TMS320 DSP
Number 33
DESIGNER'S NOTEBOOK
TEXAS INSTRUMENTS

Floating Point C Compiler: Tips and Tricks – Part I

Contributed by Karen Baldwin

Design Problem What are some of the tricks of the masters for making the most out of the C Compiler?

Solution 1. Solving the 'C40 Discontinuity Issue with Indirect Calls

The 'C40 has only relative jumps. Therefore PC discontinuities using direct-mode addressing are limited to a 24-bit range. This, coupled with the 'C40 memory map, makes it impossible to directly call a routine in on-chip memory because the compiler uses the direct form of CALL for calls to named functions. (The BR instruction is never used: all branches except returns must be within a single function, so the short conditional form is used.)

You can use indirect calls to call functions anywhere in the address space. You do this by declaring a pointer to a function and then calling via the pointer.

Listing 1: Indirect Function Call

```
int f();              /* function that resides in internal mem */
int (*ptr_to_f)() = f; /* pointer to function f */

main()
{
   (*ptr_to_f)();     /* call function f indirectly */
}
```

2. Making use of Relocatable C code

You can specify different load addresses and run addresses for a section in the linker command file. But you have to write your own loader to move the section to the run address. If using assembly language you can use .label statement to get the load address. Here is an example:

Listing 2: Naming Load and Run Addresses

```
            .global sec_start      ; run address
            .global sec_end
            .global l_sec_start    ; load address
            .global l_sec_end
  sec_start: .label l_sec_start    ; l_sec_start will contain the
                                   ; load address (sec_start = run
                                   ; address)
           ; program code goes here
```

```
sec_end:   .label l_sec_end        ; same explanation
```

Your loader makes use of these labels to write from l_sec_start to sec_start and so on. The loader is a loop that copies the code from one location to the other.

The C version is shown in listing 3.

Listing 3: A Homemade Loader in C

```
/* also asm(“ with .global ..... */
asm(”sec_start .label l_sec_start")   /* same as assembly */
                                      /* version */
/* program code goes here */
 void func(a,b,c)
<local variable declaration>
}
asm(“sec_end .label l_sec_end”)
```

3. Making a C Function Part of a Different Section

The C compiler does not directly specify a section name for the executable assembly code that it generates. It relies on the assembler defaulting the section name to .text. However, it is possible to relocate the executable code from a function into a user-defined section from within the C source.

This is accomplished by placing an asm statement that declares the new section before the actual function definition. The example below uses a macro definition to provide a general solution to defining named sections.

```
#define sect(a) asm(“ .sect ”#a)
sect(“pp”);                       /* Creates .sect “pp” in */
                                  /* asm code */
void func() { }
```

The section name will remain “pp” until changed. If other functions follow this function in the file and their code is not to be included within this section, then reset the section name before the next function definition.