The Residu Implementation Using the TMS320C6000

C6000 Applications

Digital Signal Processing Solutions

ABSTRACT

This application report describes the implementation of the Residu function on the TMS320C6000™(C6000™). The Residu function is a basic function in the GSM EFR and G.729 and other voice coding schemes. G.729 is an algorithm for the coding of speech signals at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP) while GSM EFR uses the ACELP coding scheme. The work presented in this report uses the Residu function as an example code to demonstrate that the C6000 compiler tools can achieve competitive performance compared to the estimated hand coded assembly.

Contents

List of Figures

List of Tables

1 Introduction

This application report describes the implementation of the Residu function on the TMS320C6000 (C6000). The Residu function is a basic function in the GSM EFR and G.729 and other voice coding schemes. G.729 uses the conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP) while GSM EFR uses algebraic-code-excited linear-prediction (ACELP). This report will briefly describe G.729, GSM EFR, and C6000 compiler tools. This is followed by a discussion on the different optimization methods. Lastly, some benchmarks are presented.

1.1 G.729

G.729 is defined by the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T) [2].

G.729 uses the Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP) algorithm, which is an analysis-by-synthesis algorithm and belongs to the class of speech coding algorithms knows as Code Excited Linear Prediction (CELP).

For every 10 ms speech frame, the speech signal is analyzed to extract the parameters of CELP model. Each speech frame is equally divided into 2 subframes. Most parameters are determined per subframe of 5 ms (40 samples) each. The residual signal is used to find the target signal by filtering the linear-prediction (LP) residual through a weighted synthesis filter. These parameters are quantized into 80 bits, resulting in a transmission rate of 8 kbits/s. The G.729 decoder converts the digitized signal back to an analog signal using a similar approach.

1.2 GSM EFR

The Global System for Mobile Communication enhanced full rate (GSM EFR) standard is defined by the European Telecommunication Standards Institute (ETSI).

GSM EFR uses the Algebraic-Code-Excited Linear-Prediction (ACELP) coding scheme. The GSM EFR encoder is based on Code Excited Linear Prediction (CELP) which is an analysis-by-synthesis algorithm.

For every 20 ms speech frame, the speech signal is analyzed to extract the parameters of the CELP model. Each speech frame is equally divided into 4 subframes of 5 ms each (40 samples) at the sampling frequency of 8000 sample/s. In each subframe, the residual signal is used to find the target signal by filtering the LP residual through the weighted synthesis filter. The GSM EFR decoder decodes and synthesizes the speech through the same parameters as the CELP model which does the encoding.

1.3 The Residu Function

The Residu function is used to find the residual signal r(n) which is needed for finding the target vector for adaptive codebook search in both G.729 and GSM EFR speech coders. The LP residual signal is filtered through the combination of synthesis filter 1/ $\hat{A}(z)$ and the weighting filter $A(z/\gamma_1)$ / $A(z/\gamma_2)$. After determining the excitation for the subframe, the initial states of these filters are updated by filtering the difference between the LP residual and excitation. The LP residual signal is also used in the adaptive codebook search to extend the past excitation buffer. The LP residual is given by

$$r(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i * s(n - i), \qquad n = 0, \ldots, 39. \tag{1}$$

where s(n) is the pre-processed speech signal, $\hat{a}_i$, i = 1,...,10, are the (quantized) Linear Prediction (LP) coefficients. In both G.729 and GSM EFR applications, the Residu function is represented in Q12 notation; that is, the decimal point is placed between bits 11 and 12. This notation allows handling the accuracy of speech information for fixed-point digital signal processors. For example, “0x1000” represents the number 1.0. Therefore equation 1 is modified as follows

$$r(n) = s(n) * a_0 + \sum_{i=1}^{10} \hat{a}_i * * s(n - 1) \tag{2}$$

where a_0 is equal to 0x1000, $\hat{a}_i = \hat{a}_i \times a_0$.

1.4 The TMS320C6000 DSP

The TMS320C6000 (C6000) generation of digital signal processors which is part of the TMS320 family of digital signal processors (DSPs) has the VelociTI architecture [5]. The VelociTI architecture is a high-performance, advanced, very-long-instruction-word (VLIW) architecture that is capable of many applications such as multi-channel, multi-function, and performance-driven applications. VelociTI, together with the development tool set and evaluation tools, provides faster development time and higher performance for embedded DSP applications through increased instruction-level parallelism. The TMS320C6000 include TMS320C62x, TMS320C64x and TMS320C67x devices. The TMS320C62x (C62x) and TMS320C64x (C64x) devices are fixed-point DSPs and the TMS320C67x (C67x) devices are floating-point DSPs.

The C6000 is supported by an optimized C compiler tool [4]. The C compiler accepts C source code and produces C6000 assembly language source code. The C compiler includes a shell program, an optimizer, and other utilities. The optimizer takes advantage of the features specific to the C6000 architecture. General optimizations can be applied to any C code [4]. This application note will demonstrate that the C6000 compiler tools can achieve a competitive performance compared to the estimated hand coded assembly.

The Residu function is well suited for DSPs such as the C6000. This application note presents the competitive performance that can be achieved from the C6000 compiler tools as against using hand coded assembly.

2 Optimization Methods

The key to implementing applications on the C6000 is to take advantage of the processor’s full speed. There are many methods of code development flow to increase performance. This report will discuss 3 methods using the C6000 software development flow, namely, natural C, optimized C and partitioned linear assembly.

2.1 Natural C

The natural C optimization method is the 1st step in the code development flow. The key to develop the natural C code of the Residu function is using the out of the box Residu C code without any modification i.e. no modification is required. Then, substitute the math functions with intrinsics [6]. The C6000 compiler provides intrinsics that are not easily expressed in C code. These intrinsics are mapped directly to inline C6000 instructions to optimize the C code quickly. For example, L_add math function can be expressed in C code by writing a multi-cycle function. This complicated code can be replaced by the _sadd() intrinsic, which results in a single C6000 instruction.

The Residu natural C code is given in Figure 1.

```
#define m  10     /* m = LPC order == 10  */
typedef short Word16;
typedef int   Word32;
void Residu (
    Word16 a[], /* (i)     : prediction coefficients      */
    Word16 x[], /* (i)     : speech signal                */
    Word16  y[], /* (o)     : residual signal              */
    Word16 lg   /* (i )     : size of filtering           */
)
{
    Word16i, j;
    Word32 s;

    for (i = 0;i < lg; i++)
   {
        s = L_mult (x[i], a[0]);
        for (j = 1; j <= m; j++)
        {
            s = L_mac (s, a[j], xi[ – j]);
        }
        s = L_shl (s, 3);
        y[i] = round (s);
    }
    return;
}
```

Figure 1. The Residu Natural C Code

All ETSI standard code performs data arithmetic with clearly defined math functions. These provide a standard way of completely describing the functionality for bit exact behavior. As such, these are ultimately intended to be replaced by inline versions of specific machines instructions as most involve saturation which is not easily represented in ANSI C. For this reason, all of the math functions used in the Residu function have been replaced with the supporting C6000 intrinsics, shown in Figure 2.

```
#define L_add(a,b)   (_sadd((a),(b)))
#define L_mult(a,b)  (_smpy((a),(b)))
#define extract_h(a) (_extu(a,0,16))//((unsigned)(a)>>16)
#define L_shl(a,b)   ((b) < 0 ? (a) >> (– b) : sshl((a),(b)))
#define round(a)     ((unsigned)(_sadd((a),0x8000))>>16)
```

Figure 2. The Math Function Used in the Residu Function

A complete list of all ETSI math functions are available in one of the TSM320 DSP Designer's Notebooks: ETSI Math Operations in C for the C62xx [6].

The natural C code is compiled using the cl6x shell program using -pm -op2 -o3 -oi0 -mh -mi -k -q -mw options. The program level optimization options -pm -op2 has the ability to automatically extract the information from the calling program i.e. the compiler can automatically extract all of the information regarding potential pointer aliasing, loop count, min/max trip count, alignment and cycle per iteration with program level optimization. Furthermore, it can perform program-level optimization by adding the -o3 option. The combination of -pm and -o3 options enable the compiler to see the entire program that performs several optimizations beyond the file-level optimization. With -oi option, the optimizer automatically inlines small functions when it is invoked with the -o3 option. The -oi0 option disables all size-controlled inlines. The -mh option indicates that load instructions can read an unlimited number of bytes past the beginning or end of a buffer. The -mi option ensures the Residu code is never interrupted. The -q option suppresses banners and progress messages from all the tools. The -mw option embeds software pipelined loop information (feedback) in the .asm file, as shown in Figure 3. In order to review the feedback, the k option is required to keep the natural C assembly language (.asm) file.

The software pipeline information in Figure 3 shows the inner loop of the Residu function. The compiler tries to identify what the loop counter value is, and whether it is a multiple of some number (has a known maximum trip count factor). The feedback shows the same number (40) for the known min/max trip counts and the max trip count factor. This indicates a constant value for the trip count argument, i.e. the compiler can be more aggressive to achieve a better performance. Although the feedback shows a value of 1 in loop carried dependency bound, the compiler can still maximize the loop pipeline ability. The feedback also shows the inner loop completely unrolled into the outer loop. The combined loop requires 12 load/store (assuming the coefficients were preloaded before the kernel started), 11 multiply and 11 saturated add operations. The resource bounds indicate a value of 6. The result of the software pipeline schedule shows a 6-cycle loop with a trip count of 40. This indicates that the compiler has successfully software pipelined the loop. The result shown above were obtained by running the Residu natural C on C6000 version 4.0 of code generation tools (most recent version of the C6000 software development tools.)

The TMS320C6000 optimizing C compiler user's guide [4] has documented all of the compiler options. The C6000 compiler optimization tutorial shows detailed information as shown in the following web page: http://www.ti.com/sc/c6000compiler. Also, the TMS320C6000 optimizing C compiler user's guide [4] has documented all of the compiler options.

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 40
;*      Known Maximum Trip Count         : 40
;*      Known Max Trip Count Factor      : 40
;*      Loop Carried Dependency Bound(^) : 1
;*      Unpartitioned Resource Bound     : 6
;*      Partitioned Resource Bound(*)    : 6
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     5        6*
;*      .S units                     1        2
;*      .D units                     6*       6*
;*      .M units                     6*       5
;*      .X cross paths               4        3
;*      .T address paths             6*       6*
;*      Long read paths              1        0
;*      Long write paths             0    0
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          1        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             4    4
;*      Bound(.L .S .D .LS .LSD)     5    5
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 3
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 2
;*
;*      Minimum required memory pad : 6 bytes
;*
;*      Minimum safe trip count     : 1
```

Figure 3. Software Pipeline Information: Natural C

2.2 Optimized C – Removing the Nested Loop

Two basic optimized C techniques include optimizing data flow bandwidth (uses word access instead of two short data accesses) to ease the bottleneck of data accesses and unrolling the loop to increase the number of instructions available in order to execute as many instructions in parallel as possible to improve performance.

To utilize the above techniques (optimizing data flow and unrolling the inner loop), the optimized C code has combined two loops into one loop, i.e. it totally unrolled the inner loop. To overcome the uneven partition of .L and .M units in A-side and B-side (shown on the software pipeline information of natural C code), the optimized C processes two output samples at a time. In this way, the inner loop has a loop count of 20. Figure 4 shows the combined modified loop of the optimized C code. This loop includes the totally unrolled inner loop and the outer loop. This loop requires 8 loads/stores, 22 multiplies and 22 saturated adds with a trip count of 20. The Residu optimized C code is shown in Appendix A.

Figure 5 shows software pipeline information of the optimized C. This information reveals the expected numbers that shows even distribution of the .L and .M units so that the .L's occur the same number of times in both A-side and B-side, and the same for the .M units (a total of 11 times on both A and B sides). However, the cross path becomes a bottleneck resource problem on the B-side. This causes the compiler to schedule an iteration interval of 13 cycles. Even though this loop throughput is actually lower (6.5 cycles per iteration) than the Natural C, there is no outer loop overhead and thus the overall cycle count is actually lower.

```
for ( i = 0, j = 0; j <   lg; j+=2, i++)
  {
    s1 =  L_mult((x[i]>>16), a0);       /* a0 * x[i+1]  */
    s0 =  L_mult((x[i]),     a0);       /* a0 * x[i]    */
    s1 =  L_mac(s1, a0>>16, x[    i]);  /* a1 * x[i]    */
    s0 =  L_mac(s0, a0>>16, x[i-1]>>16); /* a1 * x[i-1]  */
    s1 =  L_mac(s1, a1,     x[i-1]>>16); /* a2 * x[i-1]  */
    s0 =  L_mac(s0, a1,     x[i-1]);     /* a2 * x[i-2]  */
    s1 =  L_mac(s1, a1>>16, x[i-1]);     /* a3 * x[i-2]  */
    s0 =  L_mac(s0, a1>>16, x[i-2]>>16); /* a3 * x[i-3]  */
    s1 =  L_mac(s1, a2,     x[i-2]>>16); /* a4 * x[i-3]  */
    s0 =  L_mac(s0, a2,     x[i-2]);     /* a4 * x[i-4]  */
    s1 =  L_mac(s1, a2>>16, x[i-2]);     /* a5 * x[i-4]  */
    s0 =  L_mac(s0, a2>>16, x[i-3]>>16); /* a5 * x[i-5]  */
    s1 =  L_mac(s1, a3,     x[i-3]>>16); /* a6 * x[i-5]  */
    s0 =  L_mac(s0, a3,     x[i-3]);     /* a6 * x[i-6]  */
    s1 =  L_mac(s1, a3>>16, x[i-3]);     /* a7 * x[i-6]  */
    s0 =  L_mac(s0, a3>>16, x[i-4]>>16); /* a7 * x[i-7]  */
    s1 =  L_mac(s1, a4,     x[i-4]>>16); /* a8 * x[i-7]  */
    s0 =  L_mac(s0, a4,     x[i-4]);     /* a8 * x[i-8]  */
    s1 =  L_mac(s1, a4>>16, x[i-4]);     /* a9 * x[i-8]  */
    s0 =  L_mac(s0, a4>>16, x[i-5]>>16); /* a9 * x[i-9]  */
    s1 =  L_mac(s1, a5,     x[i-5]>>16); /* a10 * x[i-9] */
    s0 =  L_mac(s0, a5,     x[i-5]);     /* a10 * x[i-10]*/
    s1 =  L_add(s1, 4096);
    s0 =  L_add(s0, 4096);
    s1 =  L_shl(s1, 3);
    s0 =  L_shl(s0, 3);
    y[j] =   (short)(s0 >>16);
    y[j+1] = (short)(s1 >>16);
  }
```

Figure 4. The Combined Loop of Residu Optimized C Code

```
;*----------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 20
;*      Known Maximum Trip Count         : 20
;*      Known Max Trip Count Factor      : 20
;*      Loop Carried Dependency Bound(^) : 2
;*     Unpartitioned Resource Bound     : 11
;*      Partitioned Resource Bound(*)    : 13
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                    11     11
;*      .S units                     2        3
;*      .D units                     8        0
;*      .M units                    11     11
;*      .X cross paths              11       13*
;*      .T address paths             4      4
;*      Long read paths              1      1
;*      Long write paths             0      0
;*      Logical  ops (.LS)           0        2  (.L or .S unit)
;*      Addition ops (.LSD)          0        1  (.L or .S or .D unit)
;*      Bound(.L .S .LS)             7        8
;*      Bound(.L .S .D .LS .LSD)     7        6
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 13 Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 1
;*
;*      Minimum required memory pad : 4 bytes
;*
;*      Minimum safe trip count     : 1
;*----------------------------------------------------------------------*
```

Figure 5. Software Pipeline Information: Optimized C

2.3 Partitioned Linear Assembly

Writing a linear assembly code is just like writing a C code but in assembly language. Linear assembly is quite similar to the regular C6000 assembly code. Linear assembly doesn't need to specify the parallel instructions, pipeline latency, or register usage. It also does not need to indicate which functional unit is being used. Partitioned linear assembly adds partitioning information to the linear assembly.

Like the natural C and optimized C, the linear assembly is coded with a totally unrolled inner loop. Based on the software pipeline information of optimized C code, the goal for the partitioned linear assembly is essentially to solve the cross path problem. To overcome the cross path problem, the linear assembly adds partitioning information for each input and coefficient. The Residu partitioned linear assembly code is shown in Appendix B. Figure 6 shows the software pipeline information of partitioned linear assembly. This information shows all of the resources bounded with a maximum number of 11 cycles.

Furthermore, the assembly optimizer schedules 11 cycles for the iteration interval. This shows the partitioned linear assembly has achieved the goal, i.e. 11 cycle loop with loop count of 20.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : LOOP
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*     Unpartitioned Resource Bound     : 11
;*      Partitioned Resource Bound(*)    : 11
;*      Resource Partition:
;*                                 A-side   B-side
;*      .L units                     11*      11*
;*      .S units                      3        2
;*      .D units                      2        6
;*      .M units                     11*      11*
;*      .X cross paths               10       11*
;*      .T address paths              7        1
;*      Long read paths               1    1
;*      Long write paths              0    0
;*      Logical  ops (.LS)            0    0     (.L or .S unit)
;*      Addition ops (.LSD)           0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)              7    7
;*      Bound(.L .S .D .LS .LSD)      6        7
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 11 Schedule found with 3 iterations in parallel
;*      done
;*
;*      Collapsed epilog stages     : 2
;*      Collapsed prolog stages     : 2
;*      Minimum required memory pad : 8 bytes
;*
;*      Minimum safe trip count     : 1
;*----------------------------------------------------------------------------*
```

Figure 6. Software Pipeline Information: Partitioned Linear Assembly

3 Benchmark Comparison

3.1 Performance Comparison of Different Optimization Methods

The benchmark comparison for the above implementations, namely: natural C, optimized C and partitioned linear assembly codes described in Chapter 2 are summarized in Table 1. The estimated hand assembly cycle count is based on the ideal case with minimum overhead cycle count. The completed optimized C code and partitioned linear assembly are shown in Appendices A and B. All of the benchmarks were obtained using C6000 code generation tools version 4.00. The options used for the compiler were: -pm -q -op2 -o3 -oi0 -mh -mi -mw -k. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information about the compiler tools.

Table 1. Benchmarks for Residu

Residu	Nat. C	Opt. C	par. Lin. asm
cycle counts (cycles)	367	312	285

3.2 Performance Comparison Within the TMS320 Family

The performance comparison with TMS320C54x is summarized in Table 2.

Table 2. Comparison Within the TMS320 Family

		C6000		
Residu	C54x (hand asm)	Nat. C	Opt. C	par. Lin. asm
cycle counts (cycles)	541	367	312	285

4 Conclusion

In this report we have presented 3 software development flow methods (natural C, optimized C, partitioned linear assembly) for the Residu function on the TMS320C6000. The natural C, optimized C, and partitioned linear assembly code were modified and validated.

Natural C has several advantages over hand coded assembly:

- It is easier to write (since it is the out of the box C code) and debug than hand assembly.
- The TMS320C6000 compiler is able to generate very efficient code that is good enough to be equivalent to one version of the assembly code, i.e. the high performance assembly code.

Optimized C has the following advantage over hand coded assembly:

- It removes the outer loop overhead by completely unrolling the inner loop.

Partitioned linear assembly has several advantages over hand coded assembly:

- It provides flexibility of hand-coded assembly without worring about pipelining parallelism, or register allocation.
- It can improve partitioning of loops when necessary.

Speech applications exhibit a high degree of parallelism that can be exploited by VLIW architectures such as the C6000. This report has demonstrated that C6000 tools (the partitioned linear assembly) is able to achieve competitive performance w.r.t. estimated hand scheduled assembly. Furthmore, the partitioned linear assembly requires minimum development time compared to hand-coded assembly, and the performance approaches the best that can be achieved by hand.

5 References

1. [ITU-T Recommendation G.729 – CS-ACELPD, March 1996.
2. [ETSI SMG2 ITU-T Recommendation GSM 06.60 (Enhanced full rate speech transcoding), January 1996.
3. Texas Instruments, *TMS320C62x/C67x Programmer's Guide*, Texas Instruments, Inc., Literature number SPRU198, Dallas, Texas, April 1998.
4. Texas Instruments, *TMS320C6000 Optimizing C Compiler User's Guide*, Texas Instruments, Inc., Literature number SPRU187, Dallas, Texas, April 1998.
5. Texas Instruments, *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, Texas Instruments, Inc., Literature number SPRU189, Dallas, Texas, April 1998.
6. Texas Instruments, *TMS320 DSP Designer's Notebook: ETSI Math Operations in C for the C62xx*, Texas Instruments, Inc., Literature number dnp87, Dallas, Texas, April 1998.

Appendix A The Residu Optimized C Code

```
#define DWORD_ALIGNED(x) (_nassert(((int)(x) & 0x7) == 0))

typedef short Word16;
typedef int   Word32;
#define lg 40
#define L_mult(a,b)  (_smpy((a),(b)))
#define L_mac(a,b,c)    (_sadd((a),_smpy((b),(c))))
#define L_add(a,b)      (_sadd(a,b))
#define L_shl(a,b)      ((b<0) ? (a) >> (-b) : _sshl(a,b))
#define round(a)        ((unsigned)(_sadd(a,0x8000L))>>16)

void Residu_co(
  const Word32 a[], /* (i) Q12: prediction coefficients             */
  const Word32 x[], /* (i)    : speech (values x[-m..-1] are needed */
  Word16 y[],       /* (o)    : residual signal                     */
  Word16 lg         /* (i)    : size of filtering                   */
)
{
  Word16 i, j;
  Word32 s0, s1;
  Word32 a0, a1, a2, a3, a4, a5;

  DWORD_ALIGNED(a);
  DWORD_ALIGNED(x);

  a0 = a[0];
  a1 = a[1];
  a2 = a[2];
  a3 = a[3];
  a4 = a[4];
  a5 = a[5];

  for (i = 0, j = 0; j < lg; j+=2, i++)
  {
    s1 = L_mult((x[i]>>16), a0)       ; /* a0 * x[i+1]  */
    s0 = L_mult((x[i]),     a0)       ; /* a0 * x[i]    */
    s1 = L_mac(s1, a0>>16, x[i])      ; /* a1 * x[i]    */
    s0 = L_mac(s0, a0>>16, x[i-1]>>16); /* a1 * x[i-1]  */
    s1 = L_mac(s1, a1,     x[i-1]>>16); /* a2 * x[i-1]  */
    s0 = L_mac(s0, a1,     x[i-1]);     /* a2 * x[i-2]  */
    s1 = L_mac(s1, a1>>16, x[i-1]);     /* a3 * x[i-2]  */
    s0 = L_mac(s0, a1>>16, x[i-2]>>16); /* a3 * x[i-3]  */
    s1 = L_mac(s1, a2,     x[i-2]>>16); /* a4 * x[i-3]  */
    s0 = L_mac(s0, a2,     x[i-2]);     /* a4 * x[i-4]  */
    s1 = L_mac(s1, a2>>16, x[i-2]);     /* a5 * x[i-4]  */
    s0 = L_mac(s0, a2>>16, x[i-3]>>16); /* a5 * x[i-5]  */
    s1 = L_mac(s1, a3,     x[i-3]>>16); /* a6 * x[i-5]  */
    s0 = L_mac(s0, a3,     x[i-3]);     /* a6 * x[i-6]  */
    s1 = L_mac(s1, a3>>16, x[i-3]);     /* a7 * x[i-6]  */
    s0 = L_mac(s0, a3>>16, x[i-4]>>16); /* a7 * x[i-7]  */
    s1 = L_mac(s1, a4,     x[i-4]>>16); /* a8 * x[i-7]  */
```

```
    s0 = L_mac(s0, a4,     x[i–4]);     /* a8 * x[i–8]  */
    s1 = L_mac(s1, a4>>16, x[i–4]);     /* a9 * x[i–8]  */
    s0 = L_mac(s0, a4>>16, x[i–5]>>16); /* a9 * x[i–9]  */
    s1 = L_mac(s1, a5,     x[i–5]>>16); /* a10 * x[i–9]  */
    s0 = L_mac(s0, a5,     x[i–5]);     /* a10 * x[i–10] */
    s1 = L_add(s1, 4096);
    s0 = L_add(s0, 4096);
    s1 = L_shl(s1, 3);
    s0 = L_shl(s0, 3);
    y[j] =   (short)(s0 >>16);
    y[j+1] = (short)(s1 >>16);
  }
  return;
}
```

Appendix B The Residu Partitioned Linear Assembly Code

```
_Residu_psa: .cproc A_a_ptr, B_x_ptr, A_y_ptr, B_lg ;arg_list

        .no_mdep

        .reg    A_a_0       ; coef: a[0]
        .reg    A_x_ba      ; input: x[i+1,i]
        .reg    A_x_98      ; input: x[i–1,i–2]
        .reg    A_x_76      ; input: x[i–3,i–4]
        .reg    A_x_54      ; input: x[i–5,i–6]
        .reg    A_x_32      ; input: x[i–7,i–8]
        .reg    A_x_10      ; input: x[i–9,i–10]
        .reg    A_p00       ; prod: x[i  ]*a[0]
        .reg    A_p01       ; prod: x[i–1]*a[1]
        .reg    A_p02       ; prod: x[i–2]*a[2]
        .reg    A_p03       ; prod: x[i–3]*a[3]
        .reg    A_p04       ; prod: x[i–4]*a[4]
        .reg    A_p05       ; prod: x[i–5]*a[5]
        .reg    A_p06       ; prod: x[i–6]*a[6]
        .reg    A_p07       ; prod: x[i–7]*a[7]
        .reg    A_p08       ; prod: x[i–8]*a[8]
        .reg    A_p09       ; prod: x[i–9]*a[9]
        .reg    A_p0a       ; prod: x[i–10]*a[10]
        .reg    A_s0        ; sum0
        .reg    A_y0        ; y0
        .reg    B_i         ; outloop counter
        .reg    B_a_0       ; coef: a[0]
        .reg    B_a_10      ; coef: a[1,0]
        .reg    B_a_32      ; coef: a[3,2]
        .reg    B_a_54      ; coef: a[5,4]
        .reg    B_a_76      ; coef: a[7,6]
        .reg    B_a_98      ; coef: a[9,8]
        .reg    B_a_ba      ; coef: a[11,10]
        .reg    B_p10       ; prod: x[i+1]*a[0]
        .reg    B_p11       ; prod: x[i  ]*a[1]
        .reg    B_p12       ; prod: x[i–1]*a[2]
        .reg    B_p13       ; prod: x[i–2]*a[3]
        .reg    B_p14       ; prod: x[i–3]*a[4]
        .reg    B_p15       ; prod: x[i–4]*a[5]
        .reg    B_p16       ; prod: x[i–5]*a[6]
        .reg    B_p17       ; prod: x[i–6]*a[7]
        .reg    B_p18       ; prod: x[i–7]*a[8]
        .reg    B_p19       ; prod: x[i–8]*a[9]
        .reg    B_p1a       ; prod: x[i–9]*a[10]
        .reg    B_s1        ; sum1
        .reg    B_y1        ; y1

       LDW    .D1T2  *A_a_ptr++,B_a_10  ; load a[1] & a[0], a[0] = 4096
       LDW    .D1T2  *A_a_ptr++,B_a_32  ; load a[3] & a[2]
       LDW    .D1T2  *A_a_ptr++,B_a_54  ; load a[5] & a[4]
       LDW    .D1T2  *A_a_ptr++,B_a_76  ; load a[7] & a[6]
       LDW    .D1T2  *A_a_ptr++,B_a_98  ; load a[9] & a[8]
```

```
        LDH    .D1T2  *A_a_ptr,B_a_ba    ; load a[11] & a[10]
        SHR    .2  B_lg,1,B_i            ; outloop loop counter, lg/2
        MVK    .1  4096,A_a_0            ; a[0] = 4096
        MVK    .2  4096,B_a_0            ; a[0] = 4096

LOOP:
        LDW    .D2T1  *B_x_ptr--,A_x_ba   ; load x[ 1] & x[ 0]
        LDW    .D2T1  *B_x_ptr--,A_x_98   ; load x[-1] & x[-2]
        LDW    .D2T1  *B_x_ptr--,A_x_76   ; load x[-3] & x[-4]
        LDW    .D2T1  *B_x_ptr--,A_x_54   ; load x[-5] & x[-6]
        LDW    .D2T1  *B_x_ptr--,A_x_32   ; load x[-7] & x[-8]
        LDW    .D2T1  *B_x_ptr++[6],A_x_10; load x[-9] & x[-10]

        SMPY   .1  A_x_ba,A_a_0,A_p00    ; smpy(x[i  ],a[0])
        SMPYH  .1X A_x_98,B_a_10,A_p01   ; smpy(x[i-1],a[1])
        SMPY   .1X A_x_98,B_a_32,A_p02   ; smpy(x[i-2],a[2])
        SMPYH  .1X A_x_76,B_a_32,A_p03   ; smpy(x[i-3],a[3])
        SMPY   .1X A_x_76,B_a_54,A_p04   ; smpy(x[i-4],a[4])
        SMPYH  .1X A_x_54,B_a_54,A_p05   ; smpy(x[i-5],a[5])
        SMPY   .1X A_x_54,B_a_76,A_p06   ; smpy(x[i-6],a[6])
        SMPYH  .1X A_x_32,B_a_76,A_p07   ; smpy(x[i-7],a[7])
        SMPY   .1X A_x_32,B_a_98,A_p08   ; smpy(x[i-8],a[8])
        SMPYH  .1X A_x_10,B_a_98,A_p09   ; smpy(x[i-9],a[9])
        SMPY   .1X A_x_10,B_a_ba,A_p0a   ; smpy(x[i-10],a[10])

        SMPYLH .2X B_a_10,A_x_ba,B_p10   ; smpy(x[i+1],a[0])
        SMPYHL .2X B_a_10,A_x_ba,B_p11   ; smpy(x[i  ],a[1])
        SMPYLH .2X B_a_32,A_x_98,B_p12   ; smpy(x[i-1],a[2])
        SMPYHL .2X B_a_32,A_x_98,B_p13   ; smpy(x[i-2],a[3])
        SMPYLH .2X B_a_54,A_x_76,B_p14   ; smpy(x[i-3],a[4])
        SMPYHL .2X B_a_54,A_x_76,B_p15   ; smpy(x[i-4],a[5])
        SMPYLH .2X B_a_76,A_x_54,B_p16   ; smpy(x[i-5],a[6])
        SMPYHL .2X B_a_76,A_x_54,B_p17   ; smpy(x[i-6],a[7])
        SMPYLH .2X B_a_98,A_x_32,B_p18   ; smpy(x[i-7],a[8])
        SMPYHL .2X B_a_98,A_x_32,B_p19   ; smpy(x[i-8],a[9])
        SMPYLH .2X B_a_ba,A_x_10,B_p1a   ; smpy(x[i-9],a[10])

        SADD   .1  A_p00,A_p01,A_s0  ; s0 = sadd(smpy(x[-1], a[0]),
                                     ;           smpy(x[-1], a[1]))
        SADD   .1  A_s0,A_p02,A_s0  ; s0 = sadd(s0,smpy(x[-2],a[2]))

        SADD   .1  A_s0,A_p03,A_s0  ; s0 = sadd(s0,smpy(x[-3],a[3]))
        SADD   .1  A_s0,A_p04,A_s0  ; s0 = sadd(s0,smpy(x[-4],a[4]))
        SADD   .1  A_s0,A_p05,A_s0  ; s0 = sadd(s0,smpy(x[-5],a[5]))
        SADD   .1  A_s0,A_p06,A_s0  ; s0 = sadd(s0,smpy(x[-6],a[6]))
        SADD   .1  A_s0,A_p07,A_s0  ; s0 = sadd(s0,smpy(x[-7],a[7]))
        SADD   .1  A_s0,A_p08,A_s0  ; s0 = sadd(s0,smpy(x[-8],a[8]))
        SADD   .1  A_s0,A_p09,A_s0  ; s0 = sadd(s0,smpy(x[-9],a[9]))
        SADD   .1  A_s0,A_p0a,A_s0  ; s0 = sadd(s0,smpy(x[-10],a[10]))
```

```
       SADD   .2  B_p10,B_p11,B_s1 ; s1 = sadd(smpy(x[1],a[0]),
                                   ;           smpy(x[0],a[1]))
       SADD   .2  B_s1,B_p12,B_s1  ; s1 = sadd(s1,smpy(x[-1],a[2]))
       SADD   .2  B_s1,B_p13,B_s1  ; s1 = sadd(s1,smpy(x[-2],a[3]))
       SADD   .2  B_s1,B_p14,B_s1  ; s1 = sadd(s1,smpy(x[-3],a[4]))
       SADD   .2  B_s1,B_p15,B_s1  ; s1 = sadd(s1,smpy(x[-4],a[5]))
       SADD   .2  B_s1,B_p16,B_s1  ; s1 = sadd(s1,smpy(x[-5],a[6]))
       SADD   .2  B_s1,B_p17,B_s1  ; s1 = sadd(s1,smpy(x[-6],a[7]))
       SADD   .2  B_s1,B_p18,B_s1  ; s1 = sadd(s1,smpy(x[-7],a[8]))
       SADD   .2  B_s1,B_p19,B_s1  ; s1 = sadd(s1,smpy(x[-8],a[9]))
       SADD   .2  B_s1,B_p1a,B_s1  ; s1 = sadd(s1,smpy(x[-9],a[10]))

       MVk    .1  4096,A_a_0           ; a[0] = 4096

       SADD   .1  A_s0,A_a_0,A_y0      ; s0 = sadd(s0,4096)
       SADD   .2  B_s1,B_a_0,B_y1      ; s1 = sadd(s1,4096)
       SSHL   .1  A_y0,3,A_y0          ; s0 = L_shl(s0,3)
       SSHL   .2  B_y1,3,B_y1          ; s1 = L_shl(s1,3)

       SHR    .1  A_y0,16,A_y0         ; y[0] = shr(s0,16)
       SHR    .2  B_y1,16,B_y1         ; y[1] = shr(s1,16)
       STH    .1  A_y0,*A_y_ptr++      ; store y[0]
       STH    .D1T2  B_y1,*A_y_ptr++      ; store y[1]

 [B_i]SUB    .2  B_i,1,B_i            ; decrement loop counter
 [B_i]B          LOOP                 ; branch to the loop

           .endproc
```

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.