Application Report
SPRA481

Software Development Techniques for the TMS320C6201 DSP

Richard Scales

Abstract

The advancements in performance and flexibility of modern digital signal processor (DSP) devices is clearly demonstrated in the release of the new TMS320C62xx family of DSPs from Texas Instruments. The TMS320C62xx is a high-performance Very Long Instruction Word (VLIW) DSP based on TIs own Veloci (TI™) architecture. The need to support such an advanced device has fueled a need for DSP development software that is equal to the task when designing high performance DSP systems. In order to extract the optimum performance from the TMS320C62xx devices, it is necessary to use high level language (HLL) compilers that perform beyond the currently expected norm in all of the following areas:

❑ Code size, to allow greater use of on-chip memory

❑ Execution efficiency via algorithmic and functional optimizations

❑ Data throughput

❑ Utilization of on-chip features and functionality

Contents

Figures

Tables

Problem

To gain maximum benefits from the development tools and the DSP device, it is necessary for the programmer to become familiar with the functionality of both the hardware and software, which can involve a steep learning curve; however, the new development tools available for the TMS320C62xx ensure that the learning process is as smooth as possible.

LSI has been designing systems and working with these new devices for the past year and has learned much about the challenges that will face DSP programmers in the future. These challenges are described in this document, along with typical solutions and suggestions for future DSP system implementation.

Due to the complexity of the new devices, and indeed, future devices, the trend toward high-level languages (HLL) will continue and the overwhelming majority of future application programs will be fundamentally HLL-based, with assembly code used for the time-critical sections.

Solution

The code that comprises typical DSP applications can usually be split into two major categories: signal-processing code and system-control code. The system-control code is often not as time-critical as the signal-processing code and the performance of pure ANSI C code is usually more than adequate. The signal-processing code timing factor, however, often benefits to a greater degree from closer examination and this paper focuses on this code in particular.

Typically, TMS320C62xx DSP code will be generated using a top-down design technique, as follows:

- ❑ High-level ANSI C for functionality
- ❑ Optimized C code, which may include intrinsic functions
- ❑ Code sections in linear assembly
- ❑ Optimized assembly for time-critical sections

To support the new devices and development tools, Texas Instruments has introduced some programming concepts and techniques that will be new to many DSP programmers. The techniques effectively increase the number of stages that the programmer must pass through in the quest for fully optimizing his or her algorithm. The techniques are designed to structure the whole process and this will both reduce the initial design time and reduce the possibility of errors in the final code.

The algorithms presented here have been chosen because they cover some common DSP requirements, including some specifically associated with multi dimensional-array based operations like imaging. The examples provide some useful guidelines that can be applied to other applications and offer an enlightening demonstration of the required DSP software development techniques. Each algorithm was benchmarked on a 120-MHz TMS320C6201-based LSI PCI/C6200.

The algorithms described are:

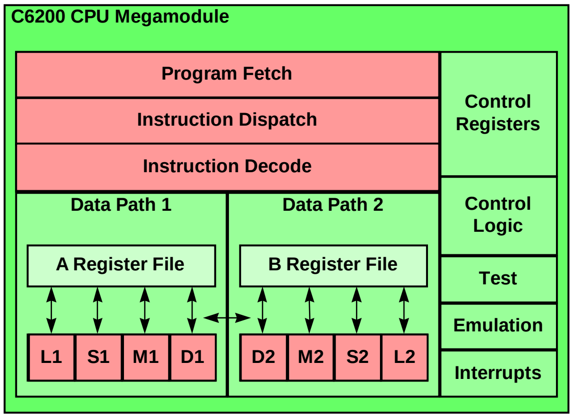
❑ Infinite-Impulse Response filter

❑ Vector add

❑ Two-dimensional 3x3 convolution

Before considering programming any microprocessor, it is necessary to fully understand the architecture of the device. The TMS320C62xx is a VLIW device, which can be viewed as a central processing core surrounded by peripheral devices that support the operation of the core. For code optimization purposes the core is the vital component. The TMS320C62xx core is shown in Figure 1.

Figure 1. The TMS320C6201 CPU Core



The TMS320C62xx architecture incorporates two virtually identical data paths, each of which is capable of performing two16-bit parallel multiply-accumulate operations per cycle. Each data path contains four independent functional units, sixteen general purpose 32-bit registers, a 32-bit load/store path to memory, and a 32-bit cross path to the other data path.

The TMS320C62xx reads a 256-bit (eight 32-bit instructions) wide instruction fetch packet; each fetch packet can contain between one and eight execution packets. An execute packet is simply one or more instructions that operate in parallel. Each instruction within an execute packet is then passed to the appropriate functional unit. A fetch packet executed as eight separate execute packets, or instructions, will take eight times as long to run as a single eight-instruction execute packet.

Each of the register banks incorporates 4 execution units as follows:

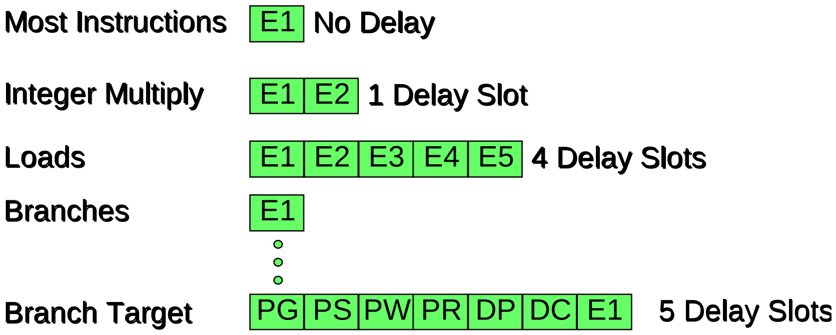
- .S Unit - Logical Unit With Shifter
- .L Unit - Logical Unit
- .D Unit - Data Unit
- .M Unit - Multiply Unit
- I.E. The TMS320C62xx has two Multipliers and six ALUs

The TMS320C62xx features a register-based architecture, with a load-store structure to program code. Each register bank consists of 16 registers and there are cross-paths between register banks A and B to allow for the cross-transfer of data.

All instructions on the TMS320C62xx are conditional; the conditions are valid on the A1, A2, B0, B1, and B2 registers. Parallel instructions are indicated with the || symbol at the start of the command line.

The TMS320C62xx device uses a pipeline to parallel the instruction execution. Of all the instructions, only three (multiply, load, and branch) operations experience delay slots, i.e. there is a delay before the result is written to the register file and before it is available for use by subsequent instructions. For cases where a single operation is being performed and there are no other instructions to execute during the delay slots, multicycle NOP instructions can be used to fill the delay slots, while minimizing the code size.

Figure 2. TMS320C62xx Instruction Delay Slots



The pipeline effects and delay slots experienced by the three instructions mentioned are shown in Figure 2. The diagram shows how the majority of instructions complete in a single execute cycle (E1) but others may require additional delay cycles. The branch operation shows that the delay is due to the number of pipeline stages it takes the branch target to reach the execute stage. The delays do not reduce the ability of the TMS320C62xx to issue a single instruction execution packet on every clock cycle.

The TMS320C62xx devices incorporate a very rich orthogonal instruction set that is supported by a powerful ANSI C compiler. Many of the powerful TMS320C62xx instructions, however, particularly the 16-bit parallel operations that operate on separate halves of 32-bit words, are unsupported by the ANSI standard. TI has, therefore, incorporated intrinsic functions within the C compiler to enable all the TMS320C62xx instructions to be executed with no function call overhead.

As described earlier, the best approach to implementing an algorithm on the TMS320C62xx is via a top-down approach, i.e., to define the algorithm at the C source level and verify that the correct results are generated. Having proved the algorithm, it is then necessary to benchmark the performance, and then optimize, where appropriate. Most DSP operations require repeated processing of arrays of data, with the same mathematical operation performed on all the samples. The DSP instructions performing the processing are repeated with maximum efficiency in a pipelined loop for all the samples, hence, they are referred to as the “piped-loop kernel” of the algorithm. For the analysis of this article, each algorithm will be developed using the top-down approach described, and the piped-loop kernel will be presented. The piped-loop kernel is often preceded by a prologue for initialization and followed by an epilogue for clear down; however, the kernel is the central part of the algorithm that processes the majority of the data and it is here that the optimization is most critical.

The first algorithm to be analyzed is the Infinite-Impulse Response (IIR) filter, which is defined by the following C code:

```
void iir (const short *coefs, const short *input, short *optr, short
*state)
{
    short   x;
    short   t;
    int     n;

    x = input[0];

    for (n = 0; n < 50; n++)
    {
        t = x + ((coefs[2] * state[0] +
            coefs[3] * state[1]) >> 15);

        x = t + ((coefs[0] * state[0] +
            coefs[1] * state[1]) >> 15);

        state[1] = state[0];
        state[0] = t;
        coefs    += 4;  /* point to next filter coefs  */
        state    += 2;  /* point to next filter states */
    }

    *optr++ = x;
}
```

The assembly code for the piped-loop kernel, produced by the C compiler is:

```
L3:        ; PIPED-LOOP KERNEL
           SHR     .S2     B4,15,B4     ;
||         SHR     .S1     A3,15,A5     ;
||         MPY     .M2X    B6,A5,B6     ;@
||         LDH     .D1     *+A6(16),A4  ;@@
||         LDH     .D2     *+B7(10),B6  ;@@

           ADD     .L1     A0,A5,A0     ;
||         MPY     .M1X    B6,A3,A3     ;@
||         MPY     .M2X    B5,A4,B5     ;@
||         LDH     .D1     *+A6(22),A3  ;@@
||         LDH     .D2     *+B7(8),B5   ;@@

           EXT     .S1     A0,16,16,A0  ;
||         STH     .D2     B5,*+B7(6)   ;@
||         MPY     .M1X    B5,A3,A4     ;@
||         LDH     .D1     *+A6(20),A3  ;@@

           ADD     .S1     8,A6,A6      ;
||         STH     .D2     A0,*B7++(4)  ;
||         ADD     .L1X    A0,B4,A0     ;
|| [ B0]   SUB     .L2     B0,1,B0      ;@
||         ADD     .S2     B6,B5,B4     ;@

           EXT     .S1     A0,16,16,A0  ;
|| [ B0]   B       .S2     L3           ;@
||         ADD     .L1     A3,A4,A3     ;@
||         LDH     .D1     *+A6(18),A5  ;@@@
```

The results show that the execute packets contain either four or five parallel instructions, hence, the TMS320C62xx processing units are not fully utilized and there is a possibility for optimizing the performance of this code. The @ characters in the comments specify the iteration of the loop that an instruction is on in the software pipeline and are automatically generated by the tools. For example, while the `shr` instructions are executing iteration j of the loop, `mpy` is executing iteration j+1 and the `ldh` instructions are executing iteration j+2. The scheduling of the iteration of the instructions within the piped loop is a result of the prologue leading up to the execution of the piped-loop kernel.

If the DSP is processing 16-bit data, then the first level of optimization will be to utilize the 32-bit external bus to increase the data rates through the CPU core by performing two parallel 16-bit reads in a single 32-bit word. The 16-bit data can then be processed using the TMS320C62xx `_mpy` and `_mpyhl` operations, which can be accessed via the C-level intrinsic functions, as shown in the following C code:

```
void iir (const int *coefs, const short *input, short *optr, short
*state)
{
    short   x;
    short   t;
    int     n;

    x = input[0];

    for (n = 0; n < 50; n++)
    {
        t= x+((_mpy(coefs[1],state[0]) + _mpyhl(coefs[1],state[1]))
>> 15);
        x= t+((_mpy(coefs[0],state[0]) + _mpyhl(coefs[0],state[1]))
>> 15);

        state[1] = state[0];
        state[0] = t;

        coefs += 2;
        state += 2;
    }

    *optr++ = x;
}
```

The assembly code for the piped-loop kernel, produced by the C compiler is:

```
L3:        ; PIPED-LOOP KERNEL

           ADD     .L2     B7,B8,B7      ;
||         ADD     .L1     A0,A3,A0      ;
||         MV      .S2     B6,B9         ;@
||         STH     .D1     A5,*+A4(6)    ;@
||         LDW     .D2     *B5++(8),B8   ;@@

           SHR     .S2     B7,15,B7      ;
||         EXT     .S1     A0,16,16,A0   ;
|| [ B0]   SUB     .L2     B0,1,B0       ;@
||         MPY     .M2X    B8,A5,B8      ;@
||         ADD     .L1X    B6,A3,A3      ;@
||         LDH     .D2     *+B4(14),B6   ;@@@

           ADD     .L1X    A0,B7,A6      ;
||         MPYHL   .M2     B8,B9,B7      ;@
||         SHR     .S1     A3,15,A3      ;@
|| [ B0]   B       .S2     L3            ;@
||         LDW     .D2     *+B5(4),B7    ;@@@
||         LDH     .D1     *+A4(12),A5   ;@@@

           ADD     .L2     4,B4,B4       ;
||         STH     .D1     A0,*A4++(4)   ;
||         EXT     .S1     A6,16,16,A0   ;
||         MPYHL   .M2     B7,B6,B6      ;@@
||         MPY     .M1X    B7,A5,A3      ;@@
```

The assembly code shows that the coefficients are loaded two at a time, as single 32-bit operations. The parallel loads optimize the data I/O efficiency but require that the coefficients are contiguous in memory, although this is not usually a problem for DSP applications. The results also show that the piped-loop kernel has now been reduced to four instruction fetch packets, which is the most efficient implementation of the IIR algorithm that is possible using pure C code. To optimize the code further, it is now necessary to use linear assembly code.

Linear assembly is similar to regular TMS320C62xx assembly code, in that TMS320C62xx instructions are used to write the code; however, it frees the programmer from some of the time-consuming aspects of pure assembly code programming, and hence, shortens development time drastically. In linear assembly code, the programmer can specify some, or all of the information required, or he/she can allow the assembly optimizer to specify it. Information such as register usage, functional unit and more can be omitted during a first-pass approach and then more detail can be added to further control CPU resource allocation and to fully utilize the device.

The following linear assembly code shows how the IIR function can be implemented, and also, how the optional parameters can be utilized:

```
.def    _iir3

_iir   .cproc   cptr0,sptr0

        .reg cptr1, s01, s10, s23, c10, c32, s10_s, s10_t
        .reg p0, p1, p2, p3, s23_s, s1, t, x, mask, sptr1, s10p, ctr

        MV   .2    cptr0,cptr1
        MV   .1    sptr0,sptr1

        MVK     50,ctr                  ; setup loop counter

LOOP:   .trip 50

        LDW     .D1T1 *cptr0,c32      ; coefAddr[3] & CoefAddr[2]
        LDW     .D2T2 *cptr1,c10      ; CoefAddr[1] & CoefAddr[0]
        LDW     .D1T2 *sptr0,s10      ; StateAddr[1] & StateAddr[0]
        MV      .2 s10,s10p        ; save StateAddr[1] & StateAddr[0]

        MPY     .M1   c32,s10,p2      ; CoefAddr[2] * StateAddr[0]
        MPYH    .M1   c32,s10,p3      ; CoefAddr[3] * StateAddr[1]
        ADD     .1 p2,p3,s23       ; CA[2] * SA[0] + CA[3] * SA[1]
        SHR     .1 s23,15,s23_s    ; (CA[2] * SA[0] + CA[3] * SA[1])
>> 15
        ADD     .2 s23_s,x,t       ; t =
x+((CA[2]*SA[0]+CA[3]*SA[1])>>15)
        AND     .2 t,mask,t        ; clear upper 16 bits

        MPY     .M2   c10,s10,p0      ; CoefAddr[0] * StateAddr[0]
        MPYH    .M2   c10,s10,p1      ; CoefAddr[1] * StateAddr[1]
        ADD     .2 p0,p1,s10_t     ; CA[0] * SA[0] + CA[1] * SA[1]
        SHR     .2 s10_t,15,s10_s  ; (CA[0] * SA[0] + CA[1] * SA[1])
>> 15
        ADD     .2 s10_s,t,x       ; x =
t+((CA[0]*SA[0]+CA[1]*SA[1])>>15)

        SHL     .2 s10p,16,s1      ; StateAddr[1] = StateAddr[0]
        OR      .2 t,s1,s01        ; StateAddr[0] = t
        STW     .D1   s01,*sptr1      ; store StateAddr[1] &
StateAddr[0]

  [ctr] ADD     .S1   -1,ctr,ctr      ; dec outer lp cntr
  [ctr] B       .S1   LOOP            ; Branch outer loop

             .endproc
```

The linear assembly is passed through the linear assembler and the resultant assembly code for the piped-loop kernel is:

```
L3:        ; PIPED-LOOP KERNEL

           AND     .L2     B3,B7,B0      ; clear upper 16 bits
||         ADD     .S2     B0,B8,B8      ;@ CA[0] * SA[0] + CA[1] *
SA[1]
|| [ A1]   B       .S1     L3            ;@ Branch outer loop
||         ADD     .L1     A4,A5,A4      ;@ CA[2] * SA[0] + CA[3] *
SA[1]
||         MPYH    .M2     B2,B1,B8      ;@@ CoefAddr[1] *
StateAddr[1]
||         MPY     .M1X    A0,B1,A4      ;@@ CoefAddr[2] *
StateAddr[0]
||         LDW     .D2     *B6,B2        ;@@@@ CoefAddr[1] &
CoefAddr[0]
||         LDW     .D1     *A3,A0        ;@@@@ coefAddr[3] &
CoefAddr[2]

           ADD     .D2     B4,B0,B9      ; x =
t+((CA[0]*SA[0]+CA[1]*SA[1])>>15)
||         OR      .L2     B0,B9,B0      ; StateAddr[0] = t
||         SHR     .S2     B8,0xf,B4     ;@ (CA[0] * SA[0] + CA[1] *
SA[1]) >> 15
||         SHR     .S1     A4,0xf,A5     ;@ (CA[2] * SA[0] + CA[3] *
SA[1]) >> 15
||         MPY     .M2     B2,B1,B0      ;@@ CoefAddr[0] *
StateAddr[0]
||         MPYH    .M1X    A0,B1,A5      ;@@ CoefAddr[3] *
StateAddr[1]
||         LDW     .D1     *A6,B1        ;@@@@ StateAddr[1] &
StateAddr[0]

           STW     .D1     B0,*A7        ; store StateAddr[1] &
StateAddr[0]
||         SHL     .S2     B5,0x10,B9    ;@ StateAddr[1] =
StateAddr[0]
||         ADD     .L2X    B9,A5,B3      ;@ t =
x+((CA[2]*SA[0]+CA[3]*SA[1])>>15)
|| [ A1]   ADD     .S1     0xffffffff,A1,A1 ;@@ dec outer lp cntr
||         MV      .D2     B1,B5         ;@@ save StateAddr[1] &
StateAddr[0]
```

The piped-loop kernel has now been reduced to three instruction fetch packets and it is clear that with eight instructions per fetch packet, that there is no further room for optimization. The pipelined code also shows that the eight TMS320C62xx processing units (.Dx, .Sx, .Mx, .Lx) are almost fully utilized. Another benefit of the assembly optimizer, as shown above, is that it puts all the original comments in the scheduled output, so it is easy to see what is going on in the code.

The following table shows the results of the different levels of optimization for the IIR filter:

Table 1. IIR Filter Benchmark Results

Development Technique	**Number Of Cycles**
ANSI C	5
C with intrinsic functions	4
Linear assembler	3

The next algorithm to be analyzed is the vector addition operation, which is defined by the following C code:

```
short Add(short *x1, short *x2, short *y,
             short count)
{
             short i;
             for (i=0; i < count; i++)
             {
                y[i] = x1[i] + x2[i];
             }
}
```

It is clear from the C source code that this operation requires three external memory accesses per sample (two reads and one write). The assembly code produced by this operation will not be able to execute in a single instruction because the TMS320C62xx has two .D units for loading and storing the data. The assembly code for the piped-loop kernel produced by the compiler is thus:

```
L31:        ; PIPED LOOP KERNEL

                    ADD     .L1X    B4,A0,A5
|| [ B0]     B       .S2     L31
||           LDH     .D1     *A3++,A0

             STH     .D1     A5,*A4++
|| [ B0]     SUB     .L2     B0,1,B0
||           LDH     .D2     *B5++,B4
```

This function executes one addition operation every two instruction cycles, which suggests that there might be room for improvement by better utilizing the currently unbalanced CPU addition resources. The routine can be rewritten using intrinsic functions to perform parallel additions as follows:

```
short Add(short *x1, short *x2, short *y,
             short *x1a, short *x2a, short *ya,
             short count)
{
             short i;
             for (i=0; i < count; i++)
             {
             y[i] = _add2(x1[i], x2[i]);
             ya[i] = _add2(x1a[i], x2a[i]);
             }
}
```

The code produced by the compiler is now:

```
L13:         ; PIPED LOOP KERNEL

           ADD2    .S2X    A3,B5,B6
|| [ B0]   B       .S1     L13
||         LDW     .D1     *A4++,A5
||         LDW     .D2     *B7++,B5

           ADD2    .S1X    B5,A5,A3
||         STW     .D2     B6,*B4++
||         LDW     .D1     *A0++,A3

           STW     .D1     A3,*A6++
|| [ B0]   SUB     .L2     B0,1,B0
||         LDW     .D2     *B8++,B5
```

The above code calculates two data samples in parallel. The assembly code above shows that the new version has balanced the CPU addition resources and now executes four addition operations every three cycles, which is a performance improvement of 167%. This implementation uses 32-bit load / store operations, in preference to 16 bits; this halves memory bandwidth requirement and is 100% optimized with respect to the load / store operations.

The TMS320C62xx is ideally suited to image-processing applications because the 16-bit operations give optimum performance and processing headroom on the pixel word width. The

2-D 3x3 convolution operation is defined by the following C code:

```
short Conv3x3(short row0[], short row1[],
                short row2[], short y[])
{
             short i;
             for (i=0; i < width-2; i++)
             {
             y[i] = row0[i]*kernel[0][0]
                     + row0[i+1]*kernel[0][1]
                     + row0[i+2]*kernel[0][2]
                     + row1[i]*kernel[1][0]
                     + row1[i+1]*kernel[1][1]
                     + row1[i+2]*kernel[1][2]
                     + row2[i]*kernel[2][0]
                     + row2[i+1]*kernel[2][1]
                     + row2[i+2]*kernel[2][2];
```

This implementation requires 6 cycles to process each pixel; however, it can be rewritten as follows:

```
short Conv3x3(short row0[], short row1[],
                short row2[], short y[])
{
  short i;
  for (x=0; I < dx; i++)
  {
    acc1 = _mpy (row0[i], a00) + _mpyh (row0[i], a00) + _mpy
(row0[i+1], a02);
    acc2 = _mpyhl (row0[i], a00) + _mpylh (row0[i+1], a00) + _mpyhl
(row0[i+1], a02);

    acc1 += _mpy (row1[i], a10) + _mpyh (row1[i], a10) + _mpy
(row1[i+1], a12);
    acc2 += _mpyhl (row1[i], a10) + _mpylh (row1[i+1], a10) + _mpyhl
(row1[i+1], a12);

    acc1 += _mpy (row2[i], a20) + _mpyh (row2[i], a20) + _mpy
(row2[i+1], a22);
    acc2 += _mpyhl (row2[i], a20) + _mpylh (row2[i+1], a20) + _mpyhl
(row2[i+1], a22);

    *y++ = acc1; *y++ = acc2;
    row0++; row1++; row2++;
  }
}
```

This implementation of the algorithm requires 9 instruction cycles to calculate the results for 2 pixels, which allows a 33% performance increase, and again parallel data loads and stores improve I/O bandwidth requirements. In this application, the TMS320C62xx core is now 100% optimized since both multipliers are used every cycle.

It is clear from the tests performed that the C compiler optimizer provides a good first pass toward optimum code development, improving the performance of the code, and is very successful at eliminating redundant variables and repeated memory accesses. The optimizer also enables big gains in the pipelining of loops. In many instances, all eight parallel instruction slots can be used. In order to achieve this high performance, the compiler must have certain knowledge of the loop, such as memory dependencies and minimum loop-trip count. These are both documented in the TMS320C62xx programmer's guide.

When the compiler cannot ensure that the trip count is large enough to pipeline a loop for maximum performance, a pipelined and a nonpipelined version of the same loop is generated. The compiler provides a statement (`_nassert`) and the assembly optimizer includes a directive (`.trip`) for indicating the minimum number of iterations of a loop for this purpose.

Performance gains can often be made by the unrolling of loops, which can optimize the data-load bandwidth or balance resources. Loop-unrolling helps many smaller code loops, however, it can lead to a significant increase in the code size when the loop contains a large number of instructions. The loop must not contain any conditional breaks or function calls, although these may be inlined.

In applications where the TMS320C62xx execution units are not fully utilized, it is often possible to parallel separate data-flow streams. This separates data dependencies and allows for greater performance.

Table 2. Tips for Optimizing TMS320C62xx Code

Use internal memory
C pointers do not necessarily beat arrays
Use intrinsics where possible
Use 32-bit loads and stores, if possible
Try unrolling loops (if short)
Separates data dependencies
Balances resources
Experiment!

Conclusion

This article has shown how the new Texas Instruments TMS320C6201 DSP is supported by a new generation of development tools. The compilers and assemblers fully utilize the on-chip features and functionality of the device; good examples include the packing of instructions into fetch packets to enable greater use of the on-chip memory and also algorithmic and functional optimizations to optimize the performance and data throughput.

Once a programmer leaves C, then linear assembly — as shown in the first example — is an excellent language for developing highly optimized routines; therefore, the time savings over using pure assembly code are enormous. There are few cases where resorting to pure assembler is essential; however, it is often advantageous to use linear assembly as the starting point, then, modify the output to save valuable coding time. A final benefit of using linear assembly is that it provides virtually 100% code-portability to future C6x family devices and beyond, which will ensure that optimized code developed now will not become redundant in a few years time.

It is clear that the future generations of DSP devices will require DSP engineers to embrace new programming techniques and disciplines. The techniques described in this application report cover several trade-offs between computational and memory requirements. The next generation of C compilers for DSPs will make life a great deal easier via the addition of such features as intrinsic functions. The choice of optimization techniques used in the development of any particular project will depend entirely on the final system requirements.

References

[1] A. V. Oppenheim, R. W. Schafer, *Discrete Time Signal Processing*, Prentice-Hall, 1989.

[2] Loughborough Sound Images Inc., *PCI/C6200 User's Guide*, Loughborough Sound Images Inc., 1997.

[3] Loughborough Sound Images Inc., *PCI/C6200 Technical Reference M*anual, Loughborough Sound Images Inc., 1997.

[4] Texas Instruments Inc., *TMS320C62xx CPU and Instruction Set*, Texas Instruments Inc., 1997.

[5] Texas Instruments Inc., *TMS320C6x Optimizing C Compiler User's Guide*, Texas Instruments Inc., 1997.

[6] Texas Instruments Inc., *TMS320C6x Assembly Language Tools User's Guide*, Texas Instruments Inc., 1997.

[6] Texas Instruments Inc., *TMS320C62xx Programmer's Guide*, Texas Instruments Inc., 1997.

INTERNET

www.ti.com

Register with TI&ME to build custom information pages and receive new product updates automatically via email.

TI Semiconductor Home Page
http://www.ti.com/sc

TI Distributors
http://www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

US TMS320

Hotline	(281) 274-2320
Fax	(281) 274-2324
BBS	(281) 274-2323
email	dsph@ti.com

Americas

Phone	+1(972) 644-5580
Fax	+1(972) 480-7800
Email	sc-infomaster@ti.com

Europe, Middle East, and Africa

Phone	
Deutsch	+49-(0) 8161 80 3311
English	+44-(0) 1604 66 3399
Francais	+33-(0) 1-30 70 11 64
Italiano	+33-(0) 1-30 70 11 67
Fax	+33-(0) 1-30-70 10 32
Email	epic@ti.com

Japan

Phone	
International	+81-3-3457-0972
Domestic	+0120-81-0026
Fax	
International	+81-3-3457-1259
Domestic	+0120-81-0036
Email	pic-japan@ti.com

Asia

Phone	
International	+886-2-3786800
Domestic	
Australia	1-800-881-011

Asia (continued)

TI Number	-800-800-1450
China	10811
TI Number	-800-800-1450
Hong Kong	800-96-1111
TI Number	-800-800-1450
India	000-117
TI Number	-800-800-1450
Indonesia	001-801-10
TI Number	-800-800-1450
Korea	080-551-2804
Malaysia	1-800-800-011
TI Number	-800-800-1450
New Zealand	+000-911
TI Number	-800-800-1450
Philippines	105-11
TI Number	-800-800-1450
Singapore	800-0111-111
TI Number	-800-800-1450
Taiwan	080-006800
Thailand	0019-991-1111
TI Number	-800-800-1450

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current and complete. TI warrants performance of its semiconductor products and related software 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. Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office. In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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 is a trademark of Texas Instruments Incorporated.
Other brands and names are the property of their respective owners.