, low cost Programmable Array Logic (PAL), Complex Programmable Logic Device (CPLD), or FPGA. This logic will perform the control functions between the AD9860/2 Mixed Signal Front-End (MxFE™) and the ASYNC external memory bus of the ADSP-BF535. Although the DSP code will not be discussed in this note, the assembly code is included in the *Appendix* as a reference. The application depicted in Figure 1 below is for an Orthogonal Frequency Division Multiplexed (OFDM) wireless portable terminal. Please note that the ADC and DAC were time-shared (Time Division Multiplexed or TDM) over the ASYNC interface of the DSP. (*The information given here applies equally to other parallel high-speed ADCs and DACs.*)

This engineering note assumes that the reader has prior knowledge of the ADSP-BF535 and the AD9860/2. If you are unfamiliar with the ADSP-BF535, please refer to the "ADSP-BF535 Blackfin® Processor Hardware Reference". The datasheet for the AD9860/2 can be found at www.analog.com

Design Goals

One of the early design goals for this project was to minimize the amount of external control logic necessary to interface the DSP and the converter(s). Driven by cost, engineering wanted to eliminate any FIFOs or memory within the external logic device. An additional constraint was to avoid routing the data buses through the logic thereby reducing the number of pins, package size, and cost of the logic device. The initial design shown in Figure 1 combines all functions (including data latching) into a single logic device. However, production models of this design will utilize inexpensive tri-state-able latches driven by a logic device. These latches or buffers will multiplex (pack) the samples from the DSP memory interface to the 12/14-bit DAC as well as buffer or de-multiplex (unpack) the 10/12-bit ADC samples to the DSP memory interface.

Design Challenges

One of the key factors in any mixed-signal/DSP design is a solid understanding of the trade-offs between the devices. The following discussion will illustrate the various tradeoffs that must be considered when interfacing ADCs/DACs to the ADSP-BF535.

The OFDM modulation scheme for this design drove the converter sample rate for this application to be 15.36MSPS. The AD9860/2 has a dual 10/12-bit, 64MSPS ADC as well as a dual 12/14-bit, 128MSPS DAC. Unlike ADI's SHARC® DSPs that have a DMA-Request and DMA-Grant (i.e. DMA can be mastered from external device), the ADSP-BF535 Blackfin processor only has one set of internal memory DMA channels (MemDMA), which must be mastered from the DSP. In addition, when the ADSP-BF535 ASYNC interface is connected to devices that do contain FIFOs or memory, all latencies must be understood. Every time the MemDMA relinquishes the bus after a burst of eight (8) transfers, it requires ten (10) SCLK cycles to begin the next transfer.

Future Blackfin processor derivatives will have programmable priority levels for the DMA controller as well as a dedicated high-speed parallel interface with DMA-Request and DMA-Grant signaling. With a dedicated PPI on future Blackfin processor products, the ASYNC memory interface will not be required to connect to parallel converters.

This approach assumes that the memory interface is dedicated to the converters. Multiplexing external SRAM/SDRAM memory with the converter(s) would be difficult and is not recommended, especially considering that there is only one MemDMA, and it would need to be shared. The existence of a large on-board L2 memory (256K bytes) minimizes the need for any external memory. However, multiplexing the parallel converter(s) with a Flash or EPROM for boot purposes is permissible.

This design uses a TDM time-slice approach for sharing the external bus between the ADCs and the DACs. Simultaneous access is not possible with the ADSP-BF535 because, as mentioned previously, there is only one memory interface that either does a read or a write, and there is only one set of MemDMA channels (source and destination).

The ADSP-BF535 will support a maximum SCLK of 133MHz (peak DMA bandwidth). At this rate, and with no external FIFO, the MemDMA could sustain a transfer (32 bit word) rate of 133M/10 (nine cycles are required for bus acquisition plus one to make the next transfer) or 13.3M words/second. Note, however, that the SCLK of the ADSP-BF535 is derived from the core clock (CCLK). Here, there are four available divide ratios: 2, 2.5, 3, and 4.

As a result, one possible combination of CCLK and divisor that will allow a 133MHz SCLK is CCLK = 266MHz and CCLK/SCLK = 2. If the core must run at 300MHz, the highest SCLK that can be obtained is 120MHz (divisor=2.5) to stay under the maximum 133MHz. Now, since the ASYNC memory interface is 32 bits wide, one can pack up to two 16-bit samples (in this case I and Q) into each word. This effectively halves the word rate that the DSP must process (with a 15.36MSPS converter sample rate, the DSP will "see" 7.68MSPS). The highest external converter sample rate, though, that the MemDMA will support under these conditions is 2 * 120/10 = 24MSPS. Furthermore, the SCLK must be an integer multiple of the converter sample rate to ensure proper phase alignment between converter timing and DSP timing and eliminate the need for any external FIFOs. So, if the 120MHz SCLK must be evenly divisible by the sample rate, the highest even divisor of 120 is 12. Therefore, the highest converter sample rate that the ADSP-BF535 will support is 2 * 12M = 24MSPS. Again, the DSP will only process half of this rate, 12MSPS, and this is, in fact, equal to the maximum rate that the MemDMA can sustain, 12M words/second. Note that higher sample rates can be processed by the ADSP-BF535 by including small external FIFOs between the converter(s) and the EBIU.

It is again noted that this application dictated a 15.36MSPS converter sample rate driven by OFDM requirements. To obtain a SCLK that is an integer multiple of this converter sample rate, then, one must choose a Phase Locked Loop (PLL) Multiplier that is an integer multiple, in turn, of one of the four available divisor ratios (2, 2.5, 3, or 4). The maximum CCLK allowed is 276.48MHz (using a PLL multiplier of 18). This, in turn, limits the SCLK to the integer multiple 276.48/3 = 92.16MHz (a divide ratio of 2 would give an SCLK over the 133MHz maximum). Under these constraints, the maximum sustained rate that the MemDMA can support is 92.16/10 = 9.21 words/second.

DMA Considerations

Careful consideration must be given to the combined, required, "sustained" DMA performance. Since the MemDMA is a shared resource over the DMA bus (DAB), other DMA activity is arbitrated on this bus. This application required a 10Mbit/second serial channel on a SPORT that also must arbitrate for the DAB. This will consume an additional 625K words/second at 16 bits/word of DMA bandwidth. The ADSP-BF535 Blackfin processor supports a total of 133M words/second peak DMA bandwidth, and the SPORT has higher arbitration priority over the MemDMA (see Table 1). Given this, the SPORT DMA should effectively utilize the ten-cycle (10) delay previously discussed and allow most, if not all, of the 9.21M words/seconds to be used by the MemDMA. There are 9.21M – 7.68M (15.36/2) = 1.53M words/second of additional bandwidth which should provide enough margin for a sustained 7.68MSPS.

DAB Master	Arbitration Priority
SPORT0 RCV DMA Controller	0 - highest
SPORT1 RCV DMA Controller	1
SPORT0 XMT DMA Controller	2
SPORT1 XMT DMA Controller	3
USB DMA Controller	4
SPI0 DMA Controller	5
SPI1 DMA Controller	6
UART0 RCV Controller	7
UART1 RCV Controller	8
UART0 XMT Controller	9
UART1 XMT Controller	10
Memory DMA Controller	11 - lowest

Table 1: Arbitration Priority

Analysis of the DMA engine within the ADSP-BF535 reveals a few other considerations. While the DMA engine supports two types of DMA transfers: descriptor-based and autobuffer-based, the ADSP-BF535 MemDMA controller does not support autobuffer-based DMA. Therefore, descriptor-based transfers must be used. The descriptor fetch from L1/L2 memory involves two (2) five-word block moves, one for the source descriptor and another for the destination descriptor. Additionally, the MemDMA has a 16-entry 32-bit FIFO that is filled from the source and emptied from the destination. If both descriptors are loaded simultaneously, this requires 39 SCLK cycles (worst case) from L2. The destination descriptor load has priority over the source load to avoid overrunning the FIFO. Thus, in this example, the amount of time required to load both descriptors simultaneously is 1/92.16M * 39 = 423 nanoseconds. The DMA engine descriptor load performance is best when the descriptors are loaded from L2 memory. If the descriptors are located in L1 memory, there are additional delays. The source plus destination descriptors' load time from L1 is 65 SCLK cycles worst case. To effectively process data at these sample rates, ping-pong buffers are normally used (this design utilizes two (2) 1024-word buffers). This technique allows data to be filled into one buffer while the core processes the other buffer. See the *Appendix* for the Blackfin assembly code that utilizes the MemDMA to pull data in from an ADC and "ping-pongs" between two internal buffers. As a reference, the complete VisualDSP++™ 3.1 project is available from ADI.

There are two phases of operation that must be analyzed. First, samples must be received from the ADC into the DSP (receiver TDM phase) and secondly, samples must be

transmitted from the DSP to the DAC (transmitter TDM phase).

Receiver TDM Phase

During the receive phase (i.e. data from the ADC), the data movement is in this direction: ADC → EBIU (source) → MemDMA → FIFO → L1/L2 (destination). At the 15.36MHz converter sample rate, a new 32-bit sample arrives at the DSP every 1/7.68M = 130.2 nanoseconds. As seen from the descriptor load time latency, 423 nanoseconds, something must be done to avoid overrunning the DSP and losing samples. Fortunately, the converters are attached to an external bus, and the address bus is not being used. Thus, when moving samples into the DSP, one can setup the source descriptor with the maximum transfer count, 65536 words, and destination descriptor with intended ping-pong buffer transfer size, 1024 words. In this way, upon interrupt from the core every 1024 words, only the destination descriptor is reloaded, and the load time is reduced to 20 SCLKs * 1/92.16M = 217 nanoseconds. As previously mentioned, this design utilizes a TDM scheme in which the ADC and DAC occupy time slices. The multiplex rate is a variable 5-8 milliseconds. Since the ADC and DAC data is interleaved, at a worst case, the interface changes from receiver to transmitter and vice versa every 8 milliseconds. Therefore, 65536 words * 130.2 nanoseconds or 8.5 milliseconds is adequate time, and the source descriptor only needs to be setup once at the beginning of each receiver TDM phase. Finally, the 16-entry MemDMA FIFO "hides" the destination descriptor load time because the source is still filling the FIFO while the destination descriptor is being loaded from memory. In a worst-case scenario, the MemDMA FIFO will only accumulate a few samples of data before the descriptor is reloaded. Then, these samples are burst into memory. So, the need for an external FIFO on the receiver side is eliminated and no samples are lost.

Transmitter TDM Phase

During the transmit phase (i.e. DAC), data movement is in the opposite direction: L2/L1 (source) → MemDMA → FIFO → EBIU (destination) → DAC. Unlike the previous mode, the source descriptor must be updated every 1024 words. This will require 20 SCLK cycles or 217 nanoseconds. However, since the MemDMA (9.21M words/second) is running slightly faster than the sample rate (7.68M words/second), this should maintain 16 samples in the MemDMA FIFO, which will feed the DAC while the descriptor loads. The destination descriptor transfer count can be fixed at 65536 words. Again, no external FIFO is required and no samples are lost.

Logic Overview and Timing

In avoiding the need for FIFOs in the external logic, it is still important to synchronize the converter clocks to the DSP SysCLKSCLK. Depending on the sample rate, this limits the available ADSP-BF535 clocking options. Minimally, SCLK must be evenly divisible by the converter sample rate, and CCLK may need to be evenly divisible by the converter sample rate as well (there is only one non-integer divisor, 2.5, and this may not be useable in some cases). External latches or buffers must be used to align the data from the converters with the timing of the DSP (See Figures 2 and 3 for sample skew and delay). The four-wire DSP SPI port is directly connected to the AD9860/2 SPI port. To ensure proper power sequencing and initialization, the DSP should reset the converter(s). In an effort to further reduce the pin count of the external logic, another option available on the AD9860/2 (not shown here) allows two (2) 10/12-bit ADC values to be time-multiplexed onto a single 10/12-bit RXDATA bus. This would eliminate one of the two (2) 10/12-bit buses, but it requires the external logic to de-multiplex the data before it is transmitted to the DSP.

All data movement is controlled or mastered by the MemDMA within the Blackfin processor. When the ADC data is read (see Figure 2), the external logic must drive the data and the ARDY signal. The external logic must sample the /AOE pin to check when data can be driven to the ADSP-BF535. The /AOE signal indicates to the external logic that the DMA controller is ready to take data. The receiver three-state machine is shown at the bottom of Figure 2.

When data is being sent out (see Figure 3) to the DAC, the external logic has to sample the /AWE signal and then drive ARDY. /AWE indicates to the external logic when the DMA controller is ready with new data. The transmitter four-state machine is shown at the bottom of Figure 3.

Conclusions

Even though the ADSP-BF535 Blackfin processor was not specifically designed to interface to high-speed parallel converters, this engineering note provides a low-cost "FIFO-free" solution for interfacing to both ADCs and DACs with sample rates up to 24MSPS if the core is constrained to run at 300MHz. If the ADSP-BF535 core can be clocked specifically at 264MHz, then the highest converter sample rate is limited only by the maximum SCLK that the ADSP-BF535 can support (133MHz) and the inter-burst 10-cycle MemDMA latency: 2 * 133M/10 = 26.6MHz.

In summary, the following table is included as a set of rules specific to the ADSP-BF535 Blackfin processor:

The Ten Commandments of Interfacing to the ADSP-BF535

1) The ADSP-BF535's maximum allowed core frequency is 300MHz.
2) ADSP-BF535's maximum allowed system clock (SCLK) is 133MHz.
3) ADSP-BF535's MemDMA has a worst-case ten-cycle (10) reacquisition latency every time the DMA bus is relinquished.
4) ADSP-BF535's SCLK shalt be derived from CCLK, and there are four available divide ratios between CCLK and SCLK (2, 2.5, 3, and 4).
5) If the ADSP-BF535's core shalt run at the maximum (300MHz), then the maximum SCLK is 120MHz. See Commandments 1, 2, and 4.
6) The maximum ADSP-BF535 MemDMA rate is 133M/10 = 13.3Mwords/sec. See Commandments 2 and 3.
7) In order to obtain the fastest ADSP-BF535 SCLK (133MHz) and MemDMA transfer rate (13.3Mwords/sec), the core shalt not run at maximum, but at 266MHz with a CCLK/SCLK divide ratio of 2. "Only" 34 MIPs are not utilized!
8) When not using external FIFOs, the ADSP-BF535's core shalt operate minimally at an order of magnitude (10X) greater than the greatest external interfacing converter sample rate to provide sufficient processing bandwidth.
9) To eliminate the need for external FIFOs, the ADSP-BF535's SCLK shalt be an integer multiple of the external interfacing converter sample rates to ensure proper phase alignment between converter timing and DSP timing.
10) To halve the sample rate that the DSP must process, the external logic shalt pack up to two 16-bit samples into each 32-bit word.

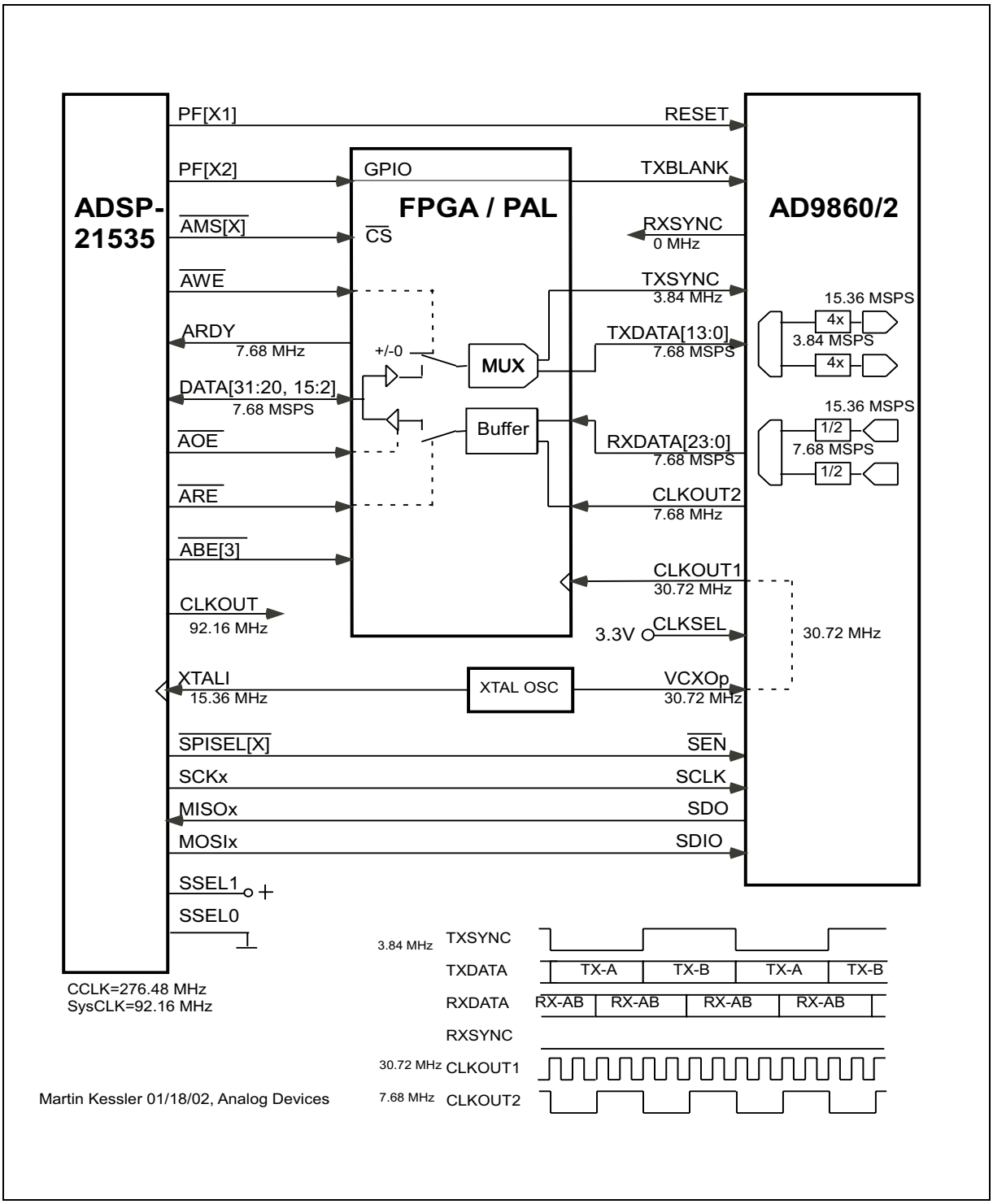

Figure 1: External Logic

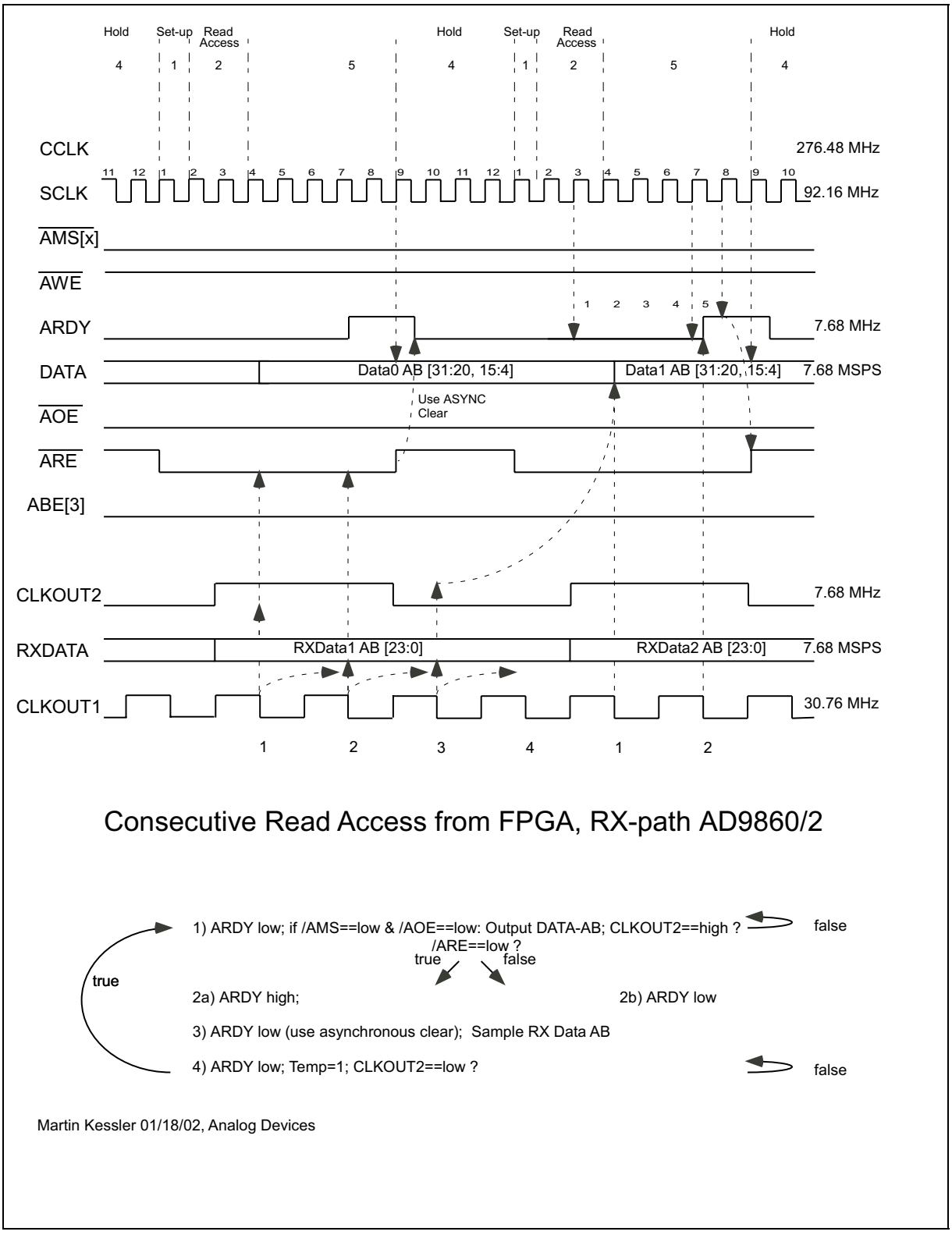

Figure 2: Receive Timing and State Machine

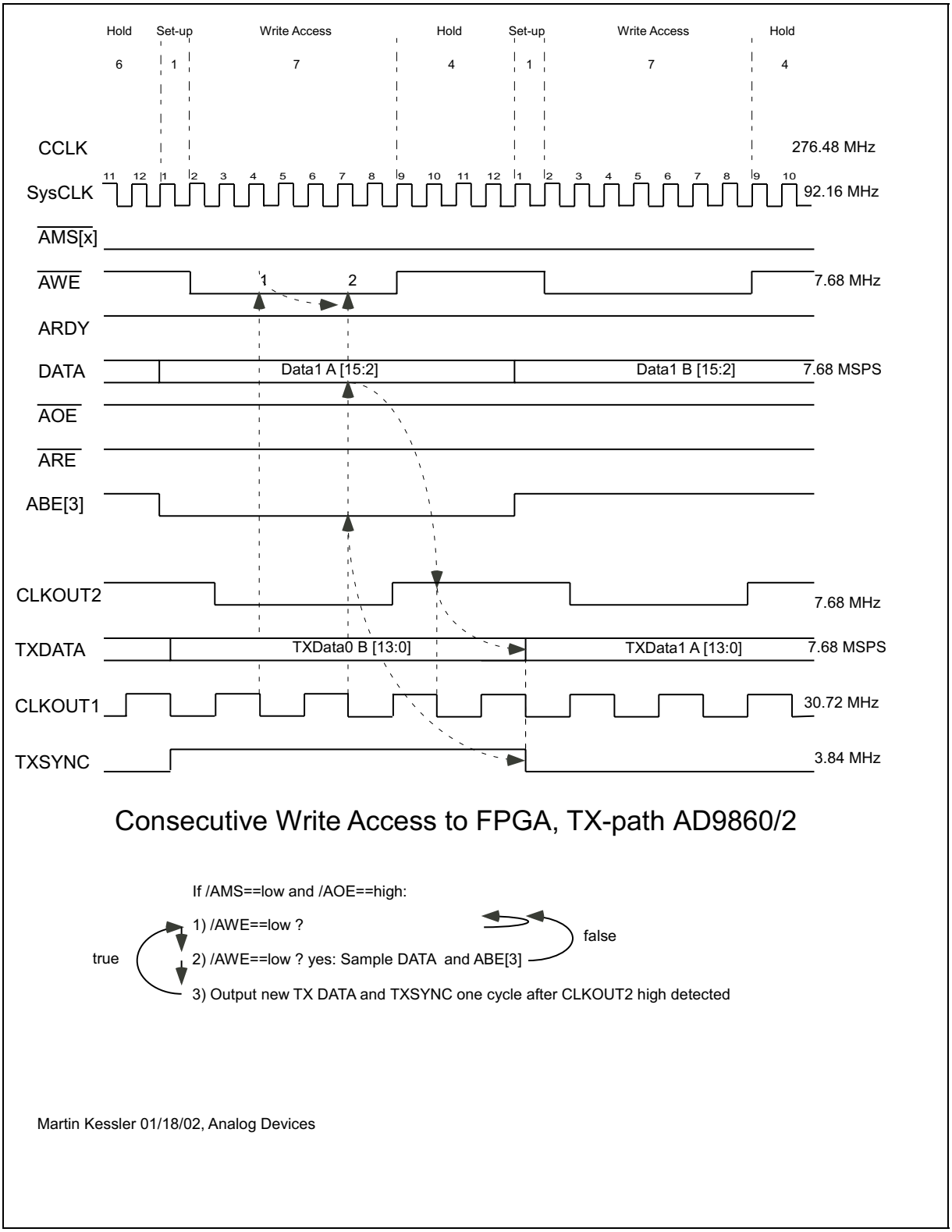

Figure 3: Transmit Timing and State Machine

Appendix

```
#include <defBF535.h>
#define RX_DMA_LENGTH_SOURCE       65535
#define RX_DMA_LENGTH_DESTINATION  1023
#define TX_DMA_LENGTH_SOURCE       1023
#define TX_DMA_LENGTH_DESTINATION  65535

#define RX_MODE  1
#define TX_MODE  2
#define PR_MODE  4
#define PT_MODE  8

#define  IO_PORT_ADDR   0x28000000        //Bank2 od asych mem, AMS2 effective
/*SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS*/
/*Program Section*/
.section program;

/* Init operating mode, stay in supervisor */
SetupOpMode:
    /* int 15 point to user program */
    p0.l     = EVT15 & 0xffff;
    p0.h     = (EVT15 >> 16) & 0xffff;
    r0       = MainProc(Z);
    r0.h     = MainProc;
    [p0]     = r0;

    /* Enable int 15 */
    p0.l     = IMASK & 0xffff;
    p0.h     = (IMASK >>16) & 0xffff;
    r0.l     = W[p0];
    r1.l     = EVT_IVG15;
    r0       = r0 | r1;
    W[p0]    = r0.l;

    /* Create int 15 */
    raise 15;

    p0.l = DeadLoop;
    p0.h = DeadLoop;
    reti = p0;
    rti;
    nop;
    nop;

DeadLoop:
    nop;
    jump DeadLoop;

MainProc:

    call InitRegs;

    /* Initialize Stack */
    SP       =UserStack (Z);
    SP.h     =UserStack;

    /* Enable interrupt,still in superviser mode */
```

```
    [--sp]   =reti;

    call SetupEvtHandler;

    call SetupCoreTimer;

    /* Setup EBIU */
    call SetupEBIUAsynchMem;

    /* Init Mode as TX, 0-RX, 1-TX, 2-PR, 3-PT */
    p0.l     = SysMode;
    p0.h     = SysMode;
    r0.l     = RX_MODE;
    W[p0]    = r0.l;

    /* Setup DMA */
    call InitCommonDMA;
    call SetupRXDMA;

    /* Setup PF pin, PF0,1,2,3 output */
    r0.l=0x000f;
    p0.l = FIO_DIR & 0xffff;
    p0.h = FIO_DIR >> 16;
    p1.l = FIO_FLAG_S & 0xffff;
    p1.h = FIO_FLAG_S >> 16;
    p2.l = FIO_FLAG_C & 0xffff;
    p2.h = FIO_FLAG_C >> 16;
    W[p0] = r0;

    /* Enable MemDMA sys int . Manual has something wrong, it is clear to
       enable interrupt*/
    i1.L = SIC_IMASK & 0xffff;
    i1.H = SIC_IMASK >> 16;
    r3   = [i1];
    bitclr(r3, 19);
    [i1] = r3;
    csync;

    /* Enable core timer interrupt & MemDMA write int */
    i1.L = IMASK & 0xffff;
    i1.H = IMASK >> 16;
    r3 =  0x2040 (Z);
    [ i1 ] = r3;
    csync;

    /* Start RX DMA operation */
    call StartRXDMA;

    /* Start core timer */
    i0.l = TCNTL & 0xffff;
    i0.h = TCNTL >> 16;
    r4 = 0x0007 (Z);

    r5=0x1;

    /* Testing DMA */
LoopA:
    p1.l = RXReadDMADp1;
```

```
    p1.h = RXReadDMADp1;

DMA_WAIT1:
    R6 = W[P1];
    cc = bittst(R6,15);
    IF cc JUMP DMA_WAIT1;
    nop;
    bitset(r6,15);
    W[p1] = R6;
    nop;
    p1.l = RXWriteDMADp1;
    p1.h = RXWriteDMADp1;
    R6 = W[P1];
    bitset(r6,15);
    W[p1] = R6;
    nop;

    p1.l = RXReadDMADp2;
    p1.h = RXReadDMADp2;

DMA_WAIT2:
    R6 = W[P1];
    cc = bittst(R6,15);
    IF cc JUMP DMA_WAIT2;
    nop;
    bitset(r6,15);
    W[p1] = R6;
    nop;
    nop;
    p1.l = RXWriteDMADp2;
    p1.h = RXWriteDMADp2;
    R6 = W[P1];
    bitset(r6,15);
    W[p1] = R6;
    nop;
    jump LoopA;

_EHANDLER:            // Emulation Handler 0
    RTE;

_RHANDLER:            // Reset Handler 1
    RTI;

_NHANDLER:            // NMI Handler 2
    RTN;

_XHANDLER:            // Exception Handler 3

off1:
    nop;
    nop;
    Jump off1;
    RTX;

_HWHANDLER:           // HW Error Handler 5
off2:
    nop;
    nop;
```

```
    Jump off2;
    RTI;

_THANDLER:            // Timer Handler 6

    [--sp]    = p1;
    [--sp]    = p2;
    [--sp]    = r5;

    p1.l = FIO_FLAG_S & 0xffff;
    p1.h = FIO_FLAG_S >> 16;

    p2.l = TimerToggle ;
    p2.h = TimerToggle ;

    r5   = W[p2](Z);

    bittgl(r5,0);
    W[p2] = r5.l;

    CC = r5 == 0x1;
    IF CC JUMP on;
    p1.l = FIO_FLAG_C & 0xffff;
    p1.h = FIO_FLAG_C >> 16;
on:
    r5   = 0x0001;
    W[p1]=r5;

    r5   = [sp ++];
    p2   = [sp ++];
    p1   = [sp ++];

    RTI;

_RTCHANDLER:     // IVG 7 Handler (RTC)
    RTI;

_I8HANDLER:      // IVG 8 Handler
    RTI;

_I9HANDLER:      // IVG 9 Handler
    RTI;

_I10HANDLER:     // IVG 10 Handler
    RTI;

_I11HANDLER:     // IVG 11 Handler
    RTI;

_I12HANDLER:     // IVG 12 Handler
    RTI;

_I13HANDLER:     // IVG 13 Handler
    [--sp]    = p0;
    [--sp]    = r0;
    [--sp]    = r1;

    /* Clear MemWDMA interrupt reg */
```

```
    p0.l = MDR_DI & 0xffff;
    p0.h = (MDR_DI >> 16) & 0xffff;
    r0.l = W[p0];
    W[p0]    = r0.l;

    /*=======*/
    p0.l = SysMode;
    p0.h = SysMode;
    r0.l = W[p0];
    cc=  bittst(r0, 1);
    if cc Jump TXProcessing;

RXProcessing:
    p0.l     = RXDMACount;
    p0.h     = RXDMACount;
    r0       = [p0];
    r0       += 1;
    [p0]     = r0;

    p0.l = FIO_FLAG_C & 0xffff;
    p0.h = FIO_FLAG_C >> 16;

    cc = bittst(r0,22);
    if cc jump   RXLedOff;

    p0.l = FIO_FLAG_S & 0xffff;
    p0.h = FIO_FLAG_S >> 16;

RXLedOff:
    r1       = 0x0002;
    W[p0]    = r1;

    cc = bittst(r0,0);
    if cc jump  RXOddFrame;

    p0.l     = RXReadDMADp2;
    p0.h     = RXReadDMADp2;
    r0       = W[p0];
    bitset(r0,15);
    W[p0]    = r0;

    p0.l     = RXWriteDMADp2;
    p0.h     = RXWriteDMADp2;
    r0       = W[p0];
    bitset(r0,15);
    W[p0]    = r0;

    jump     MemDMAISREnd;

RXOddFrame:
    p0.l     = RXReadDMADp1;
    p0.h     = RXReadDMADp1;
    r0       = W[p0];
    bitset(r0,15);
    W[p0]    = r0;

    p0.l     = RXWriteDMADp1;
    p0.h     = RXWriteDMADp1;
```

```
    r0        = W[p0];
    bitset(r0,15);
    W[p0]     = r0;

MemDMAISREnd:

    r1   = [sp++];
    r0   = [sp++];
    p0   = [sp++];
    rti;

TXProcessing:
    p0.l      = TXDMACount;
    p0.h      = TXDMACount;
    r0        = [p0];
    r0        += 1;
    [p0]      = r0;

    p0.l = FIO_FLAG_C & 0xffff;
    p0.h = FIO_FLAG_C >> 16;

    cc = bittst(r0,22);
    if cc jump   TXLedOff;

    p0.l = FIO_FLAG_S & 0xffff;
    p0.h = FIO_FLAG_S >> 16;

TXLedOff:
    r1        = 0x0004;
    W[p0]     = r1;

    cc = bittst(r0,0);
    if cc jump  TXOddFrame;

    p0.l      = TXReadDMADp2;
    p0.h      = TXReadDMADp2;
    r0        = W[p0];
    bitset(r0,15);
    W[p0]     = r0;

    p0.l      = TXWriteDMADp2;
    p0.h      = TXWriteDMADp2;
    r0        = W[p0];
    bitset(r0,15);
    W[p0]     = r0;

    jump      MemDMAISREnd;

TXOddFrame:
    p0.l      = TXReadDMADp1;
    p0.h      = TXReadDMADp1;
    r0        = W[p0];
    bitset(r0,15);
    W[p0]     = r0;

    p0.l      = TXWriteDMADp1;
    p0.h      = TXWriteDMADp1;
    r0        = W[p0];
```

```
    bitset(r0,15);
    W[p0]    = r0;

    Jump MemDMAISREnd;

_I14HANDLER:
    RTI;

_I15HANDLER:
    RTI;

////////////////////
/* Subroutine area */
////////////////////
/*---------------------------------------/
    Init all register
    Because the reset might not reset
    the register into a legal value
/---------------------------------------*/
InitRegs:
    /* Initialize all register file */
    R0 = 0; R1 = 0; R2 = 0; R3 = 0;
    R4 = 0; R5 = 0; R6 = 0; R7 = 0;
    P0 = 0; P1 = 0; P2 = 0; P3 = 0; P4 = 0; P5 = 0;
    I0 = 0 (X); I1 = 0 (X); I2 = 0 (X); I3 = 0 (X);
    M0 = 0 (X); M1 = 0 (X); M2 = 0 (X); M3 = 0 (X);
    L0 = 0 (X); L1 = 0 (X); L2 = 0 (X); L3 = 0 (X);
    B0 = 0 (X); B1 = 0 (X); B2 = 0 (X); B3 = 0 (X);
    rts;

/*---------------------------------------/
    Setup all the interrupt handler
/---------------------------------------*/
SetupEvtHandler:
    /*=================================*/
    /* Setup Event Vectors and Handlers */
    R0 = 0;
    p0.l = EVT0 & 0xffff;
    p0.h = EVT0 >> 16;

    p0   +=4;

    // Reset Handler (Int1)
    R0 = _RHANDLER (Z);
    R0.H = _RHANDLER;
    [ P0 ++ ] = R0;

    // NMI Handler (Int2)
    R0 = _NHANDLER (Z);
    R0.H = _NHANDLER;
    [ P0 ++ ] = R0;

    // Exception Handler (Int3)
    R0.L = _XHANDLER;
    R0.H = _XHANDLER;
    [ P0 ++ ] = R0;

    // IVT4 isn't used, Reserved
```

```
    [ P0 ++ ] = R0;

    // HW Error Handler (Int5)
    R0 = _HWHANDLER (Z);
    R0.H = _HWHANDLER;
    [ P0 ++ ] = R0;

    // Core Timer Handler (Int6)
    R0 = _THANDLER (Z);
    R0.H = _THANDLER;
    [ P0 ++ ] = R0;

    // IVG7 Handler, (RTC, USB, PCI)
    R0 = _RTCHANDLER (Z);
    R0.H = _RTCHANDLER;
    [ P0 ++ ] = R0;

    // IVG8 Handler (SPORT0,1-RX/TX )
    R0 = _I8HANDLER (Z);
    R0.H = _I8HANDLER;
    [ P0 ++ ] = R0;

    // IVG9 Handler (SPI0,1)
    R0 = _I9HANDLER (Z);
    R0.H = _I9HANDLER;
    [ P0 ++ ] = R0;

    // IVG10 Handler (UART0,1 --RX/TX )
    R0 = _I10HANDLER (Z);
    R0.H = _I10HANDLER;
    [ P0 ++ ] = R0;

    // IVG11 Handler (Timer 0,1,2)
    R0 = _I11HANDLER (Z);
    R0.H = _I11HANDLER;
    [ P0 ++ ] = R0;

    // IVG12 Handler (Programmable flag A/B)
    R0 = _I12HANDLER (Z);
    R0.H = _I12HANDLER;
    [ P0 ++ ] = R0;

    // IVG13 Handler (Memory DMA )
    R0 = _I13HANDLER (Z);
    R0.H = _I13HANDLER;
    [ P0 ++ ] = R0;

    // IVG14 Handler (Software Interrrupt 0)
    R0 = _I14HANDLER (Z);
    R0.H = _I14HANDLER;
    [ P0 ++ ] = R0;

    // IVG15 Handler (Software Interrupt 1)
    R0 = _I15HANDLER (Z);
    R0.H = _I15HANDLER;
    [ P0 ++ ] = R0;

    // ??????????
```

```
    P0 = EVT_OVERRIDE & 0xffff;
    P0.H = EVT_OVERRIDE >> 16;
    R0 = 0;
    [ P0 ] = R0;

    /*====================================*/
    rts;

/*-----------------------------------------/
    Setup all the interrupt handler
/-----------------------------------------*/
SetupCoreTimer:
    /* Setup core timer */
    i0.l = TCNTL & 0xffff;
    i0.h = TCNTL >> 16;
    m0=4;
    r0=0x5 (Z);
    [i0++m0] = r0;           //Ctimer Control reg
    csync;

    r1.l=0x0000;
    r1.h=0x0004;
    [i0++m0]=r1;       //Ctimer period
    csync;

    r2 = 0x0080 (Z);
    [i0++m0]=r2;       //Ctimer scale
    csync;

    [i0]=r1;           //Ctimer count
    csync;

    rts;


/*-----------------------------------------------*
 *  Name      :   SetupEBIUAsynchMem
 *  Function  :   Setup EBIU 's Aych memory for
                  ADC/DAC data
 *  Register  :
 *       In:      None
 *       Out:     None
 *       Used:
 *  Author        :     Jeff Sondermeyer
 *  Create Date   :     01/11/2002
 *  Modified  :
 *-----------------------------------------------*/
SetupEBIUAsynchMem:
    p0.l = EBIU_AMBCTL0 & 0xffff;
    p0.h = (EBIU_AMBCTL0 >> 16) & 0xffff;

    // Bank 0: Ardy =0, ArdyPol =0,  BTT =01(1), BST =01(1), BHT =10 (2),
    // BRAT =0001 (1), BWAT = 0100(4).
    r0.l = 0xffc7;     //0x4194;

    //Keep Default value for bank
    r0.h = 0xffc2;
    [p0] = r0;
```

```
    SSYNC;

    p0.l = EBIU_AMBCTL1 & 0xffff;
    p0.h = (EBIU_AMBCTL1 >> 16) & 0xffff;

    // Bank 2: Ardy =1, ArdyPol =1,  BTT =01(1), BST =00(3), BHT =11 (3),
    // BRAT =1111 (15), BWAT = 1111(15).
    r0.l = 0xffc7;

    //Keep Default value for bank
    r0.h = 0xffc2;
    [p0] = r0;
    SSYNC;

    p0.l = EBIU_AMGCTL & 0xffff;
    p0.h = (EBIU_AMGCTL >> 16) & 0xffff;

    //Clockout: Enabled, Bank0,1,2,3: enabled; 32 bits data path enabled
    r0 = 0x0007 (z);
    W[p0] = r0;
    SSYNC;

    RTS;

/*-----------------------------------------------*
 *  Name      :   InitCommonDMA
 *  Function  :   Initialize some common stuff for DMA.
                  For ex: DP BASE address, End DMA DP etc.
 *  Register  :
 *       In:      None
 *       Out:     None
 *       Used:
 *  Author        :     Jeff Sondermeyer
 *  Create Date   :     01/11/2002
 *  Modified  :
 *-----------------------------------------------*/
InitCommonDMA:

    //Initialize RXDMA count and TX DMA count
    p0.l      = RXDMACount;
    p0.h      = RXDMACount;
    r0        = 0;
    [p0++]    = r0;     //RX
    [p0]      = r0;     //TX

    //Init ending dp, all DMA DP point to this after transaction;
    p0.l = DMAEndDp;
    p0.h = DMAEndDp;
    r0.l = 0;
    W[p0]     = r0.l;

    //Init Descriptor base pointer register */
    r0.l      = RXReadDMADp1;
    r0.h      = RXReadDMADp1;
    P0.L      = DB_NDBP & 0xFFFF;
    P0.H      = (DB_NDBP >> 16) & 0xFFFF;
    W[p0]     = r0.h;
```

```
    RTS;

/*-----------------------------------------------*
 *  Name     :   SetupRXDMA
 *  Function :   Setup RX MemDMA for get data
 *               From ADC
 *  Register :
 *       In:     None
 *       Out:    None
 *       Used:
 *  Author:      Jeff Sondermeyer
 *  Create Date: 01/11/2002
 *  Modified:
 *-----------------------------------------------*/
SetupRXDMA:

    /* Source DMA */
    //Read Descripter1 Setup
    p0.l     = RXReadDMADp1;
    p0.h     = RXReadDMADp1;

    //Configuration, 32bit, int disable
    r0.l     = 0x8009;
    W[p0]    = r0.l;
    r0.l     = RX_DMA_LENGTH_SOURCE;
    W[p0+0x2]= r0;
    r0.l     = IO_PORT_ADDR  & 0xffff;
    r0.h     = (IO_PORT_ADDR >> 16) & 0xffff;
    [p0+0x4] = r0;
    r0.l     = RXReadDMADp2;
    W[p0+0x8]= r0;

    //Setup Next DP pointer
    r0       = p0;
    P0.L     = MDR_DND & 0xFFFF;
    P0.H     = (MDR_DND >> 16) & 0xFFFF;
    W[p0]    = r0.l;

    //Read Descripter2 Setup
    p0.l     = RXReadDMADp2;
    p0.h     = RXReadDMADp2;

    //Configuration, 32bit, int disable
    r0.l     = 0x8009;
    W[p0]    = r0.l;
    r0.l     = RX_DMA_LENGTH_SOURCE;
    W[p0+0x2]= r0;
    r0.l     = IO_PORT_ADDR & 0xffff;
    r0.h     = (IO_PORT_ADDR >> 16) & 0xffff;
    [p0+0x4] = r0;
    r0.l     = RXReadDMADp1;
    W[p0+0x8]= r0;

    /* Destination DMA */
    // Write descriptor1 Setup
    p0.l     = RXWriteDMADp1;
    p0.h     = RXWriteDMADp1;
```

```
    //Configuration, 32bit, int enabled on completion
    r0.l      = 0x800b;
    W[p0]     = r0.l;
    r0.l      = RX_DMA_LENGTH_DESTINATION;
    W[p0+0x2] = r0;
    r0.l      = RXDMABufferA;
    r0.h      = RXDMABufferA;
    [p0+0x4]  = r0;
    r0.l      = RXWriteDMADp2;
    W[p0+0x8] = r0;

    //Setup Next DP pointer
    r0        = p0;
    P0.L      = MDW_DND & 0xFFFF;
    P0.H      = (MDW_DND >> 16) & 0xFFFF;
    W[p0]     = r0.l;

    // Write descriptor2 Setup
    p0.l      = RXWriteDMADp2;
    p0.h      = RXWriteDMADp2;

    //Configuration, 32bit, int enabled on completion
    r0.l      = 0x800b;
    W[p0]     = r0.l;
    r0.l      = RX_DMA_LENGTH_DESTINATION;
    W[p0+0x2] = r0;
    r0.l      = RXDMABufferB;
    r0.h      = RXDMABufferB;
    [p0+0x4]  = r0;
    r0.l      = RXWriteDMADp1;
    W[p0+0x8] = r0;

    RTS;

/*-----------------------------------------------*
 *  Name      :   StartRXDMA
 *  Function  :   Start RX MemDMA for get data
                  From ADC
 *  Register  :
 *       In:      None
 *       Out:     None
 *       Used:
 *  Author        :     Jeff Sondermeyer
 *  Create Date   :     01/11/2002
 *  Modified  :
 *-----------------------------------------------*/

StartRXDMA:
    // Enable Destination
    P0.L      = MDW_DCFG & 0xFFFF;
    P0.H      = (MDW_DCFG >> 16) & 0xFFFF;
    R0        = W[P0];
    BITSET(R0,0);
    W[p0]     = R0.l;

    // Enable source
    P0.L      = MDR_DCFG & 0xFFFF;
    P0.H      = (MDR_DCFG >> 16) & 0xFFFF;
```

```
    R0       = W[P0];
    BITSET(R0,0);
    W[p0]    = r0.l;
    RTS;

/*-----------------------------------------------*
 *  Name     :   SetupTXDMA
 *  Function :   Setup TX MemDMA to send data
 *               to DAC
 *  Register :
 *       In:           None
 *       Out:     None
 *       Used:
 *  Author       :     Jeff Sondermeyer
 *  Create Date  :     01/17/2002
 *  Modified :
 *-----------------------------------------------*/
SetupTXDMA:

    /* Source DMA */
    //Read Descripter1 Setup
    p0.l     = TXReadDMADp1;
    p0.h     = TXReadDMADp1;

    //Configuration, 32bit, int disable
    r0.l     = 0x8009;
    W[p0]    = r0.l;
    r0.l     = TX_DMA_LENGTH_SOURCE;
    W[p0+0x2]= r0;
    r0.l     = TXDMABufferA;
    r0.h     = TXDMABufferA;
    [p0+0x4] = r0;
    r0.l     = TXReadDMADp2;
    W[p0+0x8]= r0;

    //Setup Next DP pointer
    r0       = p0;
    P0.L     = MDR_DND & 0xFFFF;
    P0.H     = (MDR_DND >> 16) & 0xFFFF;
    W[p0]    = r0.l;

    //Read Descripter2 Setup
    p0.l     = TXReadDMADp2;
    p0.h     = TXReadDMADp2;

    //Configuration, 32bit, int disable
    r0.l     = 0x8009;
    W[p0]    = r0.l;
    r0.l     = TX_DMA_LENGTH_SOURCE;
    W[p0+0x2]= r0;
    r0.l     = TXDMABufferB;
    r0.h     = TXDMABufferB;
    [p0+0x4] = r0;
    r0.l     = TXReadDMADp1;
    W[p0+0x8]= r0;

    /* Destination DMA */
    // Write descriptor1 Setup
```

```
    p0.l      = TXWriteDMADp1;
    p0.h      = TXWriteDMADp1;

    //Configuration, 32bit, int enabled on completion
    r0.l      = 0x800f;
    W[p0]     = r0.l;
    r0.l      = TX_DMA_LENGTH_DESTINATION;
    W[p0+0x2] = r0;
    r0.l      = IO_PORT_ADDR  & 0xffff;
    r0.h      = (IO_PORT_ADDR >> 16) & 0xffff;
    [p0+0x4]  = r0;
    r0.l      = TXWriteDMADp2;
    W[p0+0x8] = r0;

    //Setup Next DP pointer
    r0        = p0;
    P0.L      = MDW_DND & 0xFFFF;
    P0.H      = (MDW_DND >> 16) & 0xFFFF;
    W[p0]     = r0.l;

    // Write descriptor2 Setup
    p0.l      = TXWriteDMADp2;
    p0.h      = TXWriteDMADp2;

    //Configuration, 32bit, int enabled on completion
    r0.l      = 0x800f;
    W[p0]     = r0.l;
    r0.l      = TX_DMA_LENGTH_DESTINATION;
    W[p0+0x2] = r0;
    r0.l      = IO_PORT_ADDR & 0xffff;
    r0.h      = (IO_PORT_ADDR >> 16) & 0xffff;
    [p0+0x4]  = r0;
    r0.l      = TXWriteDMADp1;
    W[p0+0x8] = r0;

    RTS;

/*-----------------------------------------------*
 *  Name      :   StartTXDMA
 *  Function  :   Start TX MemDMA to send data to DA
 *  Register  :
 *       In:      None
 *       Out:     None
 *       Used:
 *  Author:       Jeff Sondermeyer
 *  Create Date: 01/11/2002
 *  Modified:
 *-----------------------------------------------*/

StartTXDMA:
    // Enable Dest
    P0.L      = MDW_DCFG & 0xFFFF;
    P0.H      = (MDW_DCFG >> 16) & 0xFFFF;
    R0        = W[P0];
    BITSET(R0,0);
    W[p0]     = R0.l;

    // Enable source
```

```
    P0.L      = MDR_DCFG & 0xFFFF;
    P0.H      = (MDR_DCFG >> 16) & 0xFFFF;
    R0        = W[P0];
    BITSET(R0,0);
    W[p0]     = r0.l;
    RTS;

/*SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS*/
/*Start code Section*/
.section start_from_here;
    r0        = SetupOpMode(Z);
    r0.h      = SetupOpMode;
    p0        = r0;
    csync;
    jump  (p0);
    nop;
    nop;

/*SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS*/
/*Stack  Section*/
.section ustack;
.align 4;
.var UserStackSpace0[1000];
UserStack:
.var UserStackSpace1[4];

/*SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS*/
/*Data Section*/

// L2 bank 2 */
.section L2_B2_data;
/* MemDMA descriptor buffer */
//First RX
.align 4;
.BYTE2   RXReadDMADp1[5];
.align 4;
.BYTE2   RXWriteDMADp1[5];

//First TX
.align 4;
.BYTE2   TXReadDMADp1[5];
.align 4;
.BYTE2   TXWriteDMADp1[5];

//END
.align 4;
.BYTE2   DMAEndDp[1];
.BYTE2   TimerToggle[1];
.align 4;
.BYTE4   RXDMACount[1];
.BYTE4   TXDMACount[1];
.align 4;
.BYTE2   SysMode[1];

// L2 Bank 3 */
.section L2_B3_data;
/* MemDMA descriptor buffer */
//Second RX
```

```
.align 4;
.BYTE2   RXReadDMADp2[5];
.align 4;
.BYTE2   RXWriteDMADp2[5];

//Second TX
.align 4;
.BYTE2   TXReadDMADp2[5];
.align 4;
.BYTE2   TXWriteDMADp2[5];

// L1 B bank, sub bank 0 */
.section L1_B0_data;
.align 4;
TXDMABufferA:
.BYTE4   RXDMABufferA[1024];

// L1 B bank, sub bank 1 */
.section L1_B1_data;
.align 4;
TXDMABufferB:
.BYTE4   RXDMABufferB
```

Document History

Version	Description
June 05, 2003	Updated to new Blackfin naming convention and reformatting. Ported example code from VisualDSP++ 2.0 to VisualDSP++ 3.1
February 12, 2002	Initial Release