Accelerating DSP Filter Loops with MIPS® CorExtend® Instructions

Document Number: MD00303
Revision 01.01
June 30, 2008

Table of Contents

List of Figures

List of Tables

Introduction and Background

1.1 Introduction

Consumer digital appliances are becoming increasingly sophisticated. There is a continual demand for increased performance and flexibility at lower costs. To meet these conflicting demands, embedded processors that traditionally only executed systems controller code are now executing computation-intensive application code as well. Executing application code on the main processor rather than on fixed function application logic has many advantages:

- By eliminating some fixed function blocks from the chip, the total cost is reduced. This is possible since the SOC (System On a Chip) die area is usually smaller.
- Cost is also reduced since the verification effort is reduced. Reducing the verification effort has the added benefit of reducing the time to market, which is a critical issue with consumer products.
- The programmability of the main processor permits flexibility in application programming. This is important when the underlying protocols or algorithms change.
- Additionally, programming for a standard architecture such as MIPS32® [1] reduces the time to market of the product due to the availability of standard tools chains.

To extract the best possible performance from specific applications, MIPS Technologies' Pro Series® cores offer the CorExtend® feature. CorExtend allows the addition of new User Defined Instructions (UDI) and new state to a standard MIPS32® architecture. The new instructions enable a MIPS32 processor to meet the performance demands of DSP-type applications. This allows a cost reduction as discussed. There is also a power advantage. For instance, the performance boost can be used to reduce power by lowering the clock frequency of the core. Power can also be reduced because the new instructions typically result in fewer dynamically executed instructions. This translates to a reduction in instruction bandwidth and hence the power consumed by that loop.

This application note uses three filter algorithms as examples to illustrate the performance increase using CorExtend. These are FIR, LMS, and Complex FIR Filters, which are standard DSP functions used in a variety of embedded applications including telecommunications/speech processing, audio/video encoding and decoding, and image processing. Table 1-1 shows some application areas for these filters.

Table 1-1 DSP Filters and Some Application Areas

Filter Name	Application Examples
FIR (Finite Impulse Response) Filter	Speech processing, general dsp functions like ECG, image processing
LMS (Least Mean Squared) Adaptive Filter	Echo Cancellation, noise cancellation, adaptive equalization
Complex FIR Filter	DSL Soft-modems

The rest of this chapter describes each filter briefly and states some of the assumptions used in their coding. For a more detailed description of these filters and their applications, we recommend a standard textbook on digital signal processing [2][3]. The last section of this chapter briefly describes the CorExtend feature [4].

1.2 Finite Impulse Response (FIR) Filter

A FIR filter with input *x(n)* and output *y(n)* is represented by the convolution sum:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) = h(0)x(n) + h(1)x(n-1) + \ldots + h(N-2)x(n-(N-2)) + h(N-1)x(n-(N-1))$$

The mathematical notation of summing two products with one of them retreating and the other advancing is called a convolution. The above convolution represents the frequency response of the current signal obtained by combining the current input with the attenuated (or amplified) echoes from past inputs. The attenuation or amplification is given by the filter coefficients *h(k).* The previous inputs are specified by x(n - k) for k = 1 to k = N - 1. These are also referred to as state variables. The filter length is N (since the filter is finite), and is the length for which the past echoes are considered to have some impact on the current output.

FIR filters have many advantages and they are simple to implement. Thus, FIR filters are widely used for their ease of design and flexibility.

The design implemented has the following parameters and assumptions:

- Sample-based - meaning it computes one output sample at a time.
- 16 bits wide integer input values.
- Filter length of 16 taps.
- Coefficients are assumed to be stored in memory ahead of time.

1.3 Least Mean Squared (LMS) Filter

LMS filters are a type of adaptive filter. It is essentially an FIR filter that adjusts its coefficients from sample to sample, such that the error between the desired and actual outputs is minimized. It is used in applications such as echo cancellation where the filter coefficients are adapted to try to accurately reproduce the echo. The algorithm is initialized by setting all coefficients to zero at time n = 0. These tap weights are improved gradually with time and updated using the relationship:

$$w_i(n+1) = w_i(n) + \mu \cdot e(n) \cdot x(n)$$

where $w_i(n)$ are the filter coefficients of the FIR filter, $e(n)$ is the error signal, $x(n)$ represent the state variables, and the factor μ is the adaptation parameter or step-size.

The LMS algorithm is said to converge when the mean squared error value converges to some small value. To prevent errors in the filter calculation where the convergence is to a non-optimal solution, it is sufficient to assume 16 bit wide coefficients, and use 16x16 multipliers for the computation.

The design implemented has the following parameters and assumptions:

- Sample-based.
- 16 bits wide fixed-point input values.
- Filter length of 16 taps.

1.4 Complex FIR Filter

The complex FIR filter is an FIR filter where the coefficients and states are complex values, and is defined as follows:

$$y(n) = REAL\{y(n)\} + IMAG\{y(n)\}i \quad \text{where}$$

$$REAL\{y(n)\} = \sum_{k=0}^{N-1} REAL\{h(k)x(n-k)\}$$

$$IMAG\{y(n)\} = \sum_{k=0}^{N-1} IMAG\{h(k)x(n-k)\}$$

The filter inputs are complex values, and the filter output is complex as well, where the real coefficient is the summation of all the real components and the imaginary coefficient is the summation of all the imaginary components of the complex multiply operation.

The use of complex variables simplifies many DSP calculations and hence these filters are used in many widely differing areas.

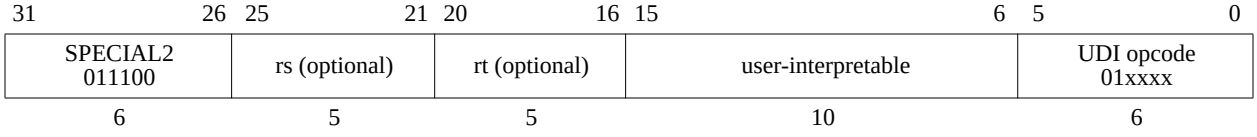
The design implemented has the following parameters and assumptions:

- Frame-based - multiple output samples are computed per call.
- An input consists of a real and imaginary part; each part is 16 bits wide placed pair-wise in memory.
- Filter length of 16 taps.

1.5 Description of CorExtend

The MIPS32 architecture reserves 16 opcodes under the SPECIAL2 main opcode for the use of User Defined Instructions. This is shown in Figure 1-2, where bits 0 through 3 are available to encode the user's instruction opcode. Note that this instruction format has 20 other available bits for the use of the instruction. Hence, more than 16 UDIs could be encoded using other user-interpretable bits of the instruction word.

Figure 1-2 Basic UDI Instruction Format



Here are some brief highlights on the flexibility and restrictions imposed by the main core on an UDI block:

- Only fixed integer instructions are allowed. No jumps, branches, loads, or stores are allowed.
- When the main core pipeline decodes to an UDI, two source operands are read using bits *rs* and *rt* as register numbers. These register contents are available on the interface to the UDI block. The UDI block can choose to ignore these if it uses only one or none of the sources for its instruction.
- The destination register value can be derived from any of the bits of the instruction, or independently chosen by the UDI block. The destination could be a GPR or an UDI block internal register.
- The UDI can have a single cycle or multiple cycle latency. All single cycle instructions and multiple cycle instructions that don't write to a GPR will not stall the core's pipeline. Multi-cycle instructions that write to a GPR will stall the pipeline.

Code Analysis Used to Define New Instructions

2.1 Introduction to New Instruction Definition and Code Analysi

We will describe in this section the basic rationale behind the new instructions, describing how they enhance application performance. Original code samples are shown in this chapter. Code samples rewritten with the new instructions are also shown. The full list of all the user defined instructions recommended for DSP filters is listed in the next chapter (see Table 3-4 on page 14).

There are three fundamental factors that dictated the design of the new instructions. They are described here:

1. When processing digital signals, arithmetic operations that overflow and underflow values by wrapping around are often not useful. Hence, much of the computation in digital speech processing involves saturation, i.e., clamping values to the largest positive or negative representable value on overflow/underflow. This can be handled in two ways. Wide accumulators with 16 guard bits can be used with overflow and clamping done once, at the end. This is a cleaner implementation and avoids frequent saturation operations. But it is expensive in hardware since wider accumulators have an area cost. Another approach is to do the overflow check and saturation after every operation. Sometimes the latter approach is mandated by a standard, for example, some of the ITU's VoIP standards like G.723.1, G.729, etc. To implement this type of operation-level saturation using general microprocessor instructions adds many cycles to the basic arithmetic operation and is a considerable overhead. Hence, the most significant improvement is obtained by defining arithmetic operation instructions that automatically check for overflow and underflow and saturate the results.
2. DSP filters are often applied to speech or voice data. When these can be represented effectively using 16 bits, two such values can be simultaneously processed in the 32 bits of a MIPS32 register. That is, the instruction is used in a SIMD (Single Instruction Multiple Data) mode of operation. Instructions with this type of operation parallelism were very effective in speeding up the filter loops.
3. The third factor that improved application performance was the ability to define new accumulator registers and new instructions that write to the new accumulators. This eliminates the bottleneck of the single hi/lo register pair of the main architecture. These new accumulators were also useful when multi-cycle pipelined instructions wrote to the internal accumulators, and hence did not stall the main integer pipeline.

The above list provides some background and will help the reader understand the reasoning behind many of the new instructions that will be used in this chapter in the code samples. A more detailed analysis of DSP-type functions is provided elsewhere [5].

In this chapter, we analyze the three filter loops. In the FIR filter example, we introduce the Multiply-Accumulate (MPA_PH) and Circular buffer pointer Add (CIRADD) instructions. In the LMS filter example, we introduce the special three-input argument Multiply-Add with Rounding (MADDQR_PH) instruction. In the final complex FIR filter example, we introduce the complementary Complex Dot-Product Accumulate instructions with the Real components (DPAR_CH) and the Imaginary components (DPAI_CH).

2.2 Analysis of the FIR Filter

Figure 2-1 shows an example C code used to compute a single sample of an FIR filter. There are two critical loops in this algorithm. The first is the integer multiply and accumulation of the terms in the convolution. The other is the shuffling of values in memory at the end, so that the very last (delayed) input state value is dropped and a new one is picked up (for the next pass through the algorithm). Since this implementation assumes that the coefficients are stored in reverse order, the convolution loop indexes both variables the same way. If the coefficients could not be stored in reverse order, then it would be a simple change in the code to index the coeff[] values in reverse order.

We have chosen not to handle overflow/underflow in this loop since we have made an assumption that all the values have been prescaled to avoid overflow in the chosen data size. But if saturation is necessary, then the appropriate saturating instructions can be defined.

Figure 2-1 FIR Filter C code

```
/* NOTE: Coefficients are stored in reverse order */
/* Update state buffer with new input sample */
state[numtaps-1] = in;
sum=0;
/* Perform the convolution sum */
   for (i = 0; i < numtaps; i++) {
     sum += state[i] * coeff[i];
}

/* Shift old input values down. */
for (i = 1; i < numtaps; i++) {
   state[i-1] = state[i];
}
```

Figure 2-2 shows an optimized MIPS32 assembly version where the two loops are combined and unrolled 4 times. The number of taps, "numtaps" is assumed to be a multiple of 4, hence unrolling the loop by 4 is a valid assumption in this example. Register $t0 is pre-loaded with the value of numtaps.

Note that the input state values and the coefficients can be represented by 16 bits. Hence the code uses load half word (lh) to get the state and coefficient values into registers. Since the loop is unrolled 4 times, the loads in lines 1, 5, 9, and 13 are loading one half word (two bytes) of the state value each time, from offset locations 0, 2, 4, and 6 based off register $a2. Similarly, the coefficient values are loaded in lines 2, 6, 10, and 14, based off register $a3.

The store half (sh) instruction in lines 3, 7, 11, and 15 represents the second loop in Figure 2-1, where the state values are shifted down. Hence the store is based off the same register $a2, but the offset is two less than the previous load of the state value (-2, 0, 2, and 4).

Once the values have been loaded into registers, the multiply-accumulate instruction (madd) in lines 4, 8, 12, and 16 accumulates the sum into the HI/LO registers.

The instruction on line 20, mflo (move from register LO) moves the final accumulated sum out of the LO register into a GPR $v0.

As we examine this loop the most striking aspect is the 16 bit data. Hence, the most obvious improvement can come from parallelizing the multiply-accumulate calculation so that two of them are done at a time out of a 32 bit register. This is the Paired-Half (PH) format, illustrated in Figure 3-1. This type of dual (or more) parallelism is generally known as register SIMD (Single Instruction Multiple Data).

The parallel multiply-accumulate operation is defined such that the result of the two multiplications is accumulated into a special accumulator register. Hence, the computation is a multiply with accumulate (MPA). Using a special accumulator register has the added advantage that the integer pipe of the processor is not stalled while the MPA instruction is executing, despite a 3 cycle instruction latency (our current assumption). This allows the integer pipe to not stall and continue to issue other instructions. If the MPA instruction is pipelined internally, these MPA instructions can be issued in sequence as well. In our implementation, we assume a single multiplier that is double-pumped to produce the two results, hence the issue rate of the MPA instructions is every other cycle (see Table 3-9 on page 16).

The accumulators are assumed to be 40 bits wide since that is sufficient for our example. Without much loss of area or generality they can be assumed to be 64 bits, if it is deemed necessary.

Figure 2-2 FIR Filter MIPS32 code

```
0   mtlo $0                 /* zero the accumulator */
dotloop:
  /* unrolled by 4 */
1   lh $a0, 0($a2)          /* load 1 state */
2   lh $t1, 0($a3)          /* load 1 coeff */
3   sh $a0, -2($a2)         /* shift input array down by one */
4   madd $1,$a0             /* multiply and accumulate */
5   lh $a0, 2($a2)          /* ... repeat 3 more times */
6   lh $t1, 2($a3)
7   sh $a0, 0($a2)
8   madd $t1,$a0
9   lh $a0,4($a2)
10  lh $t1, 4($a3)
11  sh $a0,2($a2)
12  madd $t1,$a0
13  lh $a0, 6($a2)
14  lh $t1, 6($a3)
15  sh $a0, 4($a2)
16  madd $t1,$a0
17  addiu $a3,#a4,4*2        /* increment coeff. ptr */
18  bne $a3, $t0, dotloop
19  addiu $a2,$a2,4*2        /* increment sample ptr in delay line */
20  mflo $v0                 /* get the sum */
```

When the sum is finally available, it needs to be copied out to a GPR from the internal accumulator register. To do this we define another instruction, move from UDI register (MFU). To accommodate many filter algorithms that require a specific scaling factor in their outputs, the MFU instruction allows for an optional right shift of the 40-bit accumulator value before the move to a 32-bit GPR. Corresponding to this MFU instruction, there is a need for an MTU (move to UDI register) instruction. This is used to zero out the accumulator before the convolution loop. And could also be used to restore register values if they are saved for a context switch.

A second optimization of the filter loop comes from eliminating the store instructions that are needed to implement the second loop (that shifts down the delay line). These shifts comprise nearly 25% of the instructions in the loop, and hence their elimination makes a significant contribution to the performance improvement. If we assume that the delay line state values are in a circular buffer in memory, then the shifting down simply involves moving a pointer value to indicate where to store the new delay value, each time the filter subroutine is called.

Implementing a circular buffer requires a fast way of wrapping the pointer to the beginning of the buffer when it reaches the end. In the Filter UDI, we have created a new instruction that does modulo arithmetic on a pointer and returns an updated pointer. The instruction takes the current pointer value and an immediate value which encodes the loop size and the increment value. The buffer must be properly aligned so that modulo arithmetic can be done quickly with only the lower bits of the pointer. The number of lower bits to mask is also encoded in the immediate value. The new instruction is called CIRADD. A MIPS32 implementation of a circular buffer would have required an extra comparison and conditional branch to wrap the pointer.

Figure 2-3 shows the FIR loop using MIPS32 with the new UDI. Note that in this implementation the loop is unrolled eight times, and the SIMD instructions process two values at a time. The circular buffer pointer is only used to determine where to write the new delay value. The convolution sum is always done on the delay values starting at the beginning of the delay buffer to the end, as defined in memory.

Figure 2-3 FIR Filter Code Using MIPS32 + UDI Instructions

```
             MTU $0, FiUDI_Acc0
             lw $a1, DLYsaveAddr($0)   /* ptr to cir buff loc of delay line */
             sh $a0, 0($a1)            /* store input into delay line */
             li $t8, encode(mask=0x3,loopsize=numtaps,increment=2)
             CIRADD $a2,$a2,$t8        /* incr ptr into delay line cir buff */
             sw $a2, DLYsaveAddr($0)   /* store the ptr for use next call */

           dotloop:
             /* unrolled by 8: 4 repetitions of 2-way SIMD */
             lw $a0,0($a2)             /* load 2 samples */
             lw $t1,0($a3)             /* load 2 coefficients */
             lw $a1,4($a2)             /* load 2 more samples */
             lw $t2,4($a3)             /* load 2 more coeffs */
             MPA_PH FiUDI_Acc0,$a0,$t1 /* do multiply/accumulate */
             lw $a0,8($a2)             /*  ... repeat */
             MPA_PH FiUDI_Acc0,$a1,$t2 /* another mult/acc */
             lw $t1,8($a3)
             lw $a1,12($a2)
             lw $t2,12($a3)
             MPA_PH FiUDI_Acc0,$a0,$t1
             add $a3,$a4,4*2           /* increment coeff ptr */
             bne $a3,$t0,dotloop       /* branch */
             MPA_PH FiUDI_Acc0,$a1,$t2

             MFU $v0,FiUDI_Acc0,SCALE  /* get final sum */
```

2.3 Analysis of the LMS Filter

The standard sample-based LMS filter consists of three main loops:

1. A first loop that runs a FIR filter based on the current coefficients.
2. A second loop that computes a new set of coefficients based on the error.
3. And a third loop that shuffles the states preparing for next call.

Since we have already studied and optimized the computation of the first and third loops, in this section we will only focus on the optimization of the second loop which updates the coefficients. Figure 2-4 shows the generic C code that computes one set of adapted coefficients of an LMS filter. Note that after the multiplication to adapt the coefficient to a new value, this value is rounded and truncated to obtain the new 16 bit coefficient value.

Figure 2-4 LMS Filter C code

```
/* Update coefficients */
for (j = 0; j < numtaps; j++) {
       int rounded = (mu * error * state[j]) + (coefs[j] << 15) + 0x4000;
       coefs[j] = rounded >> 15 & 0xffff;
}
```

This coefficient update is somewhat computationally expensive since it uses fixed-point arithmetic. Note that although the calculation of the "rounded" variable has two multiplications, it is really only one since ’mu’ (the rate of adaptation)

and ’error’ are constant and should be pre-computed before entering the loop. Figure 2-5 shows the optimized MIPS32 assembly of the loop in Figure 2-4.

Figure 2-5 LMS Filter MIPS32 code

```
/* Update coefficients, loop unrolled 2 times */
_coefloop:
1     lh $t0, 0($a0)            /* $t0 = state[j] */
2     lh $t1, 0($a2)            /* $t1 = coefs[j] */
3     mul $t0, $t0, $t4         /* $t0 = state[j]*(mu*error) */
4     lh $t2, 2($a0)            /* $t2 = state[j+1] */
5     lh $t3, 2($a2)            /* $t3 = coefs[j+1] */
6     mul $t2, $t2, $t4         /* $t2 = state[j+1] * (mu*error)*/
7     sll $t1, $t1, 15          /* $t1 = coefs[j] << 15 */
8     add $t1, $t0, $t1         /* $t1 = state[j]*(mu*error)+(coefs[j]<<15)*/
9     addiu $t1, $t1,0x4000     /* $t1 = state[j]*(mu*error)+(coefs[j]<<15)+0x4000*/
10    sra $t1, $t1, 15          /* $t1 = rounded[j] >> 15 */
11    sh $t1, 0($a2)
12    sll $t3, $t3, 15          /* $t3 = coefs[j+1] << 15 */
13    add $t3, $t2, $t1         /* $t3 = state[j+1]*(mu*error)+(coefs[j+1]<<15) */
14    addiu $t3,$t3,0x4000      /*$t1=state[j+1]*(mu*error)+(coefs[j+1]<<15)+0x4000*/
15    sra $t3, $t3, 15          /* $t1 = rounded[j+1] >> 15 */
16    sh $t3, 2($a2)
17    addiu $a0, $a0, 2*2       /* next iteration incr ptr to state[j+2] */
18    bne $a0, $t9, _coefloop
19    addiu $a2, $a2, 2*2       /* next iteration incr ptr to coefs[j+2] */
```

Looking at the MIPS32 assembly, it becomes apparent that a large number of instructions in the loop are spent on performing the fixed-point rounding and truncation after the multiply operation. The UDI instruction, MADDQR, (Fixed-point Multiply and Add with Rounding) addresses the overhead of this operation. The MADDQR is a 2-way SIMD instruction that performs the computation sequence of the instructions in lines 3, 7, 8, 9, and 10 in Figure 2-5. The MADDQR instruction uses the 16 bit fractional (or Q) data format (see Figure 3-3 on page 13).

The loop with the new instruction MADDQR is shown in Figure 2-6.

Key observations from this last implementation using UDIs:

- An internal UDI register, the Multiply Constant (MC) register is used to store the multiplication constant. The MTU (move to user defined register) instruction at the top of Figure 2-6 moves the pre-calculated value to the MC register before entering the loop.

- Like MPA, the MADDQR instruction is 2-way SIMD. To have sufficient instructions for scheduling purposes, the loop with the UDIs is unrolled four times.

Figure 2-6 LMS Filter MIPS32 + UDI code

```
/* Update coefficients, loop unrolled 4 times */
MTU ($t4, FiUDI_MC )               /* Set UDI mult const, $t4 = mu*error */
_coefloop:
     lw $t0, 0($a0)                /* Loads 2 states */
     lw $t1, 0($a2)                /* Loads 2 coefficients */
     lw $t2, 4($a0)
     lw $t3, 4($a2)
     MADDQR_PH ($t1, $t0, $t1 )    /* $t1 = UDI constant * $t0 + $t1 */
     addiu $a0, $a0, 4*2
     MADDQR_PH ($t3, $t2, $t3 )    /* $t3 = UDI constant * $t2 * $t3 */
     sw $t1, 0($a2)
     sw $t3, 4($a2)
     bne $a0, $t9, _coefloop
     addiu $a2, $a2, 4*2
```

2.4 Analysis of the Complex FIR Filter

Algorithmically, the frame-based complex FIR filter is very similar to the sample-based real only FIR filter. The key difference is the need to compute the convolution sum separately for the real and imaginary parts. Figure 2-8 shows the C code loop that performs the complex convolution on a frame of inputs. It is assumed that the complex number is stored in memory as a (real, imaginary) pair. Hence the outer loop computes on one complex number in each iteration, using two values each time from the inbuf[] array.

Figure 2-7 Complex FIR Filter C code

```
     /* Perform complex convolution on frame size of Nout */
     for ( k = 0, n = 0; n < numout*2; k+=2, n+=2 ) {
          int  realsum = 0;
          int  imagsum = 0;
          for ( i = 0; i < numtaps*2; i+=2 ) {
               #define a inbuf[k+i]
               #define b inbuf[k+i+1]
               #define c coef[i]
               #define d coef[i+1]
               realsum += a*c - b*d;
               imagsum += a*d + b*c;
          }
          outbuf[n] = (realsum >> 15);
          outbuf[n+1] = (imagsum >> 15);
     }
```

Figure 2-8 shows the optimized MIPS32 version of the loop. This computes one output sample. Note that start-up code that clears the hi/lo register pair is required. This would be the first iteration of the loop, which is not shown in Figure 2-8.

This implementation would benefit from having four accumulators since this eliminates the need to swap out the intermediate sum. An alternative would compute the imaginary part first and then rerun the loop to compute the real part. While such an alternative eliminates the swapping, it adds the overhead of reloading the coefficients.

Figure 2-8 Complex FIR Filter MIPS32 code

```
            addiu $t8, $a1, 0                /* Copy of COEF ptr  */
            addiu $t9, $a0, -(Ntaps-1)*2*2   /* Initialize k+i ptr.  */
            addiu $t5, $a0, -(Ntaps-1)*2*2   /* Reset SCRATCH ptr. */
            addiu $t7, $a2, (Nout*2)*2       /* $t7 = End of outer loop */
            addiu $t6, $a1, (Ntaps*2)*2      /* $t6 = End of inner loop. */
        _outer
             addiu $v0, $0, 0
             addiu $v1, $0, 0
        _inner:
            lh $t0, 0($t8)           /* load real coefficient */
            lh $t1, 2($t8)           /* load imag coefficient */
            lh $t2, 0($t9)           /* load 1 real state */
            lh $t3, 2($t9)           /* load 1 imag state */
            mult $t2, $t1
            madd $t3, $t0            /* perform imag dot product */
            addiu $t8, $t8, 2*2      /* Increment coef addr*/
            mflo $t4                 /* Move imagsum out */
            addiu $t9, $t9, 2*2      /* Increment inbuf addr */
            mult $t2, $t0
            msub $t3, $t1            /* perform real dot product */
            add $v1, $v1, $t4
            mflo $t4                 /* Move realsum out */
            bne $t8, $t6, _inner     /* End of inner loop */
            add $v0, $v0, $t4

            sra $v1, $v1, SCALE      /* Scale imagsum */
            sra $v0, $v0, SCALE      /* Scale realsum */
            sh $v0, 0($a2)           /* Store sum in outbuf */
            sh $v1, 2($a2)

            addiu $t5, $t5, 2*2
            addiu $t9, $t5, 0        /* Reset & increment k+i */
            addiu $a2, $a2, 2*2      /* Increment n by 2 */
            bne $a2, $t7, _outer     /* End of outer loop */
            addiu $t8, $a1, 0        /* Reset COEF addr i */
```

Three new instructions are introduced: DPAR and DPAI compute the dot product and accumulates the real part and the imaginary part of the sum respectively, and MFAS allows two accumulator values to be combined and saturated before copying to a single GPR. Note that the MFAS instruction is strictly not needed, it is merely a convenience. It can be replaced by a small number of MIPS32 instructions outside the inner loop.

Figure 2-9 shows the implementation of the loop with the proposed instructions. With the new instructions, the loop produces two rather than one complex output sample per execution.

Having 4 accumulators has allowed the unrolling of the outer loop. That is, two values of ’k’ are being computed simultaneously, thus producing two sample outputs at a time. This method is preferred over the option of unrolling the inner loop which would require only two accumulators and reduces the number of iterations required by half for computing a single output. By unrolling the outer loop, one load of the coefficient is eliminated for every inner loop iteration. Unrolling the outer-loop rather than the inner-loop in fact provided a 15% performance increase.

Figure 2-9 Complex FIR Filter MIPS32 + UDI code

```
            addiu $t8, $a1, 0                   /* Copy of COEF ptr  */
            addiu $t9, $a0, -(Ntaps-1)*2*2      /* Initialize k+i ptr.  */
            addiu $t5, $a0, -(Ntaps-1)*2*2      /* Reset SCRATCH ptr. */
            addiu $t7, $a2, (Nout*2)*2          /* $t7 = End of outer loop */
            addiu $t6, $a1, (Ntaps*2)*2         /* $t6 = End of inner loop. */
        _outer:
            MTU ($0,FiltUDI_Acc0)               /* zero out accumulator 0 */
            MTU ($0,FiltUDI_Acc1)               /* zero out accumulator 1 */
            MTU ($0,FiltUDI_Acc2)               /* zero out accumulator 2 */
            MTU ($0,FiltUDI_Acc3)               /* zero out accumulator 3 */
        _inner:
            lw $t0, 0($t9)                      /* k = 0 */
            lw $t2, 0($t8)                      /* load 1st coefficient */
            lw $t1, 4($t9)                      /* k = 1 */
            DPAR_CH (FiltUDI_Acc3,$t0,$t2)
            lw $t3, 4($t8)                      /* load 2nd coefficient */
            DPAI_CH (FiltUDI_Acc2,$t0,$t2)
            lw $t0, 8($t9)                      /* k = 2 */
            DPAR_CH (FiltUDI_Acc1,$t1,$t3)
            lw $t2, 8($t8)                      /* load 3rd coefficient */
            DPAI_CH (FiltUDI_Acc0,$t1,$t3)
            lw $t1, 12($t9)                      /* k = 3 */
            DPAR_CH (FiltUDI_Acc3,$t0,$t2)
            lw $t3, 12($t8)                      /* load 4th coefficient */
            DPAI_CH (FiltUDI_Acc2,$t0,$t2)
            add $t8, $t8, 4*2                   /* Increment COEFaddr*/
            DPAR_CH (FiltUDI_Acc1,$t1,$t3)
            add $t9, $t9, 4*2                   /* Increment INaddr */
            bne $t8, $t6, _inner                /* End of inner loop */
            DPAI_CH (FiltUDI_Acc0,$t1,$t3)

            MFAS_PH_2D ($t0,0,SCALE)
            MFAS_PH_2D ($t1,2,SCALE)
            sw $t1, 0($a2)                      /* Store 1st sum in OUTaddr */
            sw $t0, 4($a2)                      /* Store 2nd sum in OUTaddr */

            addiu $t5, $t5, 2*2
            addiu $t9, $t5, 0                   /* Reset & increment k+i */
            addiu $a2, $a2, 2*2                 /* Increment n by 2 */
            bne $a2, $t7, _outer                /* End of outer loop */
            addiu $t8, $a1, 0                   /* Reset COEF addr i */
```

Chapter 3

A Description of the CorExtend® Block

3.1 Data Formats and Data Types

The new Filter instructions introduce two new data formats in the GPR register set, the *paired half* (*PH*) and *complex half* (*CH*).

- In *pair half* format, a 32-bit GPR is interpreted as a vector of two (“paired”) signed 16-bit integers.
- In *complex half* format, a 32-bit GPR is interpreted as a complex vector with a 16-bit integer representing the real part and another 16-bit integer representing the imaginary part.

Figure 3-1 shows a GPR in PH data format; Figure 3-2 shows a GPR in CH data format. Positions within a vector are denoted as v[i], where i=0 for the least significant position in the vector register.

Figure 3-1 GPR in PH Data Format with Integer Data Type

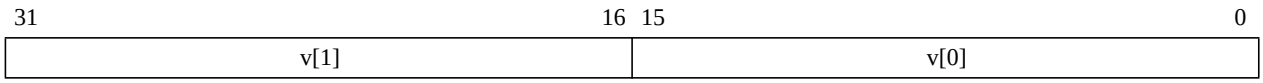
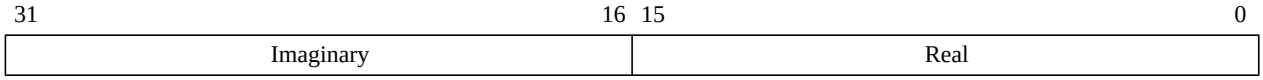
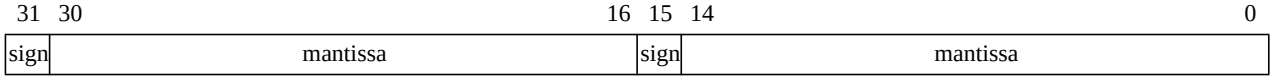


Figure 3-2 GPR in CH Data Format



Two data types are used for the PH data format: the integer data type and the fixed-point fractional data type. The Q format denotes fixed point fractions. The number of mantissa bits is specified after the letter Q. In the LMS filter, (with the PH data format), all operations use the Q15 format (a 16 bit value with one sign bit and 15 bits of mantissa).

Figure 3-3 A MIPS32 GPR in PH Data Format with Q15 Data Type



3.2 User Defined Instruction

Table 3-4 is the complete list of instructions needed for the filter codes described in the previous chapter. These have been separated into two types, arithmetic, and register move. There are five arithmetic instructions and three register move instructions as seen from the table.

Table 3-4 Instructions in the Filter UDI

Type	Instruction	Description	Use in Filter
Arithmetic	CIRADD rd, rs, rt	Circular address add	FIR, LMS
	DPAR.CH ud, rs, rt	Complex vector dot-product (real part) with accumulate.	Complex FIR
	DPAI.CH ud, rs, rt	Complex vector dot-product (imaginary part) with accumulate.	Complex FIR
	MADDQR.PH rd, rs, rt	Fractional vector multiply and add with rounding.	LMS
	MPA.PH ud, rs, rt	Integer vector multiply with accumulate.	FIR
Register Move	MFU rd, us, shift	Copy a UDI register value to a GPR.	All
	MTU ud, rs	Copy a GPR value to a UDI register.	All
	MFAS.PH.2D rd, n, shift	Copy two accumulator values to a GPR with optional right shift and saturation	Complex FIR

3.3 CorExtend Registers

There are 5 registers, 2(4*) accumulators that are 40 bits each, and a MC - multiply constant register which is 16 bits. These are listed in Table 3-5.

Table 3-5 Registers in the Filter UDI block

Register Number	Register Name	Register Size in bits	Description	Instructions that can write to the register	Instructions that read the register
0	Acc0	40	Accumulator 0	MPA, MTU	MPA, MFAS, MFU
1*	Acc1	40	Accumulator 1	MPA, MTU	MPA, MFAS, MFU
2*	Acc2	40	Accumulator 2	MPA, MTU	MPA, MFAS, MFU
3*	Acc3	40	Accumulator 3	MPA, MTU	MPA, MFAS, MFU
4	MC	16	Multiply constant register	MTU	MADDQR, MFU

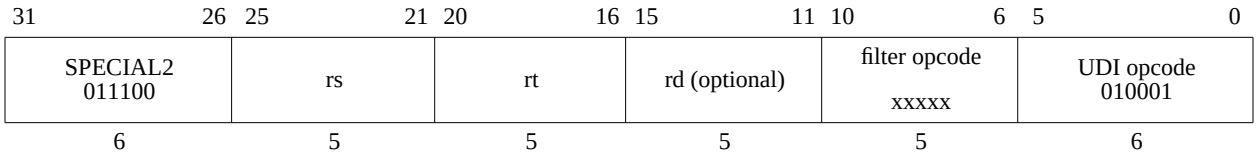
*Note: Accumulators 1, 2, & 3 are only required for the Complex FIR filter.

3.4 Instruction Bit Encoding

Note that all the instructions in Table 3-4 can be encoded into a single UDI opcode, as was done in our simulated implementation.

Figure 3-6 shows the basic UDI instruction format.

Figure 3-6 Filter UDI1 Instruction Format



Note that bits 0 to 3 are used to encode the UDI1 instruction, and the other bits from 6 to 10 are used to encode the instructions in Table 3-4.

An example encoding of the Filter UDI instructions into UDI1 (with opcode bits [3:0] = 0001) is shown in Table 3-7. Note that this encoding can be modified to any of the other possibilities to suit implementation, there is no reason to use either UDI1 or the specific opcode mapping shown in Table 3-7.

Table 3-7 Filter UDI Encoding of the opcode field or UDI1 (bits [3:0] = 0001)

filter opcode		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	-	-	-	-	-	-	-	-
1	01	-	-	-	-	-	-	MFU	MTU
2	10	-	-	-	-	-	-	MFAS.PH.2D	-
3	11	-	-	-	MPA.PH	CIRADD	MADDQR.PH	DPAR.CH	DPAI.CH

3.5 Estimated Gate Counts

For a MIPS32 implementation of the Filter CorExtend instructions on a 4KEc™, Table 3-8 shows one reasonable gate count estimate. Note that the 4KEc is about 180 K gates (this assumes 16 K 4-way caches, no RAMS, MIPS16e™, memBIST, EJTAG TAP, COP2 interface, and no PDtrace™ in TSMC 0.18g process). The proposed block adds about 13% to the total core area. Note that the multiplier is the largest component of the CorExtend block.

Table 3-8 UDI Block gate Count Estimate

Unit	Gates/Unit	# Units	Net
16x16 Multiplier	15 K	1	15 K
16+16 Adders	1.5 K	2	3 K
Accumulator Registers	0.5 K	1(4)	0.5 K(2 K)
Shifters	0.5 K	2	1 K
Overhead	3.5 K	-	3.5 K
Total			23 K (24.5 K)

To cut down on the size of the new block, note that only one multiplier is used. To execute the SIMD multiply-accumulate or the dot-product instructions, the single 16x16 multiplier is double-pumped. The multiplier is assumed to be fully pipelined, so that when it is double-pumped for the two different multiplies of the 2-way SIMD instruction, the repeat-rate of the instruction is two. That is, the dot-product or the MADDQR instruction can be issued only every other cycle. This is typically not a scheduling problem in the code as we see that at least one (load) instruction is available to be inserted between two dot-product instructions. The assumed latency and repeat rates of the UDI instructions is shown in Table 3-9.

Table 3-9 UDI Latency and Repeat Rates

Instruction	Latency	Repeat Rate
CIRADD	1	1
MPA, DPAR, DPAI, MADDQR	3	2
MFU, MTU, MFAS	1	1

Performance And Summary

4.1 Methodology

The MIPSsim™ simulator was used to evaluate the performance impact of the proposed instructions on the filter loops. The simulator can model and simulate all processor cores available from MIPS Technologies, and can estimate cycle counts fairly accurately. For the 4KEc™ core, the simulator is within 5% of the performance of the real hardware. This was verified with runs against the hardware. The simulator provides a CorExtend UDI interface that allows a user to describe the execution of the proposed instructions in C. This C library is then bolted onto the simulator which sends all UDI instructions across the interface to be emulated by the user-written C library code. The simulator interfaces with the GreenHills MULTI debugging environment to allow easy debugging as well as performance evaluation. The performance results provided in this chapter have been obtained using the methodology just described.

4.2 Performance Results

Table 4-1 shows the cycles per sample of the best compiler optimized C version of the loop, the MIPS32 version, and the MIPS32 + UDI version. The speedup columns compare the performance improvement of the basic MIPS32 assembly coding over the C version, and the User Defined Instructions version to the MIPS32 assembly version and to the C version.

Table 4-1 Performance of Filter Loops

Filter	C (best compiler optimization) cycles per sample	MIPS32 cycles per sample	MIPS32 + UDI cycles per sample	Speed-Up of MIPS32 to C	Speed-Up of MIPS32+UDI to MIPS32	Speed-Up of MIPS32+UDI to C
FIR	126	82	50	1.5	1.6	2.5
LMS	383	334	140	1.15	2.4	2.7
Complex FIR	263	251	91	1.05	2.8	2.9

Note the following interesting facts:

- The optimizing compiler does really well with the LMS and the Complex FIR codes, hence the speedups between the C versions and the MIPS32® assembly versions is fairly marginal (1.15 and 1.05 respectively).
- The basic FIR filter can be efficiently coded in MIPS32 assembly, hence the speedup of the MIPS32 version over the C version is about 1.5. Moreover, the UDI version only improves over the basic assembly version by about 1.6.
- For the LMS and Complex FIR filter, the basic MIPS32 assembly versions are not able to obtain much gain over the compiler optimized loops. Here the UDIs show a more significant improvement.
- The performance speedups of the UDI versions over the basic MIPS32 version range from 1.6 to 2.8.
- The performance speedups of the UDI versions over the C versions range from 2.5 to 2.9.

4.3 Summary

The purpose of this application note was to demonstrate how application code can be analyzed and how new instructions can be defined using the CorExtend® feature on the MIPS32® Pro Series® processor cores. The sample code chosen for this exercise were three DSP filter loops. These filters have a large and wide-ranging application in all areas of DSP, media processing, and speech processing.

We have demonstrated that with a handful of User Defined Instructions, the performance of these loops can be nearly doubled or tripled (in the case of the Complex FIR). We have shown that this exercise of determining the performance improvements of potential User Defined Instructions is possible with the available tool sets. We have also shown that the size of the UDI block is manageable and certain hardware optimizations are possible. For example, only implementing a single multiplier even if SIMD multiply instructions have been defined.

It is important to point out that the solutions shown in this application note are only examples used to illustrate the feasibility of CorExtend. It was not the goal to illustrate the best possible solution, at the expense of any amount of hardware. For example, if one were to use two multipliers and keep all the coefficients in local memory in the CorExtend block, this would give an additional 50-80% performance boost for the FIR filter. Thus, rather than using the proposed instructions as-is, this application note should be used as a guide to determine the best possible solution for a given application, for a given set of constraints for a given product.

The disadvantage of using new state such as the accumulator registers in the CorExtend block is that it makes the application code non-reentrant. Since there is no operating systems support to save and restore this extra architectural state, there can only be one instance of the application active at any given time in the processor. This restriction can be avoided if only new instructions are implemented using CorExtend without the addition of any new state.

Instead of using this feature to boost performance, the speedup obtained using these additional instructions can be used to save power and lower the core clock frequency. If the application is dominated by filtering, then this permits a 65% reduction in clock frequency. But since filtering may only be part of the overall application, perhaps the operating frequency can be lowered by only about 10%. But even this would reduce the power consumption by about 20% (because core voltage can also be reduced by 5%). The reduction in core frequency has a corollary effect in that it allows the core to be synthesized more for area/power rather than for clock speed. The resulting savings in silicon area could offset the area increase due to the CorExtend block.

For both the LMS and Complex filters, with the CorExtend instructions, the dynamic footprint of the loops was about 37-38% smaller. That is, for the same number of samples, the loop executes 37-38% fewer instructions. This reduces the instruction bandwidth and the corresponding power consumed in the loop. In the case of the Complex filter, the static footprint increases, but this should not be relevant since the loop is small and fits in the cache. Hence, CorExtend enables a power reduction as much as it enables a performance boost for the Pro Series cores.

4.4 References

[1] http://www.mips.com.publications/processor_architecture.html, "MIPS32® Architecture for programmers", Volumes I and II.

[2] "Digital Signal Processing--A Practical Approach", by Emmanuel C. Ifeachor and Barrie W. Jervis, ISBN 0-201-54413-X, published by Addison-Wesley.

[3] "Digital Signal Processing--A Computer Science Perspective", by Jonathan Stein, ISBN 0-471-29546-9, published by John Wiley & Sons, Inc.

[4] "MIPS32® 4KE™, 4KS™, and M4K™ Processor Core User Defined Instructions Implementor’s Guide"

[5] "DSP Processor Fundamentals--Architecture and Features", by Phil Lapsley, Jeff Bier, Amit Shoham, and Edward A. Lee, ISBN 0-7803-3405-1, published by IEEE Press.

Appendix A

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions and EJTAG register definitions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

Revision	Date	Author	Description
00.10	October 18, 2002	H/RT	First draft of application note
00.20	January 20, 2003	RT	Additional editing of document
01.01	June 30, 2008	EF	Update document title