

FEATURE ARTICLE

Michael Smith

The SHARC in the C

In a recent project, Mike set out to develop DSP algorithms suitable for producing an improved sound stage for headphones. Using the Analog Devices 21061 SHARC, he modified the phase and amplitude of the audio signal before it is sent to the ear, thus creating “virtual” speakers that give the effect of listening via room speakers.



In a recent project that I did, I became interested in developing DSP algorithms suitable for producing an improved sound stage for headphones. Listening to music through a standard headset (see Figure 1) leaves the listener with the impression that the music is inside their head, a different feeling than listening to the same music while using speakers. However, the sound stage of the headphones can be drastically changed if the phase and amplitude of the audio signal are modified before being sent to the ear. For example, the perceived position of a monosound signal can be altered by modifying the relative time of arrival of the same sound at the left and right ear.

Creating the effect of a series of virtual speakers with room reverberation can be handled using extensive DSP techniques, such as implementing a series of finite impulse response (FIR) and Comb filters, a cross between FIR and infinite duration impulse filters (IIR). Sampling

frequencies must be 44 to 48 kHz for good sound quality. A specialized architecture DSP processor, or sound card, is needed to process one sound bite before the next bite arrives.

For efficient code development, you need to make the appropriate language choice for the various system components. One line of debugged code takes roughly the same time and effort in any language. This means that, other things being equal, developing modules using assembly language should be avoided when higher-level languages are available.

There are a number of different components needed for this sound stage project. Standard C for the GUI interface is used to modify speaker and room characteristics. The DSP components are best handled as independent interrupts using hardware circular buffers and other custom memory addressing to take advantage of special processor architectural features. Setting up the hardware might require calling an assembly code routine directly from the higher-level language.

You must become aware of the interaction between assembly code and the C environment to handle coding in such an embedded environment. This interaction for a CISC processor was detailed in the article I

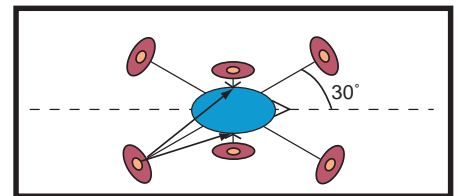


Figure 1—The sound source is perceived in the center of the head if exactly the same sound comes from both left and right earphones. If the sound is delayed before being sent to the left earphone, then the perceived position of the sound source shifts to the right side of the head. Furthermore, modeling of the audio channel using FIR and IIR filters can move the perceived sound out to a virtual speaker in front or back of the listener's head. [1]

wrote, "Some Assembly Required," (*Circuit Cellar*, 101). However, interfacing between assembly code and C functions is different on newer DSP chips because of the new architectural features in DSP processors.

On the positive side, additional data and address registers are available. Specialized hardware allows fast switching between subroutines or handling zero-overhead loops. On the negative side, many of the new processor features are not directly describable using the standard C language. What's the syntax for accessing an array using the bit-reversed, circular buffer address register operations? Some of the speed-improving hardware features impose restrictions that can't be handled through a standard C programming model.

In this article, I look at the assembly code of C and assembly code interfaces on Analog Devices' 21061 SHARC. These are compared to those found with the Software Development System's (SDS) Motorola 68K CISC C compiler.

The SHARC assembly language is C-like in format, which makes the comparison relatively straightforward. Only C functions calling assembly code functions will be considered. There is little advantage in going in the opposite direction because the whole point of switching to assembly code from a C subroutine is speed.

REGISTER COMPARISON

You must understand the function and uses of the processor registers before trying to tackle a link between assembly code and C. The available registers (programmer's model) for the 68K and 21K processors are shown in Table 1. There are as many similarities as there are differences between these processors.

The sixteen 21K data registers (R1-R15) have essentially the same functionality as the eight 68K data registers (D0-D7). However, there are

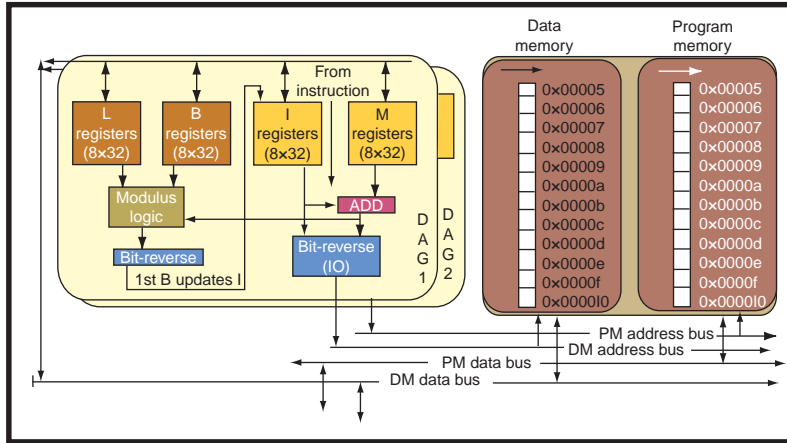


Figure 2—This is the Data Address Generation Block (DAG1) on the SHARC 2106X DSP processor. The modulus logic in DAG1 permits hardware circular buffer operations to occur with zero-overhead, but poses problems for the C programmer.

many hidden differences. There is an alternate set of 21K data registers available for fast interrupt handling, where as the 68K registers must be saved to slow external memory. The 21K data registers can be used for both integer (R0-R15) and floating point (F0-F15) operations.

The 32-bit addition register-to-register operation, $REG0 = REG0 + REG1$, is written on the two processors as:

```
ADD.L D1, D0      (68K)
R0 = R0 + R1;     (21K)
```

The 21K assembler format has a number of other advantages in addition to its C-like characteristics. First, there is no hidden source register like in the 68K syntax. At the end of a long day working on the 68K, I might start wondering if D0 was meant to be added to D1 and stored into D1, or was D1 supposed to be added to D0 and stored in D0?

Note that the use of the semicolon to signal the end of an assembly instruction permits a single 21K instruction to be written across many lines of code. This free formatting also allows documentation of the instructions describing parallel operations to multiple registers and memory accesses in a single cycle.

Invocation of the 21K processor's super-scalar capability requires syntax in the form of:

```
F0 = F1 * F4, F2 = F8 + F12;   (21K)
```

Simultaneous multiplication and addition in a single cycle is available only within certain 21K register banks. This is a limitation of the number of bits available on the 21K program data bus, even though this, at 48 bits, is much wider than the 16 bits of the 68K data bus.

MEMORY ACCESS

Figure 2 shows the Data Address

Generator DAG1 Block from the SHARC 2106X processor. The eight 21K index registers (I0-I7) play roughly the same role as the eight 68K address registers (A0-A7). However, the similarity ends there.

A main limitation of the 68K for DSP operations is the frequent conflicts between data fetches and instruction fetches on a single data bus. The 21K's Harvard architecture removes this problem by having both a *program* memory data bus (for instructions) and a *data* memory data bus. And, the SHARC's large on-chip, fast access memory provides more speed.

Even with the Harvard architecture, there will be data/data conflicts when a large amount of data is being manipulated within a tight DSP loop. On the SHARC, this problem is overcome by storing instructions within an instruction cache and allowing data fetches to occur simultaneously down both the *program* memory data bus and the *data* memory data bus.

This architectural feature is handled by special C extensions:

```
int dm data[200]
int pm coeffs[200]
```

The *dm* syntax indicates that the *data[]* array is stored in data memory for fast access through the *data* memory data bus. The *pm* syntax indicates that the *coeffs[]* array is stored in program memory for fast access through the *program* memory data bus.

	68K Processor	21K Processor
Integer data registers	D0–D7	R0–R15
Floating point registers		F0–F15, same as R registers
Subroutine return value	D0	R0
Subroutine parameters	On the stack	R4, R8, R12
Address registers	A0–A7	
Index registers	I0–I7 (data memory)	
I8–I15 (program memory)		
Modify registers		M0–M7, M8–M15
Length registers		L0–L7, L8–L15
Base registers		B0–B7, B8–B15
Volatile registers	D0, D1, A0, A1	R0, R1, R2, R4, R8, R12
I4, M4, I12, M12		
Alternate register banks		For Rx, Ix/Mx/Lx/Bx
Frame pointer (convention)	A6	I6, L6 = 0, M5, M6, M7
C stack pointer	A7	I7, L7 = 0, M5, M6, M7

Table 1—Programmer's model of the main registers on the 68K CISC processor and the 21K SHARC DSP processor.

At the assembly code level, the first bank of 21K Data Address Generator (DAG1) index registers (I0–I7) allows access to *dm* memory in parallel with access to the *pm* memory using the DAG2 index registers (I8–I15).

Access to arrays stored in memory can be accomplished using either set of DAG registers because of the SHARC's onboard memory organization. This can be confusing for the inexperienced developer because the introduced bus conflicts are handled transparently by the 21K. The conflict results in additional bus cycles being introduced, rather than the expected high-speed, parallel memory operations!

MODIFY AND VOLATILE REGISTERS

The 21K modify registers (M0–M7 and M8–M15) can be used in conjunction with the index registers to access elements in an array. This is equivalent to using a 68K data register in conjunction with an address register. Access to the 32-bit array element *data[3]* will be programmed on the two processors as follows:

```
MOVE.L #data, A0
(68K)
MOVE.L #(4*3), D0
MOVE.L (A0, D0), D1
Offset scale to 12 bytes
```

```
I4 = data;
(21K)
M4 = 3;
R1 = dm(M4, I4)
```

Note that the offset of element *data[3]* from the start of the array *data* is 12 bytes on the 68K, but three

words on the 21K. You'll have to get deep into the 21K User Manual to discover the advantages of having three 21K words, 12 bytes long under some circumstances and 16 bytes long under others.

These simple code sequences also indicate the differences in the coding conventions adopted for volatile register usage in the two development environments. When program flow requires a subroutine call, it is important that key values remain undisturbed in registers for reuse when the subroutine exits. Volatile registers can be used in a subroutine without saving every register to slow memory.

With the SDS 68K compiler, there are two volatile data registers (D0 and D1) defined, and the 21K registers (R0 and R1) have a similar usage. However, there are four additional SHARC volatile data registers—R2, R4, R8, and R12. This strange choice is not arbitrary, but it meets the requirement to have volatile registers available to use with the 21K super-scalar operations.

Both 68K and 21K coding conventions allow for two volatile address registers. However, while the 68K volatile address registers are the obvious A0 and A1, the 21K equivalent index registers are I4 and I12. This choice matches the need to access both *program* and *data* memory. The volatile 21K data registers (unlike the volatile 68K data registers) can't be used in conjunction with the index registers to step through an array. Specific volatile 21K modify registers (M4 and M12) are needed for this purpose.

Note that there are both *PREMODIFY* and *POSTMODIFY* memory accessing modes on the 21K.

```
M4 = 4;
R1 = dm(M4, I4); PREMODIFY
R2 = dm(I4, M4); POSTMODIFY
```

The value at memory location *I4 + M4* is fetched during the premodify operation, with index register *I4* left unchanged. In the postmodify operation, the memory at location *I4* is fetched, and then the index register is autoincremented so that $I4 = I4 + M4$. The postmodify operation can be described in a few bits, which allows parallel postmodify operations to be described in a single opcode to *program* and *data* memory.

LENGTH AND BASE REGISTERS

There are two sets of 21K registers that have no equivalent in the 68K architecture—the base and length registers. The length register significantly affects accessing data arrays within a C/C++ program.

This effect is hidden in the code sequences, which use the C default settings of the length and base registers. Listing 1 shows the C code and assembly language needed to calculate the sum of the first three elements in a 10-element array using an autoincrementing pointer for both 68K and 21K processors. The only difference is the need to perform a 21K load-memory-to-register operation before performing the *ADD*. The 21K doesn't have the complex addressing capabilities of the 68K CISC architecture. Mind you, a 40-MHz 21K performs the fetch-and-add operation in two clock cycles, while a 40-MHz 68K takes 16 cycles.

Listing 2 has a hidden kick in its operation, although it looks similar to Listing 1. The length (L4) and base registers (B4) establish a two-word long circular buffer. The first two memory fetches work as expected. However, in the second post-modify operation, index register *I4* is incremented to point to *VALUES[2]* and then modified by the SHARC circular buffer hardware to point back to *VALUES[0]* (see Figure 2).

For standard C array handling, the length register *Lx* associated with index register *Ix* must remain 0 or be returned to 0. Pity the poor 21K code developer who has to try maintaining C code where the base and length registers are unintentionally left modified by an interrupt service routine that is infrequently invoked!

PASSING PARAMETERS

Both registers and the C-stack are used for parameter passing on the 68K and 21K processors. One register is typically designated for returning values from a function, D0 (68K) and R0 (21K), with occasional assistance from other registers.

With the limited number of 68K registers available, parameters are typically passed to subroutines by pushing them onto the C-stack above the return address. By contrast, the first three subroutine parameters are passed via 21K data registers—R4, R8, and R12.

Even pointer values (e.g., `int * pt`) are passed through the data registers. This can invoke a string of error messages from the 21K. The 21K registers can't be used for both address and data purposes as many RISC general registers can. The 21K pointer value must be moved from the data register parameter into a (volatile) index register. Even with this complication, passing parameters via registers still saves considerable time over putting things on and off an external memory stack. As stated earlier, I'll ignore the complications arising from situations where subroutines call other subroutines.

HARDWARE AND SOFTWARE

Things get interesting with the SHARC's C programming model when the programmer attempts to pass the fourth parameter to the subroutine. On a register-windowed processor, such as the SPARC or the 29K RISC, this wouldn't be a problem. The 29K has 128 registers for parameter passing. However, the SHARC has no other allocable volatile registers, and the fourth parameter must be passed another way.

It will do you no good to try to do the 68K-trick, passing the parameter

above the return address on the stack. The standard place for the 21K return address is on a specialized high-speed PC stack. This hardware is provided without access to the data registers and can only hold a few values!

There are a number of other problems, as well. A characteristic of the C program is the presence of subroutines calling other subroutines. The innermost subroutine in a program could easily be nested 8 or 10 subroutines deep inside the main function. This type of operation cannot be handled with a shallow hardware stack using the standard SHARC subroutine CALL and RTS instructions. Also, there's the problem of all the variables and arrays declared inside the C function, or interrupt handling, when copious material must go onto the stack.

The approach taken on the SHARC brings back fond memories of my early days developing embedded systems with microprogrammable chips. If you didn't have the required instructions, you either added 'em or faked 'em.

In the SHARC C programming model, index registers I6 and I7 are set up as a frame pointer and a C-top-of-stack pointer, respectively. These registers function essentially in the same way as the 68K frame (A6) and stack pointer (A7). However, the SHARC I7 register points to the next empty stack location, whereas the 68K A7 register points to the current valid value on the stack.

Several modify registers are initialized to values 0, 1, and -1 to speed address stack operations on their way in the SHARC in the C. Of course, to

Listing 1—A comparison of C code, 68K, and 21K assembly code to calculate the sum of the first three elements of a 10-element array.

```
C code segment
    int values[10];
    int sum = 0;
    int *pt = values;
    for (count = 0; count < 3; count++)
        sum = sum +*pt++;

68K code segment
    section data
VALUES: DS.L 40

    section code
        // Clear the sum variable
MOVE.L #0, D0
        // Set up the pointer to the array
MOVE.L #VALUES, A0
        // Calculate the sum
ADD.L (A0)+, D0 // Fetch, add
ADD.L (A0)+, D0 // and increment
ADD.L (A0)+, D0
        // A0 pointing to VALUES[3]

21K code segment
.segment/dm seg_dmda;
.var VALUES[10];
.endseg;

.segment/pm seg_pmco;
        // Clear the sum variable
R0 = 0;
        // Set up the pointer to the array
I4 = VALUES;
        // Set up address modify constant
M4 = 1;
        // Calculate the sum
R1 = dm(I4, M4); // Fetch
R0 = R0 + R1; // and add
```

Listing 2—There will be interesting surprises in the C code operation if the 21K circular buffer hardware is left activated.

```
C code segment
int values[10];
int sum = 0;
int *pt = values;
for (count = 0; count < 3; count++)
    sum = sum +*pt++;

21K code segment
.segment/dm seg_dmda;
.var VALUES[10];
.endseg;

.segment/pm seg_pmco;

// Some where else in the code the 21K
// circular buffer hardware gets left activated

    B4 = VALUES;
    L4 = 2;

// Same code as before
// Clear the sum variable
R0 = 0;
// Setup the pointer to the array
I4 = VALUES;
// Set up address modify constant
M4 = 1;
// Calculate the sum
R1 = dm(I4, M4); // Fetch
R0 = R0 + R1; // and add
R1 = dm(I4, M4); // Fetch
R0 = R0 + R1; // and add
// I4 is incremented, and
// then adjusted by the
// circular buffer hardware
// to point back to VALUES[0]
R1 = dm(I4, M4); // Fetch
R0 = R0 + R1; // and add
// I4 pointing to VALUES[1]
```

prevent surprises, avoid introducing a C-stack with circular buffer characteristics. Remember to keep length registers L6 and L7 at zero, especially when interrupts are active.

The final touch is to provide some C-specific instructions to the SHARC instruction set. *CJUMP* is used when getting into a C-callable subroutine, and *RFRAME* when leaving. You'll have to search the SHARC Technical Reference Guide because these instructions are not detailed in the standard SHARC user manual.

The SHARC *CJUMP* instruction, together with other instructions hidden in the branch delay slots, is equivalent to the combined 68K instructions:

JSR with LINK #0, FP

The technical guide states, "CJUMP combines a direct or PC-relative jump with register transfer operations that save the frame and stack pointers."

However, the registers are not saved on the stack. There are some unfortunate things happening with register R2 in this instruction, but should not cause a problem if you remember that R2 is designated as a volatile register. Nothing useful should be stored there during a subroutine call.

The *RFRAME* instruction is used as part of the return sequence from an assembly language routine back to a C-calling program. This is equivalent to the 68K *UNLK* instruction, including that instruction's memory access.

Fortunately, the necessary instructions for returning to a C-calling func-

tion from assembly code are always the same and can be cut and pasted into a macro *DO_MAGIC*. In this macro, the return address from the C-stack is fetched. As part of the indirect jump, using a program data memory volatile index register, the return address is tweaked by an instruction to get it pointing in the proper direction. Finally, the *RFRAME* instruction causes the C-top-of-stack to be copied from the frame pointer and the old frame pointer recovered from the C-stack during the branch delay slots.

```
#define DO_MAGIC \
    I12 = dm(-1, I6); \ // GET \
RETURN ADDRESS \
    JUMP(1, I12) (DB); \ // ADJUST \
ADDRESS A LITTLE \
    nop; \
    RFRAME; // AD- \
JUST FRAM AND TOP-OF-STACK \
POINTERS
```

INTERRUPT HANDLING

In this article, I am more interested in the C programming side of interrupt handling on the SHARC rather than specific hardware details. However, it is not possible to completely separate the two.

With the SDS 68K compiler, the statement *#pragma interrupt()* must be added before the code for the C interrupt service routine (ISR). This informs the compiler that the ISR must be handled differently from a subroutine. In particular, volatile registers must be saved/recovered, and an *RTI* (rather than an *RTS*) instruction is needed at the end of the routine.

During the 68K main function, the starting address for the ISR routine must be placed at the appropriate location in the vector table. Then the interrupt must be activated. The equivalent of these events must be present with the SHARC chip and compiler. However, the implementation details are different.

Unlike the 68K with its interrupt vector table, the starting addresses of the SHARC interrupt service routines begin at a fixed location in memory. Each interrupt is provided with a fixed number of instructions within this

area. For longer routines, a jump must occur to code elsewhere in memory.

There is no `21K #pragma interrupt()` preprocessor command to designate an ISR as something other than a subroutine. Instead, there are three different approaches within the SHARC C code that can be used to link an interrupt to a specific ISR routine. The following code links the `IRQ1` interrupt with the C subroutine (or C-compatible assembly subroutine) `DoSomething()`:

```
#include <signal.h>
interrupt(SIG_IRQ1, DoSomething);
```

The call `interrupt()` modifies a lookup table used to inform the interrupt handler that `IRQ1` interrupts will require that all possible registers (Rx, Ix, Bx, Lx, Mx, and others) be saved to external memory. Then, the table is further modified to ensure that the `DoSomething()` routine is called as a subroutine from within the `IRQ1 ISR` routine. This adds 250 cycles to the interrupt overhead. The overhead is less than expected because the SHARC super-scalar capability is bought to bear.

Note, there are programming quirks that occur when saving the index and other DAG1 registers. Because of architectural constraints, the DAG1 registers can't be saved directly to the C-stack in data memory using instructions involving DAG1 registers.

The `IRQ1` interrupt is automatically activated by calling the `interrupt()` routine. Calling `interruptf()`, rather than `interrupt()`, changes the lookup table so that a faster interrupt register saving routine will become active. In this case, 60 cycles are added to the interrupt overhead because only the volatile registers are saved and recovered. There is also an even smaller interrupt overhead option, `interrupts()`.

SUMMING UP

This article is directed towards the developer planning to use low-level assembly code in conjunction with C code. I covered a brief overview of the C programming environment for the Analog Devices SHARC 2106X DSP

processor. Many of the similarities and differences of C coding on the 68K processor using the SDS development environment were discussed.

The SHARC has many interesting architectural features designed to optimize DSP algorithms. These include hardware stacks for loop and subroutine handling with zero-overhead circular buffer capability. You must understand the consequences of activating these features from within an assembly code routine called from C.

For more information on the SHARC internal register operations, use "SHARC Navigator LIVE" or contact Talib Alukaidey at T.Alukaidey@herts.ac.uk. Look for my next article in the August issue (121) of *Circuit Cellar* magazine, where Laurence Turner and I will look at how to get the best performance out of your processors when handling DSP algorithms. ☒

Michael Smith is a professor at the University of Calgary, Canada, where he teaches about standard and advanced microprocessors. He spent the last two years creating lessons and laboratories using ten SHARC 21061 development systems obtained from Con Korikis (Analog Devices University Support Program). You may reach him at smithmr@ucalgary.ca.

REFERENCES

- [1] E. Bessinger, "Localization of Sound Using Headphones," M. Sc. Thesis, University of Calgary, Canada
Analog Devices, ADSP-2106X SHARC User's Manual, Analog Devices.
Analog Devices, ADSP-21065L SHARC Technical Reference Guide, Analog Devices.
C Compiler Guide and Reference for the ADSP-2106X Family DSPs. Analog Devices' University Support Program, www.analog.com/dsp.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.