Using Assembly and a TI DSP*

Ricardo Radaelli-Sanchez
Douglas L. Jones

Abstract

The intent of this laboratory module is to familiarize you with the basics of the debugging environment and assembly code for the TI-54x DSP. In this module, you will be working with code that filters signals using FIR (finite impulse response) filters.

1 Processor Overview

The evaluation board assumed for this module at each station includes a Texas-Instruments 320C549 DSP chip which contains the primary core systems: internal memory, the **central processing unit** (**CPU**), and address generation logic. For more information about the DSP chip itself, look at the *TMS320C54x Reference Set, Volume 1: CPU and Peripherals*.

1.1 Memory

The TI TMS320C54x architecture's addressing is divided up into three separate memory spaces: program memory, data memory, and I/O memory. All three memory spaces are 16 bits wide. The data and I/O spaces are both 64K words long; the program memory space is 8M words long, although accessing program memory past 64K words requires special instructions. The TI TMS320C549 DSP itself has 32K words of on-board memory; the DSP evaluation board provides another 256K words of external memory. More information about the DSP's memory map is available in the *Core File Documentation* which should be available from your lab instructor.

The TMS 320C549 DSP uses memory mapped registers; all of the DSP's registers are mapped into the DSP's data memory space between `0000h` and `005Fh`. Special opcodes are provided to speed access to these memory-mapped registers.

1.2 Arithmetic Logic Unit

The central processing unit contains the fundamental components for mathematical and logical operations on data, including a 17x17-bit **multiplier**, 40-bit **adder**, 40-bit **arithmetic logic unit** (**ALU**), and **barrel-shifter**. It also contains necessary registers for performing these operations including two 40-bit **accumulators** (`A` and `B`) and a 16-bit temporary storage register (`T`).

*Version 2.15: Sep 13, 2004 10:43 am +0000
†http://creativecommons.org/licenses/by/1.0

1.3 Address Generation Unit

The address generation logic section is divided into separate program and data address generators and is responsible for fetching program instructions as well as reading and writing data to and from memory. The **program address generation logic** (**PAGEN**) contains (and maintains) the **program counter** (`PC`) register as well as other necessary registers for repeating code. For data memory manipulation, the **data address generation logic** (**DAGEN**) is used and maintains the necessary registers including the auxiliary registers `AR0` .. `AR7` acting as "pointers" to data memory, and control registers such as the **circular buffer register** (`BK`).

2 Basics of Assembly Code

We will introduce the TI TMS320C549 assembly language by showing an example of assembly language code. This code calls the "`core.asm`" file to initialize and control the six-channel surround-sound board, and then copies the signal from channel 1 of the six-channel board's A/D converter to channels 1-3 of the D/A converters, and the signal from channel 2 of the A/D to channels 4-6 of the D/A. Your laboratory assistant should have a similar file for you to use; this code block is provided here for example purposes only.

thru.asm

```
1        .copy "v:\54x\dsplib\core.asm"
2
3     .sect ".text"
4  main
5      ; Your initialization goes here.
6
7  loop
8      ; Wait for a new block of 64 samples to come in
9      WAITDATA
10
11      ; BlockLen = the number of samples that come from WAITDATA (64)
12      stm #BlockLen-1, BRC ; Repeat BlockLen=64 times
13      rptb block-1 ; ...from here to the ‘‘block’’ label
14
15      ld      *AR6,16, A ; Receive ch1
16      mar *+AR6(2)                ; Rcv data is in every other word
17      ld      *AR6,16, B ; Receive ch2
18      mar *+AR6(2)                ; Rcv data is in every other word
19
20      ; Code to process samples goes here.
21
22      sth a, *AR7+ ; Store into output buffer, ch1
23      sth a, *AR7+ ; ch2
24      sth a, *AR7+ ; ch3
25
26      sth b, *AR7+ ; Store into output buffer, ch4
27      sth b, *AR7+ ; ch5
28      sth b, *AR7+ ; ch6
29
30  block
31      b loop
```

Figure 1

2.1 Assembly fields

Take a moment to look at the assembly code listed above. Note that anything following a semicolon is a comment and is ignored during the assembly process; for example, *line 5* contains only a comment. You should also notice that there are primarily three columns in the code. The first column is reserved for **labels** used to refer to a location during the assembly process. The second column contains either **assembly directives** or **instructions**. Directives are commands for the assembler to use during the compilation

process, to assign a label for example, and do not show up in the final compiled code. Instructions on the other hand are what the processor will actually execute. The third column is reserved for operands for the directives or instructions.

2.2 Sections of assembly code

Line 3 of this code contains the directive `.sect ".text"` The `.sect` directive tells the assembler that the following lines are to be placed in memory in the section named in its operand. There are several sections defined in the assembly process; they are used to define locations in memory at which certain pieces of code or data must be placed.

There are two important sections you should pay special attention to: `.text`, in which all of your program code must be placed, and `.data`, in which your program's data will be placed. The `sect ".text"` directive tells the assembler to place the following code or data in program memory starting at `6000h`; the `sect ".data"` directive tells the assembler to place following code or data in data memory starting at `1000h` in memory. [1]

A third important section to be aware of is the `.scratch` section. Data placed in this short section (32 words from `0060h` to `007Fh` in the data memory space; of these, the first 6 locations are used by the core file and cannot be used by your code) can be accessed quickly using opcodes, such as `stm`, which are intended to access the memory-mapped registers.

Although the `thru.asm` code does not include a `.data` section, your final FIR implementation will have such a section to define where to place the states or past input samples in addition to initializing memory for the filter coefficients. In the linking process this data section is assigned starting at the location `1000h` in data memory.

2.3 Core file basics

For this class, you will use the provided `core.asm` file which initializes and configures the DSP and supports the use of the D/A and A/D converters on the six-channel board as well as the serial port on the DSP evaluation board itself. For this lab, you will be using only the D/A and A/D. If your laboratory assistant or lab materials have not told you where to obtain this file, you should ask for assistance at this time.

To describe the core file in more detail, let's analyze the **thru** code in thru.asm (Figure 1: thru.asm).

In this file, *line 1* copies the core file into the source file. All of the files you write for this class should begin with this line.

Line 3 tells the assembler that all of the following code should be placed in the `".text"` section. The `".text"` section is internal program RAM, and begins at `6000h`. Your code will begin somewhat after this, because the core file code appears before your code in the assembled object file.

Line 4 is the label "main." The "core" code jumps to this label once initialization is complete. Any initialization that is required, such as setting address registers, should go between the "main" label and the "loop" label on *line 7*.

Line 8 is a call to the `WAITDATA` macro. Note that this macro overwrites the B accumulator, so don't expect B to stay the same across calls to `WAITDATA`! This macro sets AR6 to the address of the input buffer, and AR7 to the address of the output buffer.

Since `WAITDATA` accumulates a block of 64 samples before it returns, *Lines 12-13* set up a loop that executes for BlockLen times. (The minus one is required, because on the TI DSP a repeat loop runs one more time than the number specified.) *Line 12* sets the `BRC` register, the **Block Repeat Counter**, to 63, and *Line 13* tells the DSP to loop from the next location in memory (the `ld` instruction on *Line 15*) to the location immediately before the label "block." When you use the `rptb` instruction, don't forget to include the -1; otherwise, the DSP will execute an extra instruction at the end of the loop.

[1] Certain data, including the coefficients for the `"firs"` instruction described later, must be placed in the `".text"` section as it is required to be located in program memory.

Lines 15-18 read A/D channel 1 into accumulator "A," and A/D channel 2 into accumulator "B." Note that the input data is stored in alternate memory locations. The instruction `ld *AR6,16,A` loads the memory location pointed to by AR6, the input buffer pointer, into the high bits of A. (The 16 stands for a shift of 16 bits as A is loaded.) The instruction `mar *+AR6(2)` adds two to the value of AR6. The `mar` instruction means **Modify Address Register**, and allows you to do an address-register update without doing any data loads or stores.

Any code to process single samples, including code that you will work with in this lab, should be placed at *Line 20*. The `filter.asm` file you will work with in later modules has its sample-processing code at this point, for example.

Lines 22-28 save the input (contained in accumulators A and B) into the output buffer. The output words are sequential in memory, so no `mar` directives are required; instead, the `*AR7+` addressing mode is used to increment AR7 after each word is written. The `sth` opcode writes the high-order 16 bits of A into memory. The `stl` opcode is used to move the low-order 16 bits of the A accumulator into memory. Be careful which you use; the two operations are easily confused, and this has resulted in many hard-to-find bugs in past students' code.

Line 30 is the "block" label, designating the end of the repeat block. *Line 31* is a branch (`b` opcode) back to the "loop" label, sending the DSP back to before the call to `WAITDATA` to await a new block of 64 samples.

More complete documentation for the core file is available from the *Core File Documentation*, which your laboratory assistant should be able to provide you. This documentation includes information on the DSP memory map, serial port, and extended-memory access instructions.