The SHARC in the “C”

Mike Smith

Department of Electrical and Computer Engineering,
University of Calgary,
Alberta, Canada T2N 1N4

Contact Person: M. Smith
Phone: 403-220-6142
Fax: 403-282-6855
Email: smithmr@ucalgary.ca

A version of this article was published by
Circuit Cellar Online Magazine – April 2000
http://www.chipcenter.com/circuitcellar/april00/c0400ms1.htm

Copyright – M. Smith, March 2000
In a recent project we became interested in developing DSP algorithms suitable for producing an improved sound stage for headphones. Listening to music through a standard headset (Figure 1) leaves the listener with the impression that the music is inside the head, a very different feeling to listening to the same music 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 mono sound signal can be altered simply 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, together with room reverberation, can be handled using more extensive DSP techniques such as implementing a series of FIR (finite impulse response) and Comb filters, essentially a cross between FIR and IIR (infinite duration impulse filters). Sampling frequencies will have to be in the range of 44kHz or 48kHz for good sound quality. A specialized architecture DSP processor, or sound card, will be needed for the processing on one sound bite to be completed before the next bite arrives.

For efficient code development, you need to make the appropriate choice of language for the various different 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 our sound-stage project. Standard “C” for the GUI interface 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 advantages of special processor architectural features. Setting up the hardware might require calling an assembly code routine directly from the higher-level language.

The developer must become very 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 CCI article “Some Assembly Required”, (Circuit Cellar 101)

Figure 1. The sound source is perceived in the centre 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 of the head. Further modelling of the audio channel using FIR and IIR filters can move the perceived sound out to a virtual speaker to the front or back of the listener. (From E. Bessinger).
However interfacing between assembly code and “C” functions is very different on the newer DSP chips. The differences arise because of the new architectural features present in the DSP processors. On the positive side, additional data and address registers are available. Specialized hardware becomes available to allow fast switching between subroutines, or for 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 just can’t be handled through a standard “C” programming model.

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

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

Register Comparison

The developer 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 many similarities as there are differences between these processors.

The sixteen 21K data registers (R1 – R15) have essentially the same functionality as eight 68K data registers (D0 – D7). However there are many hidden differences. There is a second, 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 both for integer (R0 – R15) and floating point (F0 – F15) operations.

The 32-bit addition register to register operation

\[ \text{REG0} = \text{REG0} + \text{REG1} \]

is written on the two processors as

\[ \text{ADD.L D1, D0 (68K)} \]
\[ R0 = R0 + R1; \quad (21K) \]

The 21K assembler format has a number of other advantages in addition to its “C-like” characteristics. First there is no hidden “source” register as in the 68K syntax. At the really meant to be added to D1 and stored into D1 or was D1 to be added to D0 and stored in D0?

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

Table 1. Programmer’s model of the main registers on the 68K CISC processor and the 21K SHARC DSP processor.
end of a long day working on the 68K, was D0
Note the use of the semicolon to signal the end of an assembly instruction which 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 of the form

\[ F_0 = F_1 \cdot F_4, F_2 = F_8 + F_{12}; \]

Simultaneous multiplication and addition in a single cycle is available only within certain 21K register banks. This is a limitation of the 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 a schematic of 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, there the similarity stops.

A major 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” data bus (for instructions) and a “data” data bus. Further speed advantages are obtained because the SHARC has a large amount of on-chip, fast access memory.

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” data bus and the “data” data bus.

This architectural feature is handled by special “C” extensions

\[
\text{int dm data[200]} \\
\text{int pm coeff[200]}
\]

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

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 accesses to the *pm* memory using the DAG2 index registers (I8 – I15).

Accesses to arrays stored in memory can be handled using either set of DAG registers because of the SHARC’s on-board memory organization. This can be of considerable advantage to the experienced developer, but a real source of confusion for the naïve since 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!

---

Figure 2. This is a schematic of 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 some interesting problems for the “C” programmer. (Diagram courtesy of Talib Alukaidey)
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] would be programmed on the two processors as follows.

```
MOVE.L #data, A0       68K
MOVE.L #(4*3), D0
MOVE.L (A0, D0), D1

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 3 words on the 21K. You’ll have to get deep into the 21K User Manual to find out the advantages behind having three 21K words sometimes 12 bytes long and, completely transparently, 16 bytes long under other circumstances.

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 the necessity of saving every register to slow memory.

With the SDS 68K compiler there are two volatile data registers, D0 and D1, defined. The 21K registers R1 and R2 essentially have a similar usage. However there are four additional SHARC volatile data register – R0, R4, R8, R12. This strange choice is not arbitrary, but matches with the specific requirements to have volatile registers available to use with the 21K super-scalar operations.

Both 68K and 21K coding conventions allow for 2 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 can’t be used in conjunction with the index registers to step through an array, unlike the volatile 68K data registers. Specific volatile 21K modify registers are needed for this purpose, M4 and M12.

Note there is both a PRE- and a POST- modify memory accessing mode 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 pre-modify operation, with index register I4 left unchanged. In the post-modify operation, the memory at location I4 is fetched, and then the index register is auto-incremented so that I4 = I4 + M4. The postmodify operation can be described in very few bits, which allows parallel post-modify operations to “program” and “data” memory to be described in a single opcode.

Hidden Effect of the SHARC 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 in particular has a big impact when accessing data arrays from within a “C/C++” program.

This effect is hidden in the code sequences given in Listing 1 which uses the “C” default settings of the length and base registers. Given in Listing 1 are the “C” code and assembly language listings to calculate the sum of the first 3 elements in a 10-element array using an auto-incrementing, pointer. The only interesting difference is the need to perform a 21K LOAD-MEMORY-TO-REGISTER operation before performing the ADD. The 21K does not have the complex addressing capabilities of the 68K CISC architecture. Mind you, a 40MHz 21K will perform the fetch-and-add operation in two cycles, whilst a 40MHz 68K would take 16 cycles.

Listing 2 has a hidden kick in its operation, although looking very similar to Listing 1. The key is that the setting of the length (L4) and base register (B4) has established a 2 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).

The point being made is that for standard “C” array handling, the length register Lx associated with index register Ix must remain 0, or be returned to 0. Woe-betide the poor 21K code developer who has to try maintain some “C” code where the Base and Length registers are unintentionally left modified by an interrupt service routine that is infrequently invoked!

Listing 1: Comparison of “C” code, 68K and 21K assembly code to sum the first three elements of a ten-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
    R1 = dm(I4, M4); // Fetch
    R0 = R0 + R1;    // and add
    R1 = dm(I4, M4); // Fetch
    R0 = R0 + R1;    // and add
    // I4 pointing to VALUES[3];
```

Listing 2: There are some “interesting” hidden 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]
```
Passing Parameters to and from subroutines

Both registers and “C”-stack are used for parameter passing on the 68K and the 21K processors. One register is typically designated for returning values from a function – D0 (68K) and R0 (68K) 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 3 subroutine parameters are past 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 assembler for the unwary programmer.

The 21K registers are not general purpose, which means that they can’t be used for both address and data purposes, as can many RISC processor registers. 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 discussed earlier, we’ll ignore the complications arising from situations where subroutines call subroutines.

SHARC Stacks -- hardware and software

Things start getting a little more “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 would be no 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 in another way.

It is no good trying to do the “68K-trick” and pass 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 with no access to the data registers and in addition, it can only hold a few values! That’s not much room to save many parameters on that stack.

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

The approach taken on the SHARC brings back fond memories of the (very early) days developing embedded systems with microprogrammable chips. If you don’t have the required instructions then either add-em, or fake-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 on the SHARC in the “C”. Of course, to avoid nasty surprises, it would be a good idea to 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 touches are to add some “C”-specific instructions to the SHARC instruction set. There is CJUMP used when getting into a “C”-callable subroutine, and RFRAME when leaving. You’ll have to go looking in the SHARC Technical Reference guide as 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 essentially 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”.

The register saving is not to the stack however. There are some nasty hidden things occurring with register R2 in this instruction. However this should not cause a problem if the programmer remembered that R2 is designated as a volatile register and nothing useful should be stored there during a subroutine call!

The RFRAME instruction would be used as part of the return sequence from an assembly language routine back to a “C” calling program. It is essentially the equivalent to the 68K UNLK instruction, including that instruction’s memory access.

Fortunately the instructions necessary to return to a “C” calling function 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 one 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.

\[
\text{#define DO\_MAGIC } \begin{cases} 
I12 = \text{dml(-1, f6)}; \\
\text{JUMP(1, I12) (DB)}; \\
\text{nop}; \\
\text{RFRAME}; 
\end{cases}
\]

**Interrupt Handling**

In this article we are 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 very 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 different from a subroutine. Instead there are 3 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 to interrupt( ) modifies a look-up table used to inform the interrupt handler that IRQ1 interrupts will require the saving of all possible registers (Rx, Ix, Bx, Lx, Mz) 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 you might expect as the SHARC superscalar capability is bought to bear.

Mind you there are some interesting programming quirks 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 interrupt( ), rather than interrupt( ), changes the look-up table so that a faster interrupt register saving routine will become active. In this case 60 cycles are added to the interrupt overhead as only the volatile registers are
saved and recovered. There is also an even smaller interrupt overhead option `interrupts()`.

**Summing up**

This article was directed towards the developer planning to use some low-level assembly code in conjunction with “C” code. We’ve covered a brief overview of the “C” programming environment for the Analog Devices SHARC 2106X DSP processor. Many of the similarities, and differences, with “C” coding on the 68K processor using the SDS development environment were given.

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

**References**


C Compiler Guide and Reference for the ADSP-2106X Family DSPs

Analog Devices University Support Program
www/analog.com/dsp

For more information on the SHARC internal register operations make use of “SHARC Navigator LIVE”
Contact T.Alukaidey@herts.ac.uk


**About the author**

Mike is a professor at the University of Calgary, Canada, where he teaches about standard and advanced microprocessors. He has spent the last two years developing lessons and laboratories using SHARC 21065 development systems obtained from Con Korikis (Analog Devices University Support Program). Mike can be contacted at smithmr@ucalgary.ca.