Application Note, V1.0, June 2002

AP32073

TirCore®1 DSP Kernel Benchmarks

TriCore®

32-bit Unified Processor

32bit

Microcontrollers

Never stop thinking

Table of Contents

Page

1 **Introduction** 2
1.1 Purpose and Audience 2
1.2 Organization 2
1.3 Results Summary 2

2 **FIR Filters** 3
2.1 Introduction 3
2.2 Real 16 x 16-bit FIR Filter with N Taps and 48-bit Accumulation 3
2.3 Real 16 x 16-bit FIR Filter with N Taps and 32-Bit Accumulation with Saturation 7
2.4 Complex 16 x 16-bit FIR Filter with N Complex Taps and 48-bit Accumulation 8
2.5 Benchmark Results and Variations 11
2.6 Programming Summary 12

3 **Vector Quantization** 13
3.1 Introduction 13
3.2 16-bit Euclidean Distance 13
3.3 Codebook Search 14
3.4 Benchmark Results and Variations 15
3.5 Programming Summary 16

4 **Block Floating-Point** 17
4.1 Introduction 17
4.2 Minimum Block Exponent 17
4.3 Scaling 18
4.4 Benchmark Results and Variations 20
4.5 Programming Summary 20

5 **Levinson-Durbin Recursion** 21
5.1 Introduction 21
5.2 Complex with 16-Bit Precision and Rounding 21
5.3 Benchmark Results and Variations 24
5.4 Programming Summary 24

6 **LMS Adaptive Filter** 25
6.1 Introduction 25
6.2 Real 16 x 16-bit FIR Delayed LMS Adaptive Filter with N Taps 25
6.3 Benchmark Results and Variations 26
6.4 Programming Summary 26

7 **FFT Butterfly** 28
7.1 Introduction 28
7.2 Complex 16 x 16-bit Radix-2 Decimation-in-Time FFT Butterfly 28
7.3 Benchmark Results and Variations 32
7.4 Programming Summary 32

1 Introduction

This Application Note describes six digital signal processing (DSP) functions which are the basic algorithms in many signal processing system applications today. Their implementation on the TriCore Unified Processor is illustrated. They make use of the TriCore-1.2 architecture as embodied in the TC10, the first commercial product.

1.1 Purpose and Audience

A purpose is certainly to give numerical measures or benchmarks for the kernels of common DSP algorithms on the TriCore and show by example its high performance. However, the primary purpose is to illustrate in detail how to program and map onto the TriCore these DSP functions in a larger system context. They can then provide a model of good programming practice on the TriCore for any DSP function. The intended audience is experienced DSP programmers.

1.2 Organization

For each of the six functions the same sequence will be used in their explanation. After an introduction to the algorithm and common variations, the data, and if needed, the program structure will be shown mapped onto the TriCore architecture. This will be followed by the annotated program listing with further explanation. The instruction count and performance measures with variations or trade-offs will be given and then a final summary of the important programming points that have been illustrated.

1.3 Results Summary

The following summarizes the instruction cycle counts for the six DSP functions with their set-up prologs, their kernels and their exiting epilogs.

Table 1
Instruction Count Benchmark Summary

Function	Prolog	Kernel	Epilog
FIR - Real with 48-bit Accumulation	5	N/2 + 2	2
FIR - Real with Saturation	5	N/2 + 2	2
FIR - Complex with 48-bit Accumulation	5	2N + 2	2
Vector Quantization - Euclidean Distance	5	3N/2 + 2	1
Vector Quantization - Distance + Codebook Search	7	3MN/2 + 7M + 2	1
Block Floating-Point - Minimum Exponent	3	N + 2	3
Block Floating-Point - Exponent + Scaling	3	2N + 4	3
Levinson-Durbin Recursion	5	3N + 2	1
LMS Adaptive Filter	5	N + 2	2
FFT Butterfly	7	5N/2 + 2	1

2 FIR Filters

The finite-impulse-response (FIR) filter is the most common DSP function in many applications. With simple, but highly repetitive arithmetic and simple data structures it is the primary benchmark to measure computation-limited speed.

2.1 Introduction

A FIR filter is a continuing computation over time of the form:

$$\text{filter_out}(t) = \sum_{n=0}^{n=N-1} \text{in_data}(t-n) \times \text{coef}(n)$$

where N is the number of multiply-accumulates or filter taps on the uniformly sampled signal input data points with the index *t*.

Common algorithm variations are for real or complex input data, real or complex filter coefficients, N being even or odd and the coefficients being symmetrical (coef [N/2 + *n*] = coef [N/2 - *n*]) or not. Common arithmetic variations involve the signal and coefficient precisions, the amount of accumulation precision and the scaling and/or saturation strategies. Additional considerations when selecting an algorithm can be speed (i.e. maximum real-time bandwidth), minimum total program size or minimum data memory size including register file usage.

Three variations are provided here that illustrate real or complex data and accumulation with or without saturation.

2.2 Real 16 x 16-bit FIR Filter with N Taps and 48-bit Accumulation

This is the common case where real input and output data and coefficients are all 16-bit signed numbers with 32-bit products being accumulated to 48 bits. The number of taps N is a multiple of four and the impulse response is not assumed to be symmetrical.

This computation is mapped onto the TriCore architecture as shown in Figure 1. If the input data and coefficients are ordered in the data register file as shown, then four 16 x 16-bit multiply-adds can be done in two instruction cycles with two `maddm.h` instructions using the data path as drawn. Simultaneously the 64-bits of input data and coefficients can be updated with two loads from the data memory. Addresses for the circular input data buffer and the linearly ordered coefficients in data SRAM are generated in the address register file by the `ld.d` load instructions.

Both the two `maddm.h` and two `ld.d` kernel 32-bit instructions can issue in parallel because they are in the 64-bit wide instruction memory as shown. They execute in a zero-overhead loop for N/4 times since four multiply-accumulates take place each pass through the loop. A single input `in_data[t]` from DRAM memory or an I/O peripheral is made to the circular data buffer to initiate the filter process and a single output, `filter_out[t]`, is made from the accumulation at its end.

An example program using this kernel is in Figure 2. The kernel is illustrated in a system context with input and output operations and written so that it can be used independently, i.e. all registers are loaded or initialized in the prolog. It, however, lacks the overhead to make it a fully parametrizable filter subroutine. It is shown with two epilogs, one that outputs the most-significant 16 bits of the 48-bit accumulation, or a second that selects a 32-bit portion for an internal register d0. Processing time without I/O is 8 + N/2 instruction cycles including 2 for entering/exiting the loop.

Figure 1
Real 16 x 16-bit FIR Filter with N Taps and 48-bit Accumulation

```
_FIR1:
           ;;input              ;four instruction cycles

  ld.da    a2/a3,saveptr        ;load previously saved circular input data
                                ;pointer of base address, index I and length N
  ld.h     d0,new_data          ;load new_data
  st.h     [a2/a3+c]2,d0        ;store new_data into circular input data buffer
                                ;in_data and increment index I
  st.da    saveptr,a2/a3        ;save circular input data pointer

           ;;prolog             ;five instruction cycles

  lea      a0,loopcount         ;load absolute the loop count N/4 minus one
  lea      a1,coefptr           ;load absolute the coefficient pointer
  ld.da    a2/a3,saveptr        ;load previously saved circular input data
                                ;pointer of base address, index I and length N
  mov      d0,#0                ;zero lower half of accumulator,
  ld.w     d5,[a1+]4            ;and in parallel load coef0 and coef1
  mov      d1,#0                ;zero upper half of accumulator,
  ld.d     e2,[a2/a3+c]8        ;and in parallel load data0-3 from circular
                                ;input buffer in memory

           ;;kernel             ;two instruction cycles

firloop:
  maddm.h  e0,e0,d2,d5ul,#1     ;add [(data0)*(coef0)+(data1)*(coef1)]
                                ;left-shifted by 16 to the accumulator,
  ld.d     e4,[a1+]8            ;and in parallel load coef2-5
  maddm.h  e0,e0,d3,d4ul,#1     ;add [(data2)*(coef2)+(data3)*(coef3)]
                                ;left-shifted by 16 to the accumulator,
  ld.d     e2,[a2/a3+c]8        ;and in parallel load data4-7
loop       a0,firloop           ;repeat N/4 times

           ;;epilog1            ;16-bit output - two instruction cycles

  st.da    saveptr,a2/a3        ;save circular input data pointer
  st.h     filter_out,d1        ;store least-significant 16-bits of most-
                                ;significant word of output data to least-
                                ;significant 16 bits of memory or I/O

           ;;epilog2            ;32-bit result - two instruction cycles

  st.da    saveptr,a2/a3        ;save circular input data pointer
  dextr    d0,d1,d0,#20         ;extract desired 32-bit portion of accumulation
```

Figure 2
Program for Real 16 x 16-bit FIR Filter with N Taps and 48-bit Accumulation

The input process will typically be initiated by an interrupt indicating new data is available. The circular addressing mode invisibly uses the base address and buffer length in the double register a2/a3 to add data to the input buffer in the oldest location. The prolog later loads all address parameters and initializes the data register file for the kernel loop. Upon completing the kernel loop the epilog is started.

Note how the data and coefficients are offset in data memory so that the double-word loads in the kernel fall on double-word boundaries. The generic data and coefficient indices shown on the

operands in the data register file are for the operands at the time of the multiply-accumulate operation. They vary as a result of the load operations going through the loop as illustrated in Figure 3 for the first two passes. The shaded registers are the operands for the multiply-accumulate.

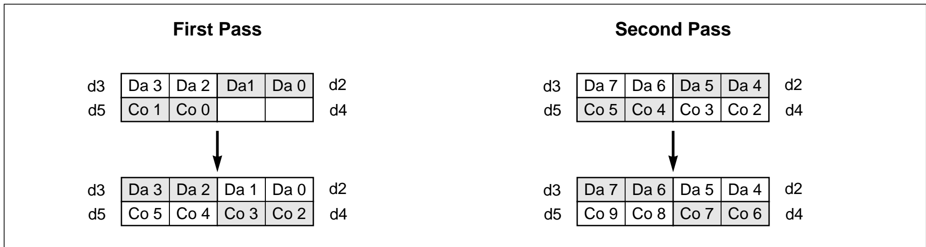

Figure 3
Operands in the Data Register File for the First Two Passes Through the Loop

The individual arithmetic operations in the first `maddm.h` instruction with the upper, lower (UL) operand configuration in the kernel are shown schematically in Figure 4. With the operation modifier n set to one the products are left-justified before addition.

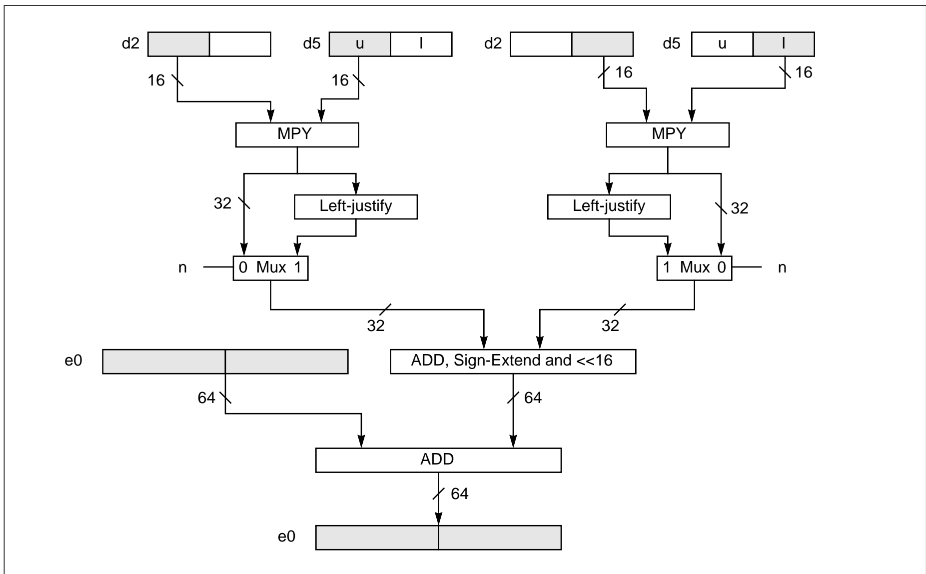

Figure 4
Arithmetic Operations in the Dual Multiply-Accumulate (Upper, Lower) Instruction `maddm.h`

2.3 Real 16 x 16-bit FIR Filter with N Taps and 32-Bit Accumulation with Saturation

The program in Figure 5 is a variation with only 32-bits accumulated and saturation is used on each of two accumulations with the `madds.h` instruction. Each addition is independently saturated after the double multiplies as shown in Figure 6 for the first such instruction. The two accumulations are added after the end of the loop. The operation modifier n is set to one on the `madds.h` instructions so that the results are left-justified as assumed fractional numbers. This precludes a guard bit for overflow but does assure an additional bit of resolution in the assumed fractional portion. The product of minus one times minus one is represented as the largest positive fraction that is less than one before accumulation.

```
_FIR2:
           ;;prolog              ;five instruction cycles

  lea      a0,loopcount          ;load absolute the loop count N/4 minus one
  lea      a1,coefptr            ;load absolute the coefficient pointer
  ld.da    a2/a3,saveptr         ;load previously saved circular input data
                                 ;pointer of base address, index I and length N
  mov      d0,#0                 ;zero even portion of accumulator,
  ld.w     d5,[a1+]4             ;and in parallel load coef0 and coef1
  mov      d1,#0                 ;zero odd portion of accumulator,
  ld.d     e2,[a2/a3+c]8         ;and in parallel load data0-3 from circular
                                 ;input buffer in memory

           ;;kernel              ;two instruction cycles

firloop:
  madds.h  e0,e0,d2,d5ul,#1      ;add (data0)*(coef0) and (data1)*(coef1)
                                 ;to the accumulations,
  ld.d     e4,[a1+]8             ;and in parallel load coef2-5
  madds.h  e0,e0,d3,d4ul,#1      ;add (data2)*(coef2) and (data3)*(coef3)
                                 ;to the accumulations,
  ld.d     e2,[a2/a3+c]8         ;and in parallel load data4-7
loop       a0,firloop            ;repeat N/4 times

           ;;epilog              ;32-bit output - two instruction cycles

  adds     d0,d1,d0              ;add the odd to the even accumulation,
  st.da    saveptr,a2/a3         ;and in parallel save circular input data pointer
  st.w     filter_out,d0         ;store 32-bit output data to memory or I/O
```

Figure 5
Program for Real 16 x 16-bit FIR Filter with N Taps and 32-bit Accumulation with Saturation

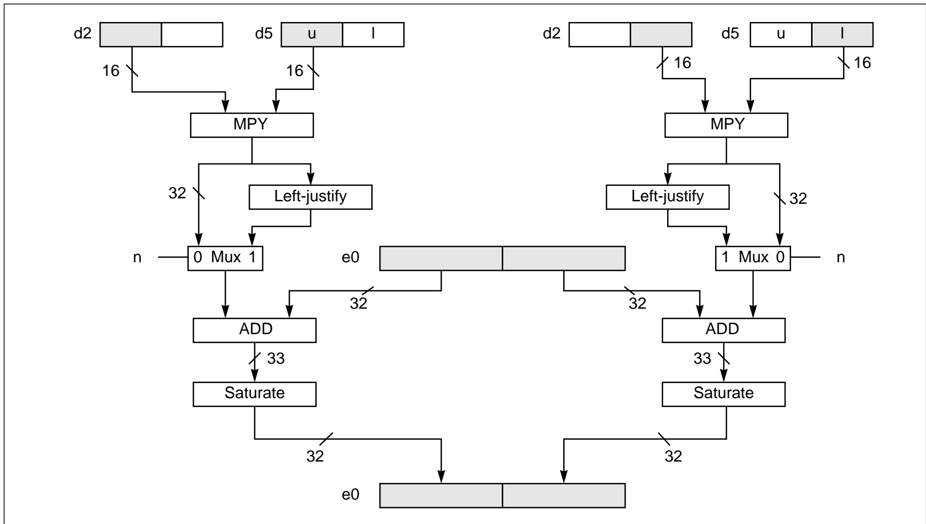

Figure 6
Arithmetic Operations in the Dual Multiply-Accumulate (Upper, Lower) with Saturation Instruction `madds.h`

2.4 Complex 16 x 16-bit FIR Filter with N Complex Taps and 48-bit Accumulation

If the data and impulse response are complex, with N real and N imaginary parts, then if the data memory and register files are configured as in Figure 7 the computation program in Figure 9 applies. Now the single real multiply-accumulate is replaced by a complex multiply-accumulate of four multiplies, three additions and a subtraction. There are now two half-words in the input data full-word buffer of length N with N real and N imaginary parts. The same is true for the coefficients.

The loop processes first even and then odd data and coefficient real and imaginary pairs. Figure 8 shows the first instruction in the kernel which subtracts the product of the two imaginary parts from the product of the real parts and adds them to the running real accumulation.

Note that the data and coefficients are not offset in data memory because there are no double-word loads in the kernel that must fall on double-word boundaries. Data register operands vary as a result of the load operations going through the loop as illustrated in Figure 10 for the first pass. The shaded registers with matching degrees of shading are the operands for the two multiplies. This more straightforward ordering of operands is the result of having four groups of parallel instructions in the loop. N must still be a multiple of only four as in the previous FIR filter examples.

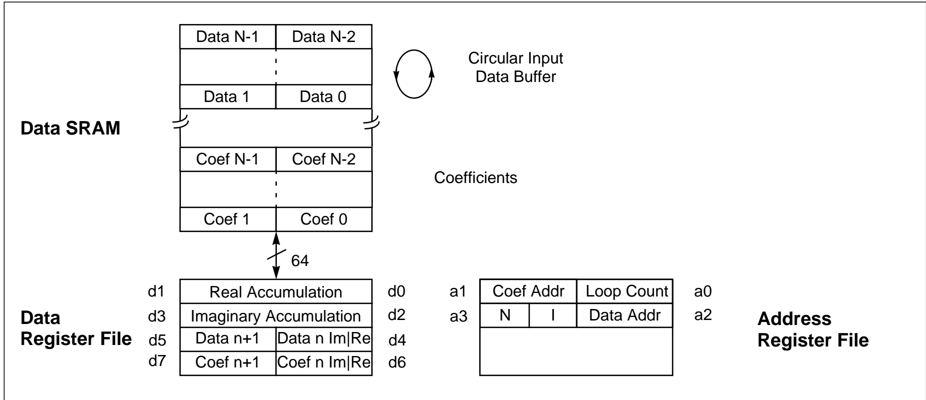

Figure 7
Register File Assignments and Data Memory Organization in the Complex 16 x 16-bit FIR Filter with N Complex Taps and 48-bit Accumulation

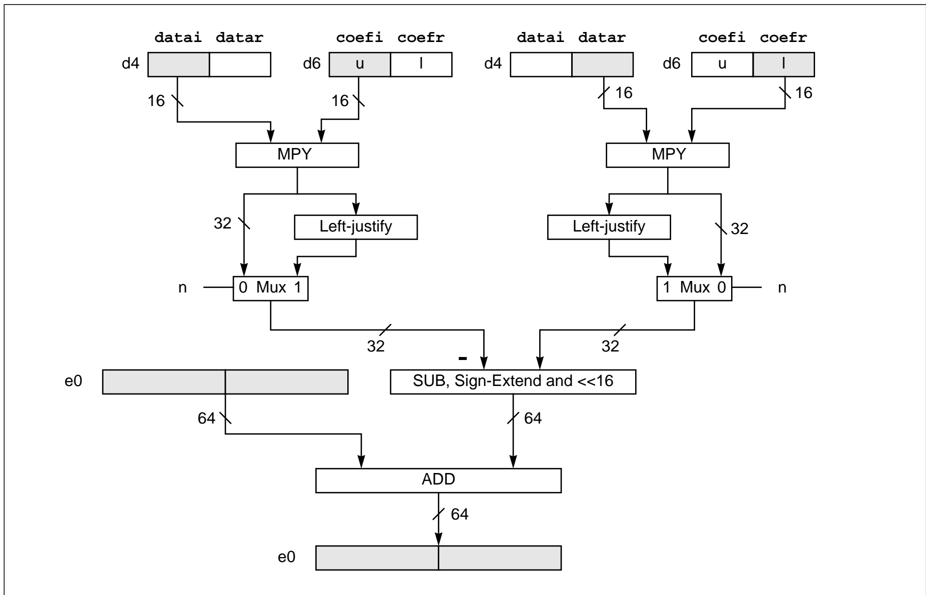

Figure 8
Arithmetic Operations in the Dual Multiply-Subtract-Add (Upper, Lower) Instruction `msubadm.h`

```
_FIR3:
            ;;prolog            ;five instruction cycles

   mov      e0,#0               ;zero real accumulator
   lea      a0,loopcount        ;and load absolute the loop count N/2 minus one
   mov      e2,#0               ;zero imaginary accumulator
   lea      a1,coefptr          ;and load absolute the coefficient pointer
   ld.da    a2/a3,saveptr       ;load previously saved circular input data
                                ;pointer of base address, index I and length N
   ld.w     d6,[a1+]4           ;load coefr0 and coefi0
   ld.w     d4,[a2/a3+c]4       ;load datar0 and datai0 from circular real
                                ;input buffer in memory

            ;;kernel            ;four instruction cycles

firloop:
   msubadm.h e0,e0,d4,d6ul,#1   ;add [(datar0)*(coefr0)-(datai0)*(coefi0)]
                                ;left-shifted by 16 to the real accumulation,
   ld.w     d7,[a1+]4           ;and in parallel load coefr1 and coefi1
   maddm.h  e2,e2,d4,d6lu,#1    ;add [(datar0)*(coefi0)+(datai0)*(coefr0)]
                                ;left-shifted by 16 to the imaginary accumulation,
   ld.w     d5,[a2/a3+c]4       ;and in parallel load datar1 and datai1
   msubadm.h e0,e0,d5,d7ul,#1   ;add [(datar1)*(coefr1)-(datai1)*(coefi1)]
                                ;left-shifted by 16 to the real accumulation,
   ld.w     d6,[a1+]4           ;and in parallel load coefr2 and coefi2
   maddm.h  e2,e2,d5,d7lu,#1    ;add [(datar1)*(coefi1)+(datai1)*(coefr1)]
                                ;left-shifted by 16 to the imaginary accumulation,
   ld.w     d4,[a2/a3+c]4       ;and in parallel load datar2 and datai2
loop        a0,firloop          ;repeat N/2 times

            ;;epilog            ;32-bit output - two instruction cycles

   dextr    d0,d1,d0,#20        ;extract desired 32-bits of real accumulation,
   st.da    saveptr,a2/a3       ;and save circular input data pointer
   dextr    d2,d3,d2,#20        ;extract desired 32-bits of imaginary accumulation
```

Figure 9
Program for Complex 16 x 16-bit FIR Filter with N Complex Taps and 48-bit Accumulation

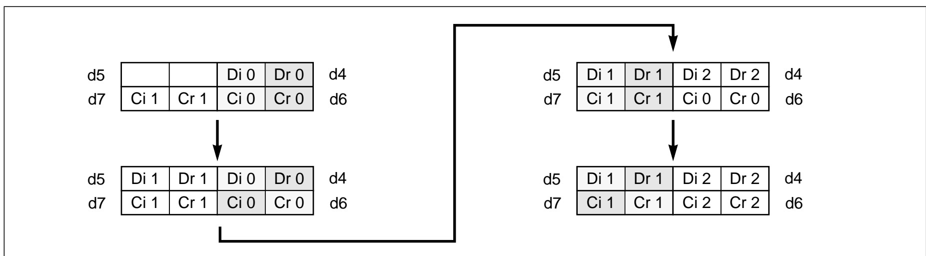

Figure 10
Operands in the Data Register File for the First Pass Through the Loop

2.5 Benchmark Results and Variations

As Table 2 shows for a 16 x 16-bit FIR filter, there is no speed difference with the TriCore whether saturation or higher precision accumulation is used; it is essentially an instruction cycle for every two filter taps. The complex FIR, where there are four times as many multiply-accumulates is approximately four times slower or one-half tap per instruction cycle.

Table 2
FIR Filter Instruction Count Benchmark Summary

Function	Prolog	Kernel	Epilog
FIR - Real with 48-bit Accumulation	5	N/2 + 2	2
FIR - Real with Saturation	5	N/2 + 2	2
FIR - Complex with 48-bit Accumulation	5	2N + 2	2

These examples have assumed that N, the number of filter taps, was both even and a multiple of four for ease in ordering the data. Reducing the value of N from a multiple of four offers no speed advantage for this precision so the longer impulse response may just as well be used. In a related manner, since multiply-adds take no longer than additions alone, there is no speed advantage with a symmetrical impulse response although coefficient memory is reduced.

These examples are all for 16-bit data and coefficients, the most common precision for DSP applications. If higher precision is needed, the programs scale linearly to 32 x 32-bits with 64-bits of accumulation because of the wide data buses. The same data organization applies where a new n index equals twice the old n index. Four `maddm` instructions are used in the kernel loop so that the processing time only doubles with twice the precision. There is no speed advantage for precisions of less than 16 x 16-bits for the multiply-accumulate, however the byte addressing and full shifting and sign-extension capabilities make for easy reductions in data memory with lower precision data. The I/O, often of a different precision, likewise benefits from this capability in speed since I/O peripherals too are directly mapped into the memory space.

With the TriCore’s large SRAM data buffers, the data structures in these examples will work directly for most FIR filters found in modern application, FIR filters seldom exceed 256 taps. But if necessary these same structures work with the on-chip DRAM because it too has a 64-bit data bus. The speed is only nominally reduced by the amount of refresh time.

The example FIR filter programs shown here are typical for many applications where the filtering is but one of many tasks. Where they need the speed of an individual block of code without the overhead of a generic subroutine. However the parameters are restored to memory after use rather than remaining in dedicated registers. If real-time filtering is a large component of the processing then the fast context switching ability of the TriCore should be considered as a way of keeping the speed of dedicated registers without having to reserve their use.

2.6 Programming Summary

These FIR programs illustrate how to use the following basic characteristics of the TriCore architecture which are generally helpful in DSP and won't be repeated for the DSP functions in the remainder of this Application Note.

- Parallel Arithmetic Operations in Parallel with Wide Data Transfers
  - The FIR clearly illustrates how the 32-bit multiplier and ALU can be divided to do parallel arithmetic operations on half-word data of 16 bits in the register file.
  - Double-word 64-bit load and store transfers show how complex address generation capability is available to supply multiple half-word operands from memory to the register file for parallel arithmetic operations. These transfers take place in parallel with the arithmetic operations because of the dual issuing of instructions.
- Arithmetic Operations
  - The FIR illustrates how data registers and operands of different precisions are ordered and aligned for the fastest processing. Arithmetic operations for accumulation, saturation, alignment (the Q formats) and scaling (extraction) are shown in dealing with data with different precision requirements and with the common DSP formats of two's complement signed fractional numbers.
- Data Structure
  - FIR filters use the simple but important circular data buffer structure that is largely invisible in the program execution.
- Control Structure
  - The FIR uses well the zero-overhead `LOOP` instruction for its tight inner loop. The prologs and epilogs clearly show how the loop is initialized and terminated.

None of these are special techniques that are of particular interest in programming FIR filters, they are all broadly applicable.

3 Vector Quantization

Vector quantization is a common form of communications speech encoding that is compute intensive. The best fit is found between an incoming speech sample vector and a codebook of reference coefficient vectors. This involves finding the minimum distance between the sample vector and all codebook entries.

3.1 Introduction

The squared difference between a sample vector *i* of length N and a codebook coefficient vector *j* of length N is of the form:

$$\text{distance}_{ij} = \sum_{n=0}^{n=N-1} [\text{sample_vector}_i(n) - \text{coefficient}_j(n)]^2$$

Because the signal is speech the precision is almost always 16-bits. The only common variations are combining it with the minimizing process comparing other codebook entries or to use a weighting function on the distances to compensate for non-uniform energies in the codebook entries.

The minimum distance for a sample vector *i* with a codebook of M entries is of the form:

$$\text{minimum_distance}_i = \underset{j=0}{\overset{j=M-1}{\text{MIN}}} [\text{distance}_{ij}]$$

The index of the codebook entry *j* with the minimum distance is the desired result, not the actual minimum distance for the *i* sample vector.

3.2 16-bit Euclidean Distance

If the sample vector and coefficient vector are ordered in the data register file as shown in Figure 11, then two 16-bit differences can be computed at once with the dual subtraction `subs.h` instruction illustrated in Figure 12 and the two squaring and summation operations can be done with the `madds.h` instruction.

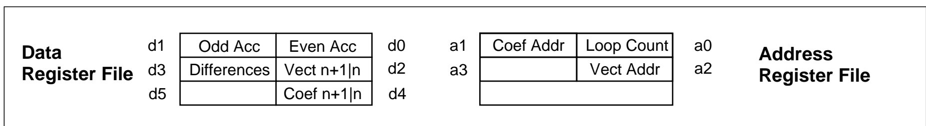

Figure 11
Register File Assignments in the Euclidean Distance Computation

The complete program is in Figure 13 with the initializing prolog and final addition of the two running even and odd accumulations in the exiting prolog. For vectors of length N the kernel takes 3N/2 + 2 instruction cycles, including entering/exiting the loop, where N is even. The `madds.h` instruction takes two instruction cycles before the integer unit is free for the `subs.h` instruction.

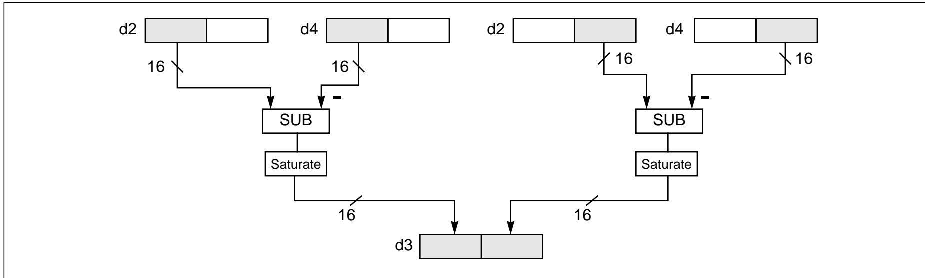

Figure 12
Arithmetic Operations in the Dual Subtract Instruction `subs.h`

```
_VQ1:
            ;;prolog                ;five instruction cycles

   lea      a0,loopcount            ;load absolute the loop count N/2 minus one
   lea      a1,coefptr              ;load absolute the coefficient pointer
   lea      a2,vectptr              ;load absolute the vector pointer
   mov      d0,#0                   ;zero the even distance accumulation d0
   ld.w     d4,[a1+]4               ;and in parallel load coef0 and coef1
   mov      d1,#0                   ;zero the odd distance accumulation d1
   ld.w     d2,[a2+]4               ;and in parallel load vect0 and vect1

            ;;kernel                ;three instruction cycles

vqloop:
   subs.h   d3,d2,d4                ;d3 = vect0 - coef0 | vect1 - coef1
   ld.w     d4,[a1+]4               ;and in parallel load coef2 and coef3
   madds.h  e0,e0,d3,d3ul,#1        ;d0 = d0 + [vect0 - coef0] * [vect0 - coef0]
                                    ;d1 = d1 + [vect1 - coef1] * [vect1 - coef1]
   ld.w     d2,[a2+]4               ;and in parallel load vect2 and vect3
loop        a0,vqloop               ;repeat N/2 times

            ;;epilog                ;one instruction cycle

   adds     d0,d1,d0                ;add the odd to the even distance accumulation
   st.w     distance,d0             ;and in parallel store distance
```

Figure 13
Program for Euclidean Distance in Vector Quantization

3.3 Codebook Search

The codebook search can be added to the previous distance calculation. An outer loop repeats the distance computation for each *j* of the M codebook coefficient vector entries and uses the unsigned less-than `lt.u` instruction to keep a running minimum and coefficient vector start address.

If the data memory and register files are ordered as shown in Figure 14, then the program in Figure 15 finds the minimum distance and stores the address pointer to the next coefficient vector from the one that produced the minimum. Note that the codebook entries are contiguous.

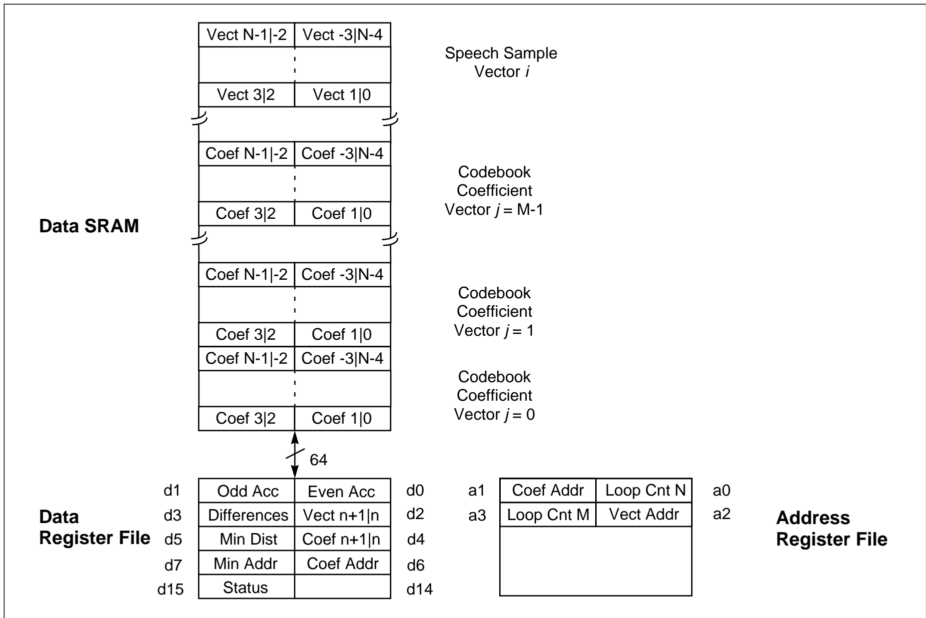

Figure 14
Register File Assignments and Data Memory Organization in the Codebook Search

The distance kernel inner loop still takes 3N/2 + 2 instruction cycles while the codebook search kernel outer loop takes 5M + 2 instruction cycles including the loop entering/exiting. The total kernel count is thus M(3N/2 + 2) + 5M + 2 instruction cycles per speech sample input vector.

3.4 Benchmark Results and Variations

The total instruction cycle count is in Table 3. N the number of samples in the vectors is assumed even while M the size of the codebook is not. For odd N, making the last entry in each vector zero preserves the speed advantage of N being even.

Table 3
Vector Quantization Instruction Count Benchmark Summary

Function	Prolog	Kernel	Epilog
Vector Quantization - Euclidean Distance	5	3N/2 + 2	1
Vector Quantization - Distance + Codebook Search	7	M(3N/2 + 2) + 5M + 2 = 3MN/2 + 7M + 2	1

```
_VQ2:
            ;;prolog                 ;seven instruction cycles

   lea      a0,nloopcount            ;load absolute the n loop count N/2 minus one
   lea      a1,coefptr               ;load absolute the coefficient pointer
   lea      a2,vectptr               ;load absolute the sample vector pointer
   lea      a3,jloopcount            ;load absolute the j loop count M minus one
   movh     d5,#0xffff               ;initialize the minimum distance d5
   mov      d0,#0                    ;zero the even distance accumulation d0
   ld.w     d4,[a1+]4                ;and in parallel load coef0 and coef1
   mov      d1,#0                    ;zero the odd distance accumulation d1
   ld.w     d2,[a2+]4                ;and in parallel load vect0 and vect1

            ;;jkernel                ;five instruction cycles
            ;;nkernel                ;three instruction cycles

vqjloop:
vqnloop:
   subs.h   d3,d2,d4                 ;d3 = vect0 - coef0 | vect1 - coef1
   ld.w     d4,[a1+]4                ;and in parallel load coef2 and coef3
   madds.h  e0,e0,d3,d3ul,#1         ;d0 = d0 + [vect0 - coef0] * [vect0 - coef0]
                                     ;d1 = d1 + [vect1 - coef1] * [vect1 - coef1]
   ld.w     d2,[a2+]4                ;and in parallel load vect2 and vect3
loop        a0,vqnloop               ;repeat N/2 times

   adds     d0,d1,d0                 ;add the odd to the even distance accumulation
   lt.u     d15,d0,d5                ;if d0 distance less than d5 minimum then d15 ≠ 0
   cmov     d5,d15,d0                ;if d15 ≠ 0 then d0 moves to minimum d5
   mov.d    d6,a1                    ;move current coefficient pointer to d6
   cmov     d7,d15,d6                ;if d15 ≠ 0 then d7 becomes the minimum coefficient
                                     ;address pointer plus N
loop        a3,vqjloop               ;repeat M times

            ;;epilog                 ;one instruction cycle

   st.a     minimum_address,d7       ;store address pointer to minimum distance
                                     ;coefficient vector address plus N
```

Figure 15
Program with Codebook Search Added to Euclidean Distance for Vector Quantization

3.5 Programming Summary

In addition to the basic characteristics of the TriCore architecture used in the FIR filter benchmarks, vector quantization illustrates the following additional ones:

- Arithmetic Operations and Data Structure Operations
  - The codebook search shows the tight link between the data arithmetic evaluations and concurrent operations in the address generation for the data structures. This use of conditional instructions without ponderous branch instructions can increase speeds substantially over conventional DSP architectures without such linking and conditional execution. This is valuable for what is a very common function in most encoding systems.

4 Block Floating-Point

Floating-point arithmetic can often be avoided in digital signal processing if the full dynamic range of the fixed-point data types are utilized. So-called block floating-point is a common technique used to improve the effective precision of a given fixed-point data type.

4.1 Introduction

In block floating-point, the individual data samples of a block of input data are scaled equally so that the largest sample (the one with the smallest fractional exponent) is just less than overflow. This is done in two steps, scanning the block of N samples for the minimum exponent:

$$\text{minimum_exponent}_i = \operatorname*{MIN}_{n=0}^{n=N-1} \exp[\text{in_data}_i(n)]$$

and then scaling the samples of the block with that exponent:

$$\text{out_data}_i(n) = \operatorname*{SHIFT}_{n=0}^{n=N-1} [\text{in_data}_i(n)] \ll \text{minimum_exponent}_i$$

4.2 Minimum Block Exponent

The dual count-leading-signs `cls.h` instruction can be used to determine the exponent of two 16-bit half-words at a time. It returns two counts of the redundant sign bits in two numbers which is the same as the number of left shifts to individually normalize them. The dual minimum `min.h` instruction illustrated in Figure 16 can then be used to determine two running minimums of exponents. Using the half-word data and other register allocations in Figure 17 then the kernel in the program of Figure 18 in N instruction cycles finds an even and an odd minimum. N is the number of data samples in the block and is even. In the epilog the minimum of these two is stored.

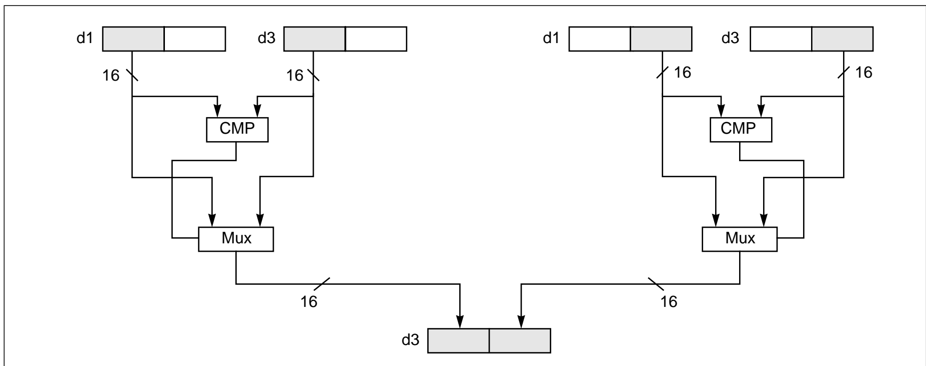

Figure 16
Operations in the Dual Minimum Instruction `min.h`

Data Register File	d1	Exp n+1|n	In n+1|n	d0	a1	In Addr	Loop Count	a0	Address Register File
	d3	Min n+1|n	Out n+1|n	d2	a3		Out Addr	a2	

Figure 17
Register File Assignments in the Minimum Block Exponent Computation

A prolog of three instructions cycle counts initiates the kernel which takes N + 2 cycles including entering/exiting the loop and then three cycles for the epilog.

```
_bex1:
           ;;prolog            ;three instruction cycles

  lea      a0,loopcount        ;load absolute the loop count N/2 minus one
  movh     d3,#16              ;initialize the odd minimum exponent d4 upper
  lea      a1,inptr            ;and in parallel load absolute input data pointer
  adds.u   d3,d3,#16           ;initialize the even minimum exponent d4 lower
  ld.w     d0,[a1+]4           ;and in parallel load in_data0 and in_data1

           ;;kernel            ;two instruction cycles

bexloop:
  cls.h    d1,d0               ;d1 = leading sign count d0 lower | cls d0 upper
  min.h    d3,d1,d3            ;if d1 < d3, d3 = d1 for upper and/or lower
  ld.w     d0,[a1+]4           ;and in parallel load in_data2 and in_data3
loop       a0,bexloop          ;repeat N/2 times

           ;;epilog            ;three instruction cycles

  sh       d1,d3,#-16          ;d1 = right-shifted even minimum exponent
  extr.u   d0,d3,#0,#16        ;d0 = extracted odd minimum exponent
  min.h    d3,d1,d0            ;d3 = minimum of d0 and d1
  st.h     minimum_exponent,d3 ;and in parallel store minimum_exponent
```

Figure 18
Program for Minimum Block Exponent

4.3 Scaling

Using the same register assignment in Figure 17, the scaling can be added to the program in Figure 18 to complete the block floating-point function as shown in Figure 20. The dual arithmetic shift `sha.h` instruction illustrated in Figure 19 maintains the same half-word use of the data registers. The scaling prolog nicely interleaves with the minimum block exponent epilog and there is no scaling epilog. Thus, the scaling adds only the additional N + 2 instruction cycles of its loop with entering/exiting to the minimum block exponent cycle count.

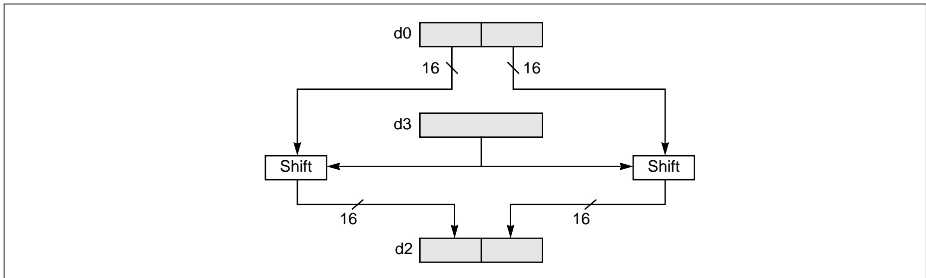

Figure 19
Operations in the Dual Arithmetic Shift Instruction `sha.h`

```
_bex2:
            ;;bex prolog          ;three instruction cycles

   lea      a0,loopcount          ;load absolute the loop count N/2 minus one
   movh     d3,#16                ;initialize the odd minimum exponent d4 upper
   lea      a1,inptr              ;and in parallel load absolute input data pointer
   adds.u   d3,d3,#16             ;initialize the even minimum exponent d4 lower
   ld.w     d0,[a1+]4             ;and in parallel load in_data0 and in_data1

            ;;bex kernel          ;two instruction cycles

bexloop:
   cls.h    d1,d0                 ;d1 = leading sign count d0 lower | cls d0 upper
   min.h    d3,d1,d3              ;if d1 < d3, d3 = d1 for upper and/or lower
   ld.w     d0,[a1+]4             ;and in parallel load in_data2 and in_data3
loop        a0,bexloop            ;repeat N/2 times

            ;;bex epilog/
            ;;s prolog            ;three instruction cycles

   sh       d1,d3,#-16            ;d1 = right-shifted even minimum exponent in d3
   lea      a1,inptr              ;and in parallel load absolute input data pointer
   extr.u   d0,d3,#0,#16          ;d0 = extracted odd minimum exponent in d3
   lea      a2,outptr             ;and in parallel load absolute output data pointer
   min.h    d3,d1,d0              ;d3 = minimum of d0 and d1
   lea      a0,loopcount          ;and in parallel load the loop count N/2 minus one

            ;;s kernel            ;two instruction cycles

sloop:
   ld.w     d0,[a1+]4             ;load in_data0 and in_data1
   sha.h    d2,d0,d3              ;out_data d2 = in_data d0 << minimum_exponent d3
   st.w     [a2+]4,d2             ;store out_data0 and out_data1
loop        a0,sloop              ;repeat N/2 times

            ;;epilog              ;no instruction cycles
```

Figure 20
Program for Minimum Block Exponent with Scaling

4.4 Benchmark Results and Variations

The total instruction cycle count is in Table 4. N the number of samples is assumed even. For odd N, making the last entry in the input data samples zero preserves the speed advantage of N being even. Block floating-point is most often used with low-precision data types like the 16-bit half-word example programs. However, the distinctive instructions for counting leading signs, minimization and shifting all apply to full-word 32-bit precision.

Table 4
Block Floating-Point Instruction Count Benchmark Summary

Function	Prolog	Kernel	Epilog
Block Floating-Point - Minimum Exponent	3	N + 2	3
Block Floating-Point - Exponent + Scaling	3	2N + 4	3

4.5 Programming Summary

In addition to the basic characteristics of the TriCore architecture used in the FIR filter and vector quantization benchmarks, block floating-point illustrates the following additional ones:

- Arithmetic Data Operations
  - The counting of leading redundant sign bits which is a powerful instruction that is broadly useful for removing redundant data for precision conserving purposes or encoding functions of many applications.
  - The ability to scan for independent dual minimums on half-words with a single cycle instruction.

5 Levinson-Durbin Recursion

The Durbin modification of the Levinson recursion is used in many time-varying signal processing applications, but most commonly in linear predictive speech coding.

5.1 Introduction

Computationally the Levinson-Durbin algorithm is generically the repeated complex multiplication of a data variable by a coefficient where:

$$\text{data}(t) = \text{data}(t-1) \times \text{coefficient}$$

so for a vector of N complex coefficients the output at point *t*, using the nomenclature of previous examples is:

$$\text{out_data}(t) = \prod_{n=0}^{n=N-1} \text{out_data}(t-1-n) \times \text{coefficient}(n)$$

As with all recursive functions precision is a major consideration. However, for speech there is a tremendous economy with 16-bits and N is often moderate being in the order of ten.

5.2 Complex with 16-Bit Precision and Rounding

For the common case where N is moderate then 16-bit precision for real and imaginary data and coefficients is adequate if rounding is used on all arithmetic operations. In the TriCore the half-word family of multiply `mulr.h` instructions and the family of `maddsur.h` multiply-add-subtract instructions provide a particularly flexible and powerful pair for doing complex multiplications with rounding.

For example the multiply in Figure 21 is but one of four possible configuration of multiplying the upper and lower half-words of register d2 with the various half-words of d4. Illustrated is the lower, lower (LL) choice, but UU, LU, and UL are also possible. Note that the 32-bit products can selectively be left-justified before the rounding to a 16-bit result. The multiply-add-subtract shown in Figure 22 is even more tuned to complex multiplication because it has the same half-word register operand flexibility combined with doing an additive accumulation in parallel with a subtractive accumulation, both with rounding.

With the simple pairing of real and imaginary coefficients and data in the full-word memory and register files shown in Figure 23 these two instructions can do the four multiplies, single addition and single subtraction of a complex multiply in three instruction cycles. N is the desired number of recursions and coefficients. Note that the out_data vector has N + 1 entries running from n = -1 to n = N - 1. The n = -1 value is the initial t = -1 input data value. The lone N - 1 value is usually the only desired result, but all intermediate values are saved.

The complete recursion program is in Figure 24. After initialization the kernel creates the first two real and imaginary products in the temporary register d0. The operands and result are labelled in Figure 21. Real values are always in the lower 16-bits of the half words, and imaginary in the upper. Then the remaining two multiplies and addition/subtraction on d0 proceed as labelled in Figure 22 with the resulting out_data rolling back to the start of the loop in register d2 again in Figure 21.

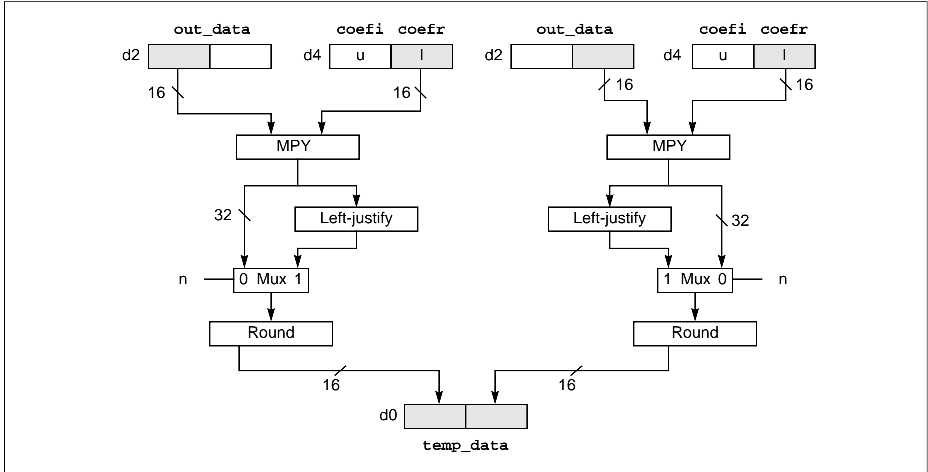

Figure 21
Operations in the Dual Multiply (Lower, Lower) with Rounding Instruction `mulr.h`

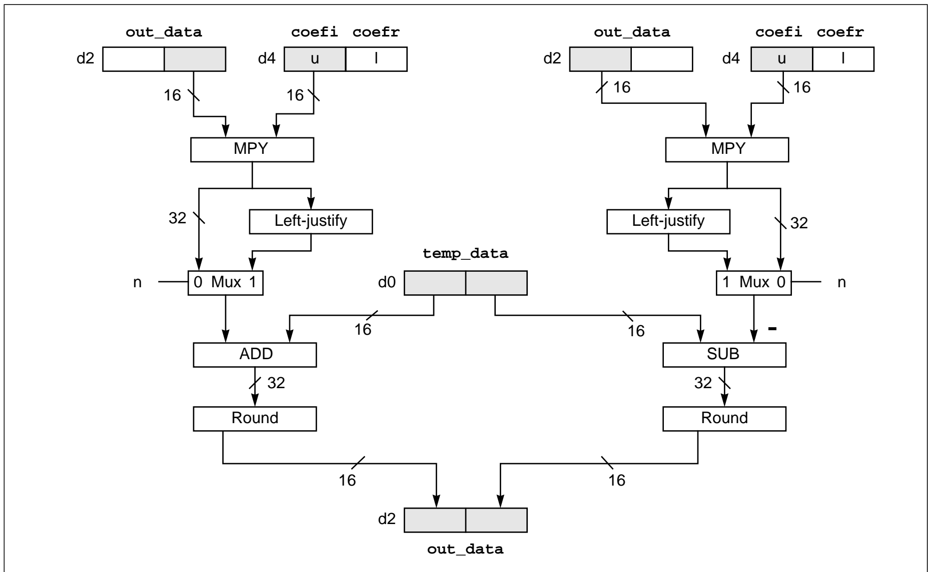

Figure 22
Arithmetic Operations in the Dual Multiply-Add-Subtract (Upper, Upper) with Rounding Instruction `maddsur.h`

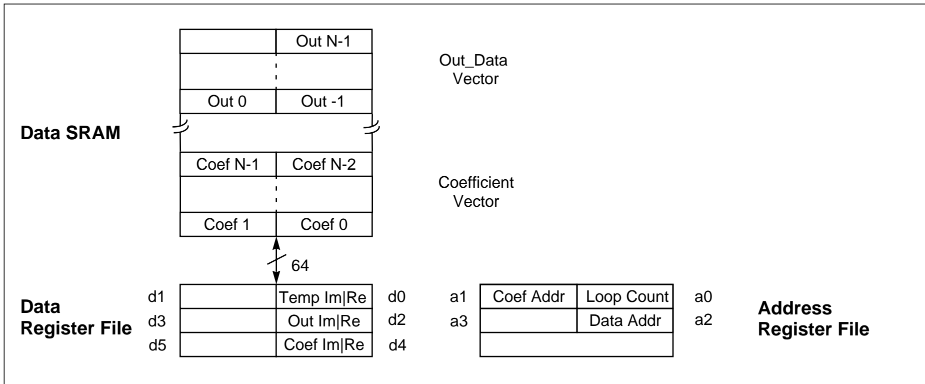

Figure 23
Register File Assignments and Data Memory Organization for the Levinson-Durbin Recursion

The final desired result is stored in the epilog. The complete processing time is 3N + 8 instruction cycles including 2 for entering/exiting the loop. The `maddsur.h` instruction takes two instruction cycles because the multiplier-accumulator pipeline latency delays the register d2 operand needed for the next `mulr.h` instruction in the loop.

```
_LD1:
             ;;prolog            ;five instruction cycles

   lea       a0,loopcount        ;load absolute the loop count N minus one
   lea       a1,coefptr          ;load absolute the coefficient vector pointer
   lea       a2,out_dataptr      ;load absolute the out_data pointer for N + 1 values
   ld.w      d4,[a1+]4           ;load coefr0 and coefi0 in d4
   ld.w      d2,[a2]             ;load initial data out_datar(-1) and out_datai(-1)

             ;;kernel            ;three instruction cycles

ldloop:
   mulr.h    d0,d2,d4ll,#1       ;temp_data = out_datai * coefr | out_datar * coefr
   st.w      [a2+]4,d2           ;and in parallel store current out_data
   maddsur.h d2,d0,d2,d4uu,#1    ;out_data = temp_datai + out_datar * coefi |
                                 ;temp_datar - out_datai * coefi
   ld.w      d4,[a1+]4           ;and in parallel load next coefr and coefi
loop         a0,ldloop           ;repeat N times

             ;;epilog            ;one instruction cycle

   st.w      [a2+]4,d2           ;store last out_data, the N + 1st value
```

Figure 24
Program for the Levinson-Durbin Recursion

5.3 Benchmark Results and Variations

The total instruction cycle count is in Table 5. N the number of recursions is assumed even. For odd N, making the last coefficient entry zero preserves the speed advantage of N being even. Then the last recursion output is ignored.

Table 5
Levinson-Durbin Instruction Count Benchmark Summary

Function	Prolog	Kernel	Epilog
Levinson-Durbin Recursion	5	3N + 2	1

A natural variation would be to use the 32-bit full-word precision including rounded 32-bit products. This would remove the two-to-one speed advantage of the dual operations and increase register and memory usages..

5.4 Programming Summary

In addition to the basic characteristics of the TriCore architecture used in the earlier benchmarks, this recursion illustrates the following additional ones:

- Arithmetic Data Operations
  - The large family of half-word operand register selections available with some instructions. As shown this can particularly be used to select operands out of previously generated results to speed recursive functions.
  - The dual multiplication with both addition and subtraction is a natural operation for use with complex arithmetic. Again the family of operand selections is helpful for complex data. Also, with rounding included in these instructions precision loss with half-words is minimized.

6 LMS Adaptive Filter

Adaptive filters, that is, filters whose coefficients are changed dynamically to adapt to some function are common in signal processing systems. It is difficult to do without using digital techniques.

6.1 Introduction

Adaptive filters are usually FIR because they are unconditionally stable and they are adapted by minimizing some error function derived by comparison with a reference signal. If the individual filter coefficients are linearly changed by the output error function times some gain factor multiplied by the input, then the error is minimized on a least-mean-squared (LMS) basis. For an N-tap filter the computation is of the form:

$$\text{filter_out}(t) = \sum_{n=0}^{n=N-1} \text{in_data}(t-n) \times \text{coef}_{t-1}(n)$$

where

$$\text{coef}_t(n) = \text{coef}_{t-1}(n) + \text{error}_t \times \text{in_data}(t-n)$$

for n equal zero to N - 1, and

$$\text{error}_t = \mu(\text{reference}_t - \text{filter_out}_t)$$

Mu (µ) is the correction gain factor. A common variation for computational simplicity is where the previous error function is used in the coefficient change so that:

$$\text{coef}_t(n) = \text{coef}_{t-1}(n) + \text{error}_{t-1} \times \text{in_data}(t-n)$$

This so-called delayed LMS computation avoids an extra coefficient and a data fetch in the kernel. All of the other variations discussed earlier for FIR filters apply. The adaptive filter is a recursion also because future outputs depend upon previous inputs.

6.2 Real 16 x 16-bit FIR Delayed LMS Adaptive Filter with N Taps

The program in Figure 26 uses a real N-tap FIR filter with 16 x 16-bit multiplies and 48-bit accumulation. The data memory and register files organization for the computation are shown in Figure 25. It is assumed the input data will continue to be loaded in the circular buffer, that initial coefficients have been loaded and that the error function will be determined from the previous data output compared with the reference signal and entered in register d8 as two repeated half-words.

Notice again how coefficients and data are offset in data memory for alternating double-word load and stores within the kernel loop on double-word boundaries. The coefficients shown with negative indices are for an initial false storing of updated coefficients that are not used. Also the final false Data N and Data N+1 are loaded but never used.

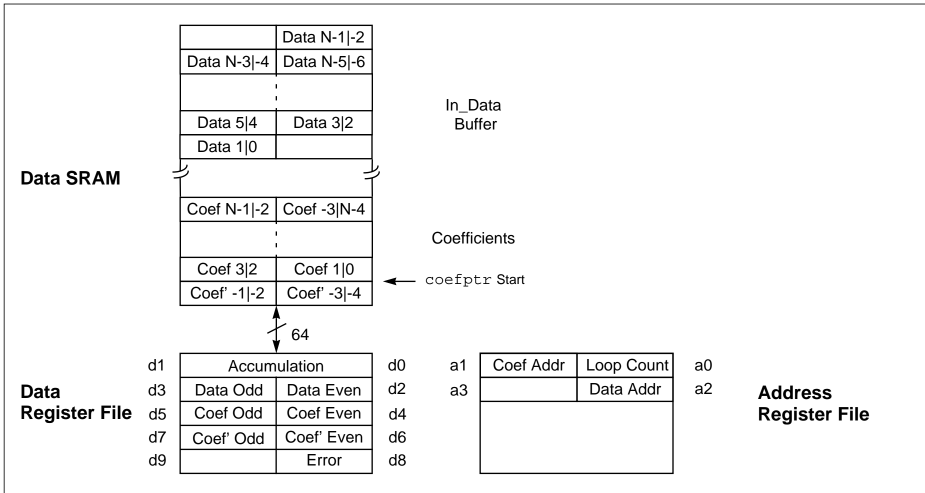

Figure 25
Register File Assignments and Data Memory Organization for the LMS Adaptive FIR Filter

6.3 Benchmark Results and Variations

The total instruction cycle count is in Table 6. The number of filter taps N is a multiple of four.

Table 6
LMS Adaptive Filter Instruction Count Benchmark Summary

Function	Prolog	Kernel	Epilog
LMS Adaptive Filter	5	4(N/4) + 2 = N + 2	2

A natural variation would be to use the 32-bit full-word precision including rounded 32-bit products. This would remove the two-to-one speed advantage of the dual operations and increase register and memory usages.

6.4 Programming Summary

In addition to the basic characteristics of the TriCore architecture used in the earlier benchmarks, this recursion on an FIR filter illustrates the following additional ones:

- Data Structures
  – The shared data structures for two separate computations, the filter and the coefficient updating, can be used recursively while still maintaining the efficiency of double-word load and stores and dual arithmetic operations.

```
_LMS:
            ;;prolog             ;five instruction cycles

   lea      a0,loopcount         ;load absolute the loop count N/4 minus one
   movh     d8,#0x3000           ;initialize odd error value,
   lea      a1,coefptr           ;and in parallel load the coefficient pointer
   mov      d8,#0x3000           ;initialize even error value,
   lea      a2,dataptr           ;and in parallel load the input data pointer
   mov      d0,#0            ;zero even portion of accumulator,
   ld.w     d3,[a2+]4            ;and in parallel load odd data0 and data1
   mov      d1,#0                ;zero odd portion of accumulator,
   ld.d     e4,[a1+]8            ;and in parallel load even and odd coef0-3

            ;;kernel             ;four instruction cycles

lmsloop:
   maddm.h  e0,e0,d3,d4ul,#1     ;add [(data0)*(coef0)+(data1)*(coef1)]
                                 ;to the accumulation,
   st.d     [a1]-16,e6           ;and in parallel store coef'0-3 for next loop
   maddrs.h d6,d4,d3,d8ul,#1     ;coef'0 = (coef0)*(error) and
                                 ;coef'1 = (coef1)*(error)
   ld.d     e2,[a2+]8            ;and in parallel load even and odd data2-5
   maddm.h  e0,e0,d2,d5ul,#1     ;add [(data2)*(coef2)+(data3)*(coef3)]
                                 ;to the accumulation,
   maddrs.h d7,d5,d2,d8ul,#1     ;coef'2 = (coef2)*(error) and
                                 ;coef'3 = (coef3)*(error)
   ld.d     e4,[a1+]8            ;and in parallel load even and odd coef4-7
loop        a0,lmsloop           ;repeat N/4 times

            ;;epilog             ;two instruction cycles

   st.d     [a1]-16,e6           ;store last coef' from loop
   st.h     filter_out,d1        ;store least-significant 16-bits of most-
                                 ;significant word of output data to least-
                                 ;significant 16 bits of memory or I/O
```

Figure 26
Program for LMS Adaptive 16-bit Real FIR Filter with N Taps

7 FFT Butterfly

The discrete Fast Fourier Transform (FFT) in it's many variations is fundamental to DSP because it is the link between time domain and frequency domain processing. It is a primary benchmark because of the repeated computation-intensive butterfly kernel and the complex addressing necessary for it's data structures. We examine here only the computations in the butterfly.

7.1 Introduction

The FFT butterfly is a computation of the form:

$$\text{data_x}(n)' = \text{data_x}(n) + \text{data_y}(n) \times \text{coef}(n)$$

and

$$\text{data_y}(n)' = \text{data_x}(n) - \text{data_y}(n) \times \text{coef}(n)$$

where the x and y data points and the coefficient in general are complex. This is for a two-data-point computation or radix-two and for decimation-in-time. Each point expands to four multiplies and four additions and/or subtractions:

$$\text{data_xr}(n)' = \text{data_xr}(n) + \text{data_yr}(n) \times \text{coefr}(n) - \text{data_yi}(n) \times \text{coefi}(n)$$
$$\text{data_xi}(n)' = \text{data_xi}(n) + \text{data_yi}(n) \times \text{coefr}(n) + \text{data_yr}(n) \times \text{coefi}(n)$$

and

$$\text{data_yr}(n)' = \text{data_xr}(n) - \text{data_yr}(n) \times \text{coefr}(n) + \text{data_yi}(n) \times \text{coefi}(n)$$
$$\text{data_yi}(n)' = \text{data_xi}(n) - \text{data_yi}(n) \times \text{coefr}(n) - \text{data_yr}(n) \times \text{coefi}(n)$$

It is shown as an in-place computation where the primed data points replace the original ones.

Common variations in the butterfly are for higher radices, decimation-in-frequency, not in-place computations, alterations for the particular pass through the data and various precision and overflow strategies.

7.2 Complex 16 x 16-bit Radix-2 Decimation-in-Time FFT Butterfly

This example is a general one where N points of data and the N/2 coefficients are both complex. The 16-bit data and coefficients are packed in a 32-bit full-word as shown in the data register file in Figure 27. The N/2 different x and y data points are in order in separate SRAM data buffers as are the coefficients. This permits simple linear addressing load and store operations within the butterfly kernel for a single pass through the total of N data points. In the complete multiple-pass FFT more elaborate addressing is required.

The in-place butterfly kernel in the program in Figure 28 completes the two loads, the eight multiply-add/subtracts and two stores in just five instruction cycles. The `maddsur.h` instruction and the following `st.w` take a total of two instruction cycles. The stored operand is delayed because of the multiplier-accumulator's latency.

The data is assumed to be scaled so there is no need for saturation or an overflow strategy. Rounding is used to minimized the effects of maintaining 16-bit precision throughout the computation. Figure 29 and Figure 30 show these arithmetic operations and operands for the first (`maddr.h`) and last (`msubadr.h`) instructions in the kernel.

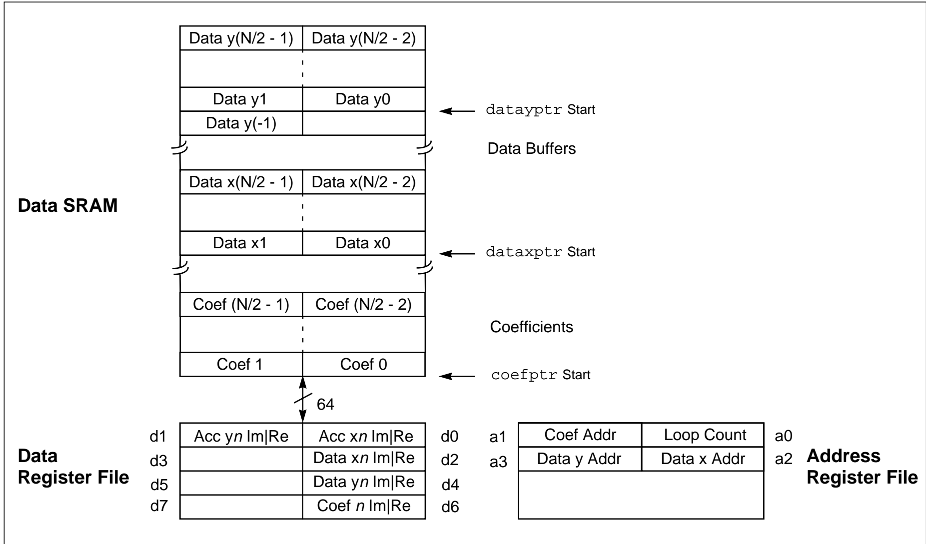

Figure 27
Register File Assignments and Data Memory Organization in the Complex 16 x 16-bit FFT with N Points and 16-bit Rounded Accumulation

```
_DIT:
              ;;prolog             ;seven instruction cycles

   lea        a0,loopcount         ;load absolute the loop count N/2 minus one
   lea        a1,coefptr           ;load absolute the coefficient pointer
   lea        a2,dataxptr          ;load absolute the Data x pointer
   lea        a3,datayptr          ;load absolute the Data y pointer
   ld.w       d6,[a1+]4            ;load coefr0 and coefi0
   ld.w       d4,[a3+]4            ;load datayr0 and datayi0
   ld.w       d2,[a2+]4            ;load dataxr0 and dataxi0

              ;;kernel             ;five instruction cycles

fftloop:
   maddr.h    d0,d2,d4,d6ll,#1     ;Acc x0 = dataxr0 + (datayr0)*(coefr0) |
                                   ;dataxi0 + (datayi0)*(coefr0)
   st.w       [a3]-8,d1            ;and in parallel store Data y(-1)' = Acc y(-1)
   maddsur.h d0,d0,d4,d6uu,#1      ;Acc x0 = accxr0 - (datayi0)*(coefi0) |
                                   ;accxi0 + (datayr0)*(coefi0)
   st.w       [a2]-4,d0            ;and in parallel store Data x0' = Acc x0'
   msubr.h    d1,d2,d4,d6ll,#1     ;Acc y0 = datayr0 - (datayr0)*(coefr0) |
                                   ;datayi0 - (datayi0)*(coefr0)
   ld.w       d2,[a2+]4            ;and in parallel load Data x1
   msubadr.h d1,d1,d4,d6uu,#1      ;Acc y0 = accyr0 + (datayi0)*(coefi0) |
                                   ;accyi0 - (datayr0)*(coefi0)
   ld.w       d4,[a3+]4            ;and in parallel load Data y1
loop          a0,fftloop           ;repeat N/2 times

              ;;epilog             ;one instruction cycle

   st.w       [a3]-8,d1            ;store Data y(N/2 - 1)' = Acc y(N/2 - 1)
```

Figure 28
Program for the Complex 16 x 16-bit Radix-2 Decimation-in-Time FFT Butterfly

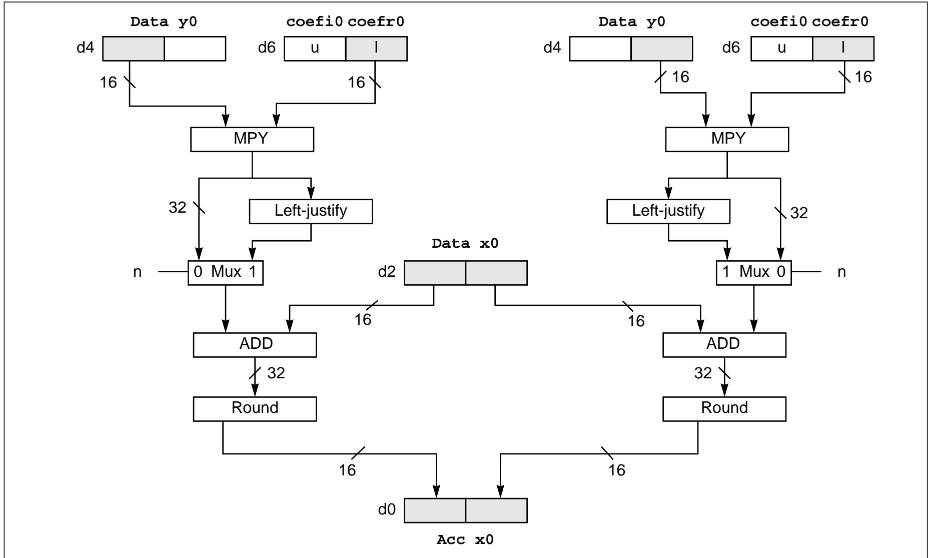

Figure 29
Arithmetic Operations in the Dual Multiply-Add (Lower, Lower) Instruction `maddr.h`

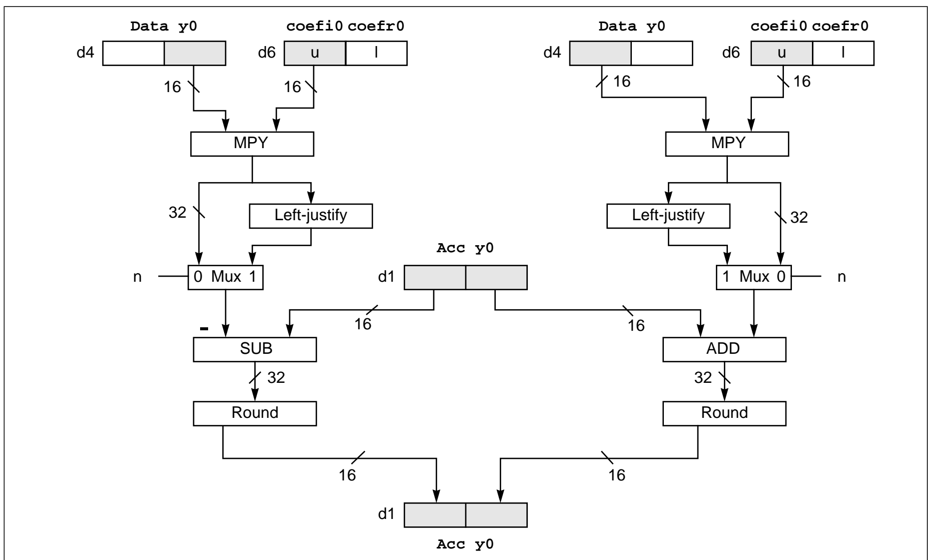

Figure 30
Operations in the Dual Multiply-Subtract-Add (Upper, Upper) Instruction `msubadr.h`

7.3 Benchmark Results and Variations

The total instruction cycle count is in Table 7 for one pass through the data, including the two for entering/exiting the loop. The total number of data points N in the pass is a multiple of two.

Table 7
FFT Butterfly Instruction Count Benchmark Summary

Function	Prolog	Kernel	Epilog
FFT Butterfly	7	5(N/2) + 2 = 5N/2 + 2	1

Many variations are possible with the FFT. This example illustrates the simplest kernel which fully utilizes the parallel arithmetic operations of the TriCore. Variations in arithmetic, data structures and re-use of pass-independent program code can all be assessed in comparison to this basic kernel.

7.4 Programming Summary

In addition to the basic characteristics of the TriCore architecture used in the earlier benchmarks, the FFT butterfly illustrates the following additional ones:

- Data Structures
  - How even when the computation is extensive (multiple operand and operation combinations), load and store transfers can be embedded to permit in-place computation to minimize the need for large data memories. For the most common of DSP precisions, 16 bits, the loads and stores for the FFT are only single word.

Infineon goes for Business Excellence

"Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.
Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction."

Dr. Ulrich Schumacher

http://www.infineon.com

Published by Infineon Technologies AG