Realtime DSP:

The TMS320C30 Course

Revision 3

20 February 1994

John Reekie
School of Electrical Engineering
University of Technology, Sydney
PO Box 123
Broadway NSW 2007
Australia

email: johnr@ee.uts.edu.au

Contents

Introduction

This is a course on programming a special type of micro-processor—the DSP, or Digital Signal Processor. This particular course is based on the Texas Instruments TMS320C30 device, which is representative of the Texas Instruments' floating-point DSP family. Other members include the TMS320C31 and TMS320C40. There are several other floating-point DSPs readily available, including the Motorola DSP96002 and the Analog Devices ADSP-21020.

The focus of the course is on *DSP programming*. It does not teach digital signal processing theory, nor does it teach computer programming, since it assumes some knowledge of both. I have attempted to make the course useful to people who have a *need to know*—in other words, for people who need material that explains concisely what they need to know to complete a particular task. I have also tried to structure these notes so that they can continue to be used as reference material.

I expect that the course could be useful as:

— A professional development course for practicing engineers

— An introductory course for thesis and research students working in DSP

They can also form the basis of an introductory undergraduate subject, although additional exercises and tutoring will be needed.

There are some rough edges and unfinished sections in this document. I hope to produce another release that corrects these deficiencies in late 1994.

Assumed knowledge

The course assumes that your have at least some familiarity with the C programming language and some experience with microprocessors and assembler coding (although not TMS320C30 assembler). You do not need any knowledge of the digital signal processing theory to do this course (and the course won't teach you any), although you will definitely need some knowledge about the application area for which to intend to write real-time DSP programs. It is well beyond the scope of this particular course to give you that knowledge.

Incidentally, the breadth of applications in which DSPs can be used is surprisingly large, and growing: modems and faxes, data encryption, data transmission, speech compression, speech recognition and synthesis, image compression and enhancement, robot vision, digital audio, music synthesis, vehicle navigation, seismic and spectral analysis, radar and sonar, servo and motor control, ECG monitoring, auditory aids, prosthetics.

Hardware and software requirements

To make effective use of this course, you will of course need access to Texas Instruments' development tools. The development system we use in our laboratory at the University Technology, Sydney, is the Texas Instruments EVM (Evaluation Module). This is a relatively cheap development system with a single channel of on-board A/D/A conversion. However, there are many other suitable platforms, as well as a TMS320C30 simulator.

The software tools you will need access to are:

- Floating-point DSP C Compiler, version 4.5 or later. (You can probably still make use of these notes if you do not have the C compiler.)
- Floating-point DSP Assembler and Linker, version 4.5 or later.
- TMS320C30 Source Debugger

You will need access to the following manuals:

- *TMS320C3X User's Guide*, literature number SPRU031B.
- *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide*, literature number SPRU034E.
- *TMS320 Floating-Point DSP Assembly Language Tools User's Guide*, literature number SPRU035A.
- *TMS320C3X C Source Debugger*, literature number SPRU053A.

The DSPKit library, and other materials

These notes are only one component of the complete course. Other components include a set of tutorial exercises, and a library called *DSPKit*. DSPKit is a library of TMS320C30 C and assembler code used in our laboratory to simplify the task of building real-time DSP systems.

If you are taking this course at the University of Technology, Sydney, you will already have copies of the exercises and the *DSPKit* documentation.

If you have received these notes over the Internet, then I'm afraid neither the tutorial exercises nor the *DSPKit* documentation or software are available just yet. I hope that I will be able to release the *DSPKit* software during 1994. If you do read these notes, I would be grateful if you would let me know how useful you found them *without* any additional materials!

Terms of use

This document was produced for non-profit educational use. You can use them without charge for your own personal education, or as notes for a non-fee-paying course at a University or other tertiary institution. If you use these notes for these purposes, please send me your comments and suggestions. My electronic mail address is johnr@ee.uts.edu.au.

If you are wish to use or develop these notes for any other purpose, you must obtain permission or a license from me.

Note that you must distribute this document intact, including the title page and copyright notice. The notice on page (iii) explains it all; here is a copy in case it got lost:

The Development Tools

This chapter describes the TMS320C30 development tools. Read only the first section at this stage—the rest is intended as reference material. If you need further information on the development tools, refer to the software manuals.

Overview

The *cl30* program compiles, assembles, and links source files to form an executable file, as illustrated in Figure 1. Each of the tools—compiler, assembler, and linker—can be invoked individually as well as by the *cl30* shell.

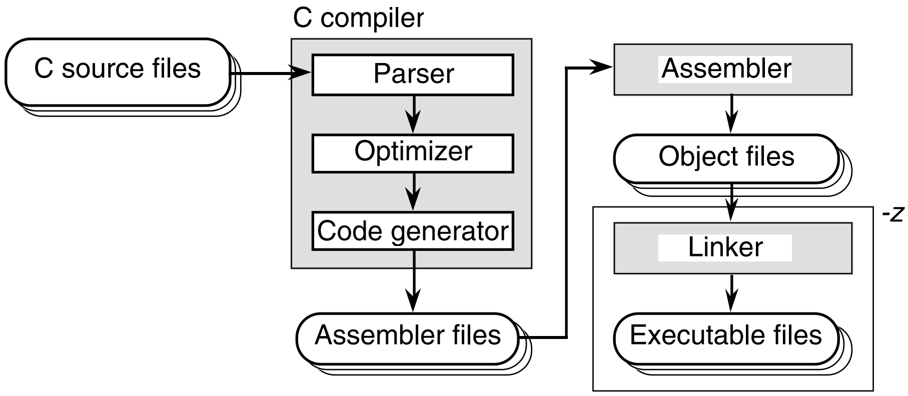


Figure 1. Development tools flow

The compiler shell

Invoking the *cl30* shell

cl30 [–options] [input_files] [-z link_options]

The input files can be C source files (suffix ".c"), assembler files (suffix ".asm"), or object files (suffix ".obj"). The options control the operation of the compiler—the most commonly-used options are listed below.

Compiler Options

Options for symbolic debugging

–g	Enable symbolic debugging. This option must be set if you wish to use the C source debugger on your programs, and so is set as part of the default *DSPKit* configuration.
–a*x*	Supply the –*x* option to the assembler.
–as	Give the assembler the *-s* option, causing it to retain symbolic debugging information. This option must be set to enable symbolic source debugging.

Options for source code manipulation

–c	Compile and assemble without linking.

–n	Compile without assembling or linking. The compiler-generated assembly files are not deleted.
–k	Keep the assembly language file produced by the C compiler, but (unlike the *–n* option) assemble and link the files. Normally, the compiler deletes this file after it assembles it; use this option if you want to examine the code produced by the compiler.
–s	Insert C source code as comments into the compiler-generated assembler files. This option turns on the *-k* option.
–po	Generate pre-processed source files, without compiling, assembling, or linking. This is useful for checking macro expansions and correct inclusion of *#include* files. The generates files have a ".pp" suffix.
–pl	Generate pre-processed source files, but (unlike the *–po* option) compile, assemble, and link the files as well.

Options for optimisation

–o	Enable full optimisation. Different degrees of optimisation can be enabled with the *-o1* and *-o1* options. This may occasionally be useful if the more aggressive options disrupt correct program operation[1]. See the *C Compiler User's Guide* for more information.
–mn	The *–g* optimisation disables certain optimisation because they disrupt debugger operation. The *–mn* optimisation re-enables them. Use this option when, for example, measuring the performance of optimised code.[2]
–mc	Faster float-to-integer conversion. If this option is set, negative numbers are rounded downwards towards infinity, rather than towards zero (as specified by the ANSi standard).
–mm	Enable fast integer multiplies. Integer multiplies in C use only 24×24-bit multiplication if this option is set, rather than the default 32×32-bit multiplies.

Miscellaneous options

–q	Quiet mode. Suppress the printing of functions names—useful for long compiles.
–mb	Select the large memory model. See *More on Addressing* for information on the small and large memory models.
–d	Define a compiler constant. *-d*name has the same effect as having the pro-processor instruction *#define name* at the start of every source file. *-dname* = *defn* has the same effect as having the pro-processor instruction *#define name defn* at the start of every source file.

1 This may happen with (for example), code written without portability in mind, code that does not use *volatile* correctly, code that uses *asm* statements within the C code, and under certain pointer usage conditions (see the *–ma* option in the C Compiler manual).

2 You will have to be a bit careful when using the debugger, though. Because of the extensive code re-arrangement of code performed by the optimiser, break-point are best set within the dis-assembly window rather than in the C source window.

Linker options

The *–z* option specifies the start of the linker options. Usually, this will include a linker command file, and a few additional options. See *Invoking the linker*.

The C_OPTION environment variable

The *C_OPTION* environment variable (in MS-DOS) can be used to set default options that are always used by the compiler. For example, the default *DSPKit* initialisation sets *C_OPTION* to:

–g –as –mn

The assembler

Invoking the assembler

Usually, you will probably just use the *cl30* shell to assemble and link files. Sometimes, however, you may need to invoke the assembler separately. The assembler is invoked as follows:

asm30 [–options] input_file [object_file [listing_file]]

By default, the generated object file has the same name as the input file, but with an ".obj" suffix.

Assembler options

–s	Put all symbols into the object file. Without this option, only global symbols are put into the object file, making debugging more difficult.
–l	Produce a listing file.
–mb	Define the *.BIGMODEL* symbol. By convention, this symbol is tested to conditionally assemble code for large-memory model programs.
–q	Suppress progress information.

Assembler directives

- .text

  Start a new section for program code. The *.text* section is the default section.

  ```
        .text
   ; program code goes here
  ```

- .data

  Start a new assembler data section.

  ```
        .text
   ; data declarations go here
  ```

- .bss *symbol*, *value*

  Reserve *value* words for the variable named *symbol*.

  ```
        .bss array, 100 ; allocate space for 100 words
  ```

- .global *symbol1*, *symbol2*, ...

  Declare the listed symbols as global. List sub-routines and global variables *defined* in this file, **and** external sub-routines and global variables *used* by this file.

```
        .global   sqrt    ; declare external symbol
        .global   poly    ; declare exported symbol
```

- *symbol* .set *value*

  Define an assembler constant.

  ```
  eps .set  1.0e-4    ; error tolerance
  ```

- .word *value1*, *value2*, ...

  Set initialised memory to the listed (integer) values.

  ```
  bitrev  .word 0,4,2,6,1,5,3,7   ; a table of integers
          .word table             ; the address of bitrev
  ```

- .float *value1*, *value2*, ...

  Set initialised memory to the listed (floating-point) values.

  ```
  lookup  .float 1.3, 4.5 ; define table of floats
  ```

- .end

  Signal the end of the assembler file.

  ```
      .end    ; EOF
  ```

The linker

Invoking the linker

The linker is invoked as follows:

lnk30 [–options] object_files

All linker options and object files can also be specified to the *cl30* program following the *–z* option.

Linker options

–o file	Generate an executable file named "*file*.out". If no executable file is specified, a file named "a.out" is generated.
–m file	Generate a map file named "*file*.map." The map file shows where in memory all variables and functions are located.
–q	Quiet run.

Linker command files

Many linker options are put into a linker command file, such as that shown in Figure 2. These options are:

–cr	Link with C conventions. Static data is loaded directly into RAM when the program is loaded.
–c	Link with C conventions. Static data is copied from ROM into RAM during program initialisation.
–heap n	Set the size of the C system heap. The default size is 1024 words.
–stack n	Set the size of the system stack. The default size is 1024 words.
–l libfile	Link with the library *libfile*. Unresolved references are resolved by loading code from the library.

Memory layout

The linker command file is also used to specify the layout of memory in the TMS320C30 system, and how the memory is allocated to the various program "sections." Figure 3. shows how memory from different object files is allocated into memory.

The *MEMORY* keyword introduces the specification of the different regions of memory. Typically, different regions of memory differ in access speed. The version shown in Figure 2 is for the TMS320C30 EVM board.

The SECTIONS keyword specifies which program sections are loaded into which memory regions. The version shown in Figure 2 puts the system stack into one internal RAM block, and the fast system heap (see the documentation for the *DSPMem* memory management software) into the other. All other sections are placed into the main external RAM.[3]

```
/*
 *  vam.cmd
 */

-cr             /* C link with smart loading */

-stack  1024    /* set the stack to this size */
-heap   2048    /* set the main heap to this size */

-l sys.dbg      /* link with DSPKit libraries. Change the */
-l modules.lib  /* library suffix to ".dbg" to link with */
-l evmlib.lib   /* the debugging versions. It is a good */
-l vlib.lib     /* idea to always link with sys.dbg. */
-l clib.lib
-l utility.lib
-l evmrts.lib

MEMORY
{
   VECS:    org = 0      len = 0x40   /* interrupt vectors */
   ROM:     org = 0x40   len = 0x3fc0 /* external memory */
   RAM0:    org = 0x809800  len = 0x400 /* internal RAM 0 */
   RAM1:    org = 0x809c00  len = 0x400 /* internal RAM 1 */
}

SECTIONS
{
   vectors:  {} > VECS  /* interrupt vectors */
   .text:    {} > ROM   /* program code */
   .cinit:   {} > ROM   /* C initialization tables */
   .data:    {} > ROM   /* initialised assembler data */
   .stack:   {} > RAM0  /* system stack */
   .bss:     {} > ROM   /* main variable space */
   .const:   {} > ROM   /* constant storage space */
   .sysmem:  {} > ROM   /* main system heap */
   .fastmem: {} > RAM1  /* fast system heap */
}
```

Figure 2. Typical DSPKit linker command file

3 Additional material on linker sections is required here.

Main memory sections are:

- .text Program code
- .data Initialized data storage
- .bss Uninitialized data storage

Each program file uses a certain amount of each of these sections. The linker joins the sections from each program file into a contiguous block of memory. Each section is loaded into a different portion of the TMS320C30's memory space before program execution.

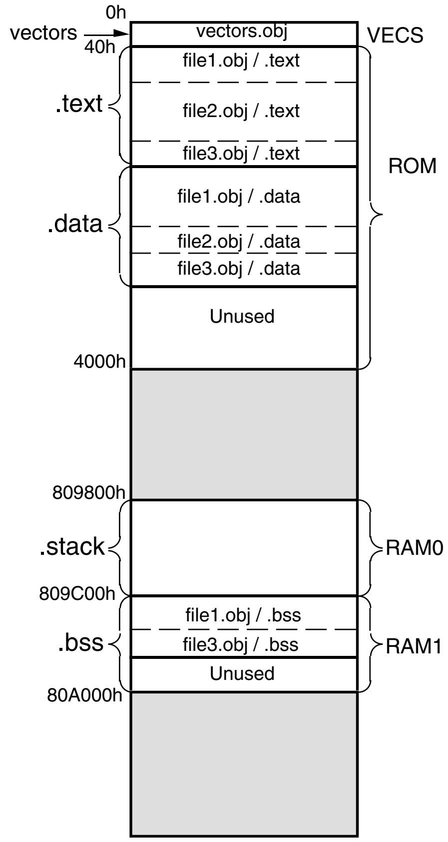


Figure 3. Linker sections

Basic Instructions

This chapter describes the basic instructions of the TMS320C30 instruction set. After completing this chapter, you will be able to write simple assembler programs.

Although the TMS320C30 has a large instruction set, there is a significant amount of commonality between instructions. Understanding these commonalities makes it much easier to understand how to use the instruction set effectively. This chapter has been laid out so as to emphasise these commonalities.

Architecture overview

Figure 4 is a simplified representation of the TMS320C30 CPU architecture (for a more complete picture, see page 2-4 of the *TMS320C3X User's Guide*.

The TMS320C30 contains the following registers:

r0 to r7 Extended-precision registers. These register are the main data manipulation registers, and can hold either 40-bit floating-point numbers or 32-bit integer numbers.

Notice that not only are these registers fed directly from the multiplier and ALU (arithmetic logic unit), but have a direct path back into the multiplier and ALU. This enables a multiplier and ALU operation to be performed every instruction cycle.

ar0 to ar7 The auxiliary registers. These registers are commonly used to hold address values (and are often more conveniently thought of as address registers), but can also be used for general-purpose integer arithmetic. Although they are 32 bits wide, only the lower 24 bits are used for address arithmetic.

The auxiliary registers are connected directly to a pair of integer ALUs, called the ARAUs (auxiliary register arithmetic units). This enables two address values to be updated every instruction cycle.

ir0 and ir1 The index registers. These registers hold values used as operands to the ARAUs.

sp, pc, st Stack pointer, program counter, and status register. These registers are the essential processor control registers.

Others There are a number of registers, used to control other aspects of the processor's operation. These will be covered as they are needed. See pages 2-5 to 2-8 of the *TMS320C3X User's Guide* for an overview of all registers.

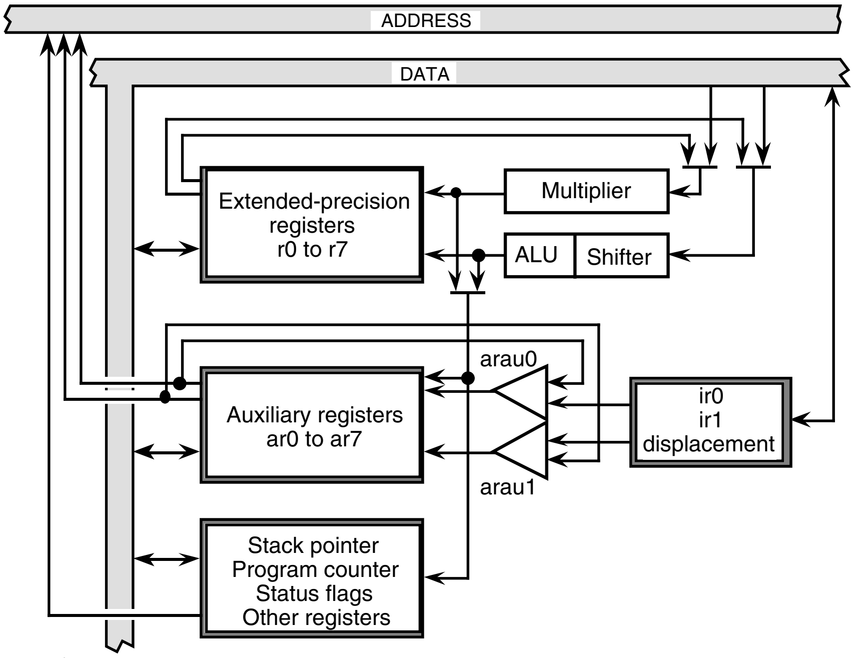


Figure 4. Simplified TMS320C30 Architecture

Addressing modes

The TMS320C30 has four addressing modes for operands. Each of these modes will be illustrated with the *ldi* instruction, which will be used to load a value into the *r0* register. This is only an introduction to these modes—they will be examined in more detail later.

- *Immediate*

  The operand is contained in the instruction itself. For example,

  ```
  ldi     32,r0        ; load the value 32 into r0
  ```

- *Register*

  The operand is a register. For example,

  ```
  ldi     sp,r0        ; copy the sp register into r0
  ```

- *Direct*

  The operand is the address of a variable in memory. For example,

  ```
  ldi     @count,r0    ; load the variable count into r0
  ```

- *Indirect*

  The operand is an auxiliary register containing the address of a variable in memory. This is like dereferencing a pointer in C. For example,

  ```
  ldi     *ar2,r0      ; load the variable pointed to
                       ; by ar2 into r0
  ```

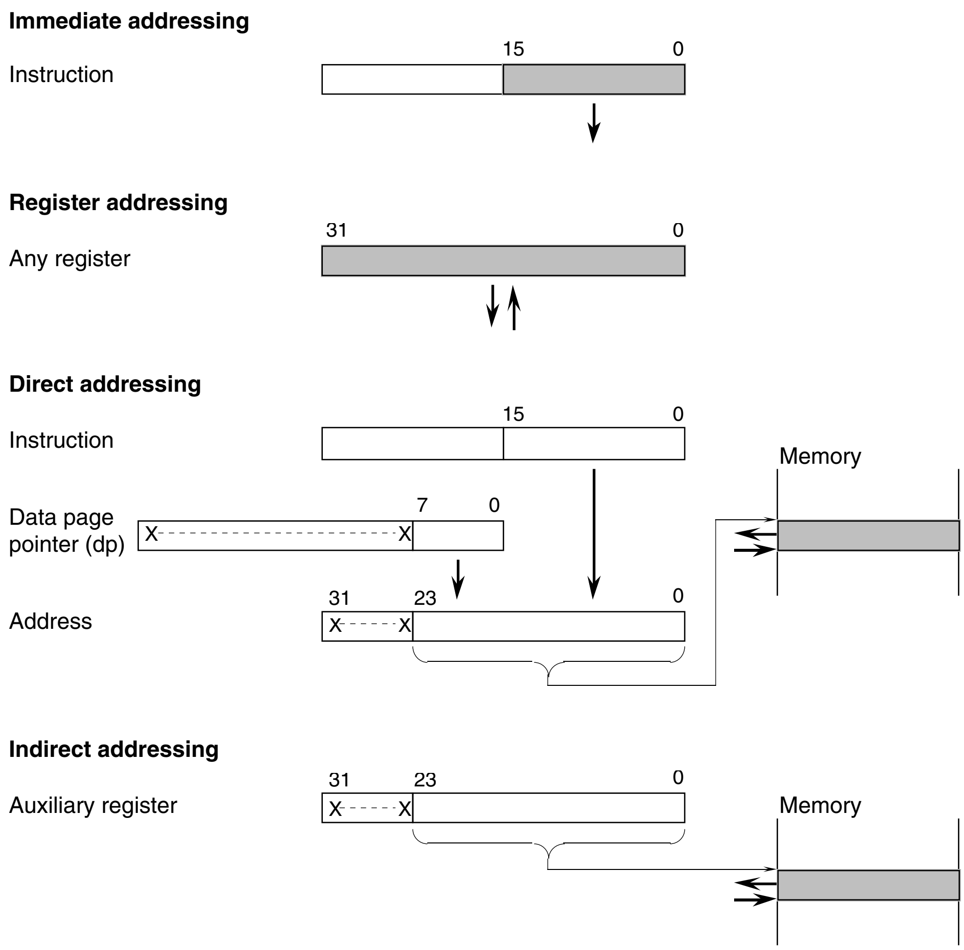


Figure 5. Addressing Modes

Basic data instructions

The data instructions are those that load, store, and operate on data. The other major group of instructions, control flow instructions, will be covered in the next section. Not all data instructions are covered here, just the most commonly-used ones.

Load instructions

Figure 6 illustrates the *load* instruction group. These instructions load data into a register, either from memory or from another register. Immediate data (data contained in the instruction itself) can also be loaded.

Figure 6 also illustrates the format that will be used to describe the addressing mode of groups of instructions. In this case, the source operand can be accessed in any of the four addressing modes: immediate, register, direct, or indirect. The destination register is always a register.

There are two “flavours” of load instruction:

- *ldi*

  Load a 32-bit integer into a register. Register operands can be any register. We have already seen examples of this instruction.

- *ldf*

  Load a floating-point number into a register. Register operands must be one of registers *r0* to *r7*. The lower eight bits of the 40-bit destination register are cleared. For example,

```
ldf     r0,r3       ; copy r0 into r3
ldf     -1.0,r5     ; load -1.0 into r5
ldf     *ar4,r1     ; indirect load off ar4
ldf     @sum,r1     ; Load the sum variable
```

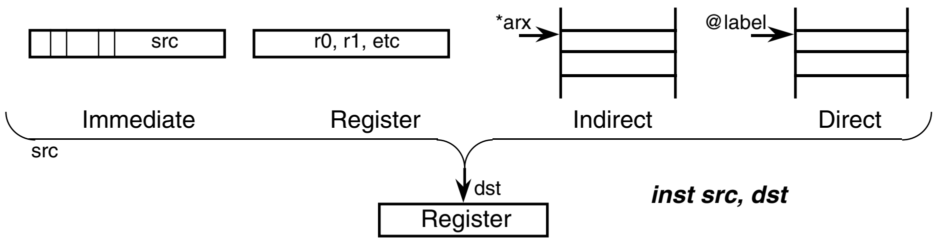


Figure 6. Format of the *load* instructions

Store instructions

Figure 7 illustrates the format of the store instructions. Store instructions write data from a register into memory—the source operand is thus always a register, and the destination operand must be in addressed in either the direct or indirect addressing mode.

- *sti*

  Store a 32-bit integer into memory. The source operand can be any register. For example,

```
sti     r0,*ar0     ; indirect variable store
sti     st,@mask    ; direct variable store
```

- *stf*

  Store a floating-point register into memory. The source operand must be one of registers *r0* to *r7*, and is truncated from a 40-bit float to a 32-bit float. For example,

```
stf     r0,*ar1     ; indirect variable store
stf     r2,@maxval  ; direct variable store
```

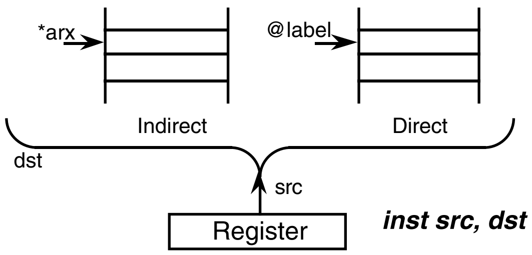


Figure 7. Format of the *store* instructions

Two-operand unary instructions

Unary instructions are those that operate on a single number, such as negation, absolute value, and float-integer conversion. Figure 8 illustrates their format. As usual, only registers *r0* to *r7* can be specified for floating-point instructions. The instructions are:

- *absi* and *absf*

  Take the absolute value of the source operand.
- *negi* and *negf*

  Negate the source operand.
- *float*

  Convert an integer to a float. The destination operand must be one of *r0* to *r7*.
- *fix*

  Convert a float to an integer. The source operand must be one of *r0* to *r7*. This instruction generates an overflow error if the source operand is too large to be represented by a 32-bit integer.

Here are some examples:

```
negi    r0,ar0      ; negate r0 (integer), store in ar0
negf    r3,r2       ; negate r3 (float), store in r2
absi    ar4,ir0     ; absolute value of ar4 in ir0
float   r0,r2       ; r0 (integer) to float, store in r2
fix     r2,ir1      ; r2 (float) to integer, store in ir1
```

If the source operand and the destination operand are the same, then only one operand needs to be specified. For example, the following instructions are the same:

```
absf    r1,r1       ; absolute value of r1
absf    r1          ; absolute value of r1
```

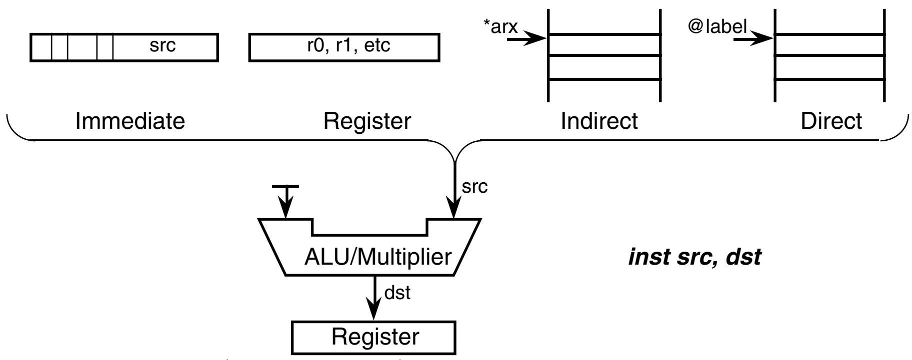


Figure 8. Format of the two-operand unary instructions

Two-operand binary instructions

Binary instructions operate on two numbers and produce a result. The two-operand versions of these instructions operate store the result of the operation in the same register as the second operand. The format of these instructions is illustrated in Figure 9.

This group of instructions includes the common arithmetic and bit-wise logical operations. Arithmetic operations are provided in both integer and floating-point versions, logical operations are integer-only:

- *addi*, *addf*, *mpyi* and *mpyf*

  Add or multiply two numbers.
- *subi* and *subf*

  Subtract the first operand from the second.
- *and*, *or*, and *xor*

  Take the bit-wise *and*, *or*, or *exclusive-or* of two integers.

Here are some examples:

```
addi    1,r0         ; add 1 to r0 (integer)
subi    @length,ar2  ; subtract the length variable
                     ; from ar2
mpyf    r0,r0        ; square r0 (float)
or      0400H,ir0    ; set bit 10 of ir0
```

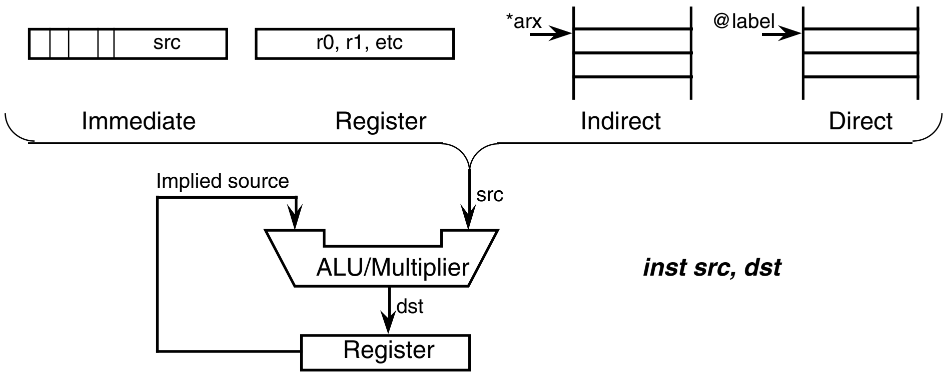


Figure 9. Format of the two-operand binary instructions

Three-operand binary instructions

All of the instructions listed in the previous section also have three-operand versions. In this case, the result of the operand does not have to be stored in the same location as one of the inputs, but can be in some other register.

Figure 10 shows the format of the three-operand instructions. Note that both operands must be register or indirect—*you cannot use immediate values or direct addressing in three-operand instructions*. This is because every instructions must fit into a 32-bit word, and two immediate or direct operands will simply not fit!

Here are some examples:

```
addi3   r0,r1,r2    ; r2 = r0 + r1 (integer)
subf3   r0,*ar0,r1  ; r1 = *ar0 - r0 (float)
mpyf3   r0,r0,r1    ; square r0 and put result in r1
and3    ir0,r0,ar3  ; ar3 = ir0 && r0 (integer)
```

The “3” can be omitted from the instruction mnemonic. This is the style that is used in this course, since I think it clearer and easier to read. For example,

```
addi    r0,r1,r2    ; r2 = r0 + r1
subf    r0,*ar0,r1  ; r1 = *ar0 - r0
```

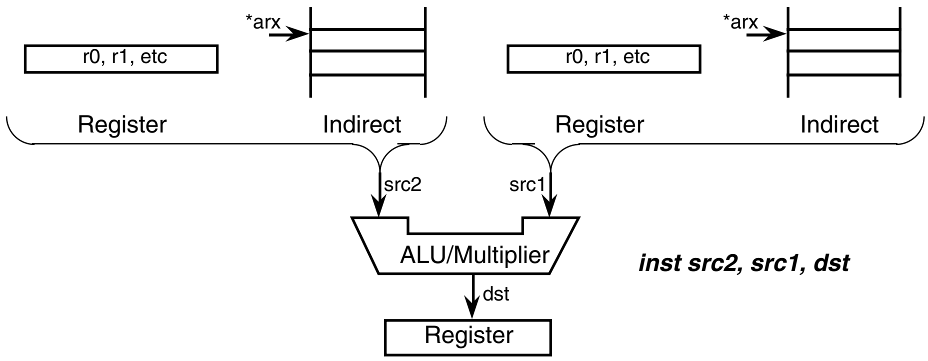


Figure 10. Format of the three-operand binary instructions

Basic control flow instructions

Control flow instructions change program execution. The current program address is contained in the program counter register, or *pc*. In this section, we will cover the basic sub-routine call and branch (*goto*) instructions.

Branch

The branch instruction, *br*, causes program execution to jump to the specified address. The address is stored in the instruction word as 24 bits, so *br* can jump to anywhere in the TMS320C30's 16-megabyte address range.

```
          ...
+-----  br      nextup      ; jump to nextup
|         ...
|         ...
+-> nextup:
          ...
```

Figure 11. The branch instruction

Subroutine call and return

The sub-routine call instruction, *call*, causes execution to branch to a separate subroutine. When it completes, the called subroutine must execute a sub-routine return instruction, *rets*, to return to the calling program.

Subroutine call and return is discussed in depth in the chapter on interfacing C and assembler code. Briefly, however, the *call* instruction causes the address of the instruction just after the *call* (the return address) to be pushed onto the stack. Then, the address of the specified sub-routine is loaded into the program counter, so that program execution continues in that sub-routine. The destination address is stored as 24 bits in the instruction word, so the called sub-routine can be located anywhere in the TMS320C30's 16 megabyte address space.

The *rets* instruction pops the return address off the stack and loads it into the program counter. Thus, the instruction executed after *rets* is the one following the *call.* Because the stack is used, subroutine calls can be nested to an arbitrary depth.

```
Caller                          Callee
...                        +-> process:
...                        |       ...
call    process  ----+             ...
            <-------+              ...
...                        |       ...
...                        +-----  rets
...
```

Figure 12. Subroutine call and return

The decrement-and-branch instruction

The decrement-and-branch instruction, *db*, decrements an auxiliary register and branches to the specified location if the register becomes negative. This is useful for implementing *for*-loops in assembler. The auxiliary register is loaded with the number of times through the loop *less one*, and decremented at the end of the loop. Figure 13 illustrates this use of *db*.

```
    for ( ar0 = n-1; ar0 >= 0; ar0-- )
    {
        ...
        block
        ...
    }
```

```
            ...
            ldi     @n,ar0      ; load the counter into ar0
            subi    1,ar0       ; set to n-1
    +-> loop:
    |       ...
    |       block               ; execute loop body n times
    |       ...
    +-----  db      ar0,loop    ; decrement and branch
            ...
```

Figure 13. *for*-loop using the decrement-and-branch instruction

Note that this loop will always execute at least once.

Conditional Instructions

The TMS320C30 provides a complete set of conditional instructions and codes to access them. These codes and instructions are similar to those in conventional micro-processors, with some additions:

- A flag for arithmetic underflow as well as overflow
- Latched underflow and overflow flags
- Conditional load, call, and return instructions as well as conditional branches

This chapter has two sections. The first lists the conditions, flags, codes and instructions. The second gives examples of the use of conditional instructions. You should read quickly through the first section, and refer to its tables while studying the second section.

Condition flags, codes, and instructions

Condition flags

Table 1 lists the condition flags. These flags are contained in the status register, and are set and cleared according to the result of executing certain instructions. These flags can then be used to select the execution of *conditional* instructions—that is, instructions that execute only if the flags indicate a certain condition.

Table 1. Conditional Flags

Flag	Meaning	Instructions that set
z	Result is zero	Integer and floating-point arithmetic, loads
n	Result is negative	Integer and floating-point arithmetic, loads
c	Result set carry bit	Integer arithmetic, rotate and shift
v	Result overflowed	Integer and floating-point arithmetic
uf	Result underflowed	Floating-point arithmetic
lv	Latched overflow flag	Integer and floating-point arithmetic
luf	Latched underflow flag	Floating-point arithmetic

The *z*, *n*, *c*, *v*, and *uf* flags are set only until execution of the next instruction that affects them. The *lv* and *luf* flags are latched—that is, they must be cleared explicitly by the program.

Instructions that affect condition flags

Table 2 lists the instructions that affect the condition flags.

Table 2. Instructions that affect condition codes

Instruction	Codes affected	Remarks
absf, addf, mpyf, negf, subf	*n, z, v, uf, lv, luf*	Only if the instruction destination is one of R0 to R7. The three-operand versions of these instructions (*addf3* etc), and the parallel arithmetic-and-store versions (*addf* | |*stf* etc) affect the flags in the same way.
cmpf, cmpf3	*n, z, v, uf, lv, luf*	The destination register must (obviously) be one of *r0* to *r7*.
absi, addc, addi, ash, mpyi, negi, subb, subi	*n, z, c, v, lv* *(uf)*	Only if the instruction destination is one of *r0* to *r7*. The three-operand and parallel arithmetic-and-store versions of these instructions affect the flags in the same way. The *uf* flag is always set to zero.
cmpi, cmpi3	*n, z, c, v, lv* *(uf)*	The instruction destination can be any register. The *uf* flag is always set to zero.
lsh, rol, rolc, ror, rorc	*n, z, c* *(v, uf)*	Only if the instruction destination is one of *r0* to *r7*. The three-operand and parallel arithmetic-and-store versions of these instructions affect the flags in the same way. The *v* and *uf* flags are always set to zero.
and, andn, or, xor, float, ldi, ldf, pop, popf	*n, z* *(v, uf)*	Only if the instruction destination is one of *r0* to *r7*. The three-operand and parallel arithmetic-and-store versions of and, andn, os, and xor instructions affect the flags in the same way. Parallel load and load-store instructions (*ldi* | |*ldi*, *ldi* | |*sti* etc) do not affect any flags. The *v* and *uf* flags are always set to zero.
tstb, tstb3	*n, z* *(v, uf)*	The instruction destination can be any register. The *v* and *uf* flags are always set to zero.
ldi | |*ldi, ldi* | |*sti, ldf* | |*ldf, ldf* | |*stf, push, pushf, sti, stf, ldicond, ldf-cond*	*None*	None of these instructions affect *any* flags. They are listed here to remind you so!

Condition codes

Condition codes are attached to certain instructions. For example, the conditional branch instruction is listed below as b*cond*, where *cond* can be any one of the codes listed in Table 3. For example, the “branch-if-zero” instruction is:

```
bz  label       ; branch of last result was zero
```

Table 3 lists the most commonly used condition codes. (For a complete list, see page 10-11 of the *TMS320C3X User's Guide*.)

Table 3. Condition codes

Condition code	Flags	Name	Description
u	Don’t care	Unconditional	The instruction is always executed
lt	*n*	Signed less-then	The *second* operand of a subtract or compare was less than the *first* operand
le	*n* || *z*	Signed less-or-equal	The *second* operand of a subtract or compare was less than or equal to the *first* operand
gt	!*n* && !*z*	Signed greater-than	The *second* operand of a subtract or compare was greater than the *first* operand.
ge	!*n*	Signed greater-or-equal	The *second* operand of a subtract or compare was greater than or equal to the *first* operand.
eq	*z*	Equal	The two operands were equal
ne	!*z*	Not equal	The two operands were not equal
z	*z*	Zero	The result is zero
nz	!*z*	Not zero	The result is not zero
p		Positive	The result is greater than zero
n		Negative	The result is less than zero
nn		Not negative	The result is greater than or equal to zero

Conditional instructions

Table 4 lists all the instructions that can take a condition. Note that none of the conditional instructions can themselves affect the condition flags. For example, the instruction

```
ldf     @val,r3
```

will set the flags according to the value loaded into *r3*. However, a conditional load will not set the flags:

```
ldfz    @val,r3
```

The following instructions are therefore *not* the same:

```
ldf     @val,r3     ; load val and set flags
ldfu    @val,r3     ; load val but do not set flags
```

Also, don’t forget that only loads into *r0* to *r7* set the flags!

Table 4. Conditional Instructions

Instruction	Description
ldi*cond*, ldf*cond*	Load from memory into a register conditionally
b*cond*, b*cond*d	Standard and delayed conditional branches
db*cond*, db*cond*d	Standard and delayed decrement-and-branch
call*cond*, rets*cond*, reti*cond*	Conditional sub-routine call and return
trap*cond*	Conditional software trap

Conditional instruction examples

In this section, we will study how to use the conditional instructions to implement particular C constructs and expressions.

if-then

For this construct, program execution jumps past the code block if the condition is not true. Note that the condition code must be the inverse of what you might expect. That is, the block of instructions is to be executed only if the comparison test shows that $r0 \geq 0.0$. Therefore, the branch is to be executed if this is *not* the case, hence the *blt*.

```
    if ( r0 ≥ 0.0 )
    {
        ...
        block
        ...
    }
```

```
            ...
            cmpf    0.0,r0      ; compare r0 with 0.0
    +-----  blt     label1      ; skip past instructions
    |       ...
    |       block               ; do block of instructions
    |       ...
    +-> label1:
            ...
```

Figure 14. Assembler equivalent of *if-then* construct

if-then-else

In this case, there are two code blocks. The first is skipped if the condition is not true. Note the branch at the end of the first block.

```
    if ( r1 > r0 )
    {
        ...
        block1
        ...
    }
    else
    {
        ...
        block2
        ...
    }
```

```
                ...
                cmpf    r0,r1        ; compare r1 against r0
    +-----      ble     label1       ; branch if r1 ≤ r0
    |           ...
    |           block1               ; do if r1 > r0
    |           ...
  +-+-----      b       label2       ; skip past second block
  | |
  | |
  | +-> label1:
  |             ...
  |             block2               ; do if !(r1 > r0)
  |             ...
  |
  +---> label2:
                    ...
```

Figure 15. Assembler equivalent of *if-then-else* construct

while

A *while*-loop requires a test-and-branch at the beginning of the loop in case the condition is already true. Then, a test-and-branch is required at the end of each loop.

```
    while ( r0 ≥ 0.0 )
    {
        ...
        block
        ...
    }
```

```
                ...
                cmpf    0.0,r0       ; compare r0 with 0.0
  +-------      blt     done         ; branch past loop
  | +-> loop:
  | |           ...
  | |           block                ; do loop body
  | |           ...
  | |           cmpf    0.0,r0       ; end-of-loop test
  | +-----      bge     loop         ; ..and loop again
  +---> done:
                ...
```

Figure 16. Assembler equivalent of *while* loop

while loop with limit counter

The decrement-and-branch instruction can be used to effect to implement *while*-loops that must only execute a certain maximum number of times. For example, suppose one were attempting to evaluate a series approximation function, but wanted to impose a limit on execution time. Figure 17 illustrates this code.

```
    count = 0;
    while (r0 > eps && count < limit)
    {
        make_new_r0;
        count++;
    }
```

```
            ...
            ldi     @_limit,ar0 ; load counter
            subi    1,ar0
  +-------  bn      done        ; exit if limit was zero
  |
  |         ldf     @_eps,r1
  |         cmpf    r0,r1       ; test
  +-------  ble     done        ; already satisfies test
  |
  | +-> loop:
  | |       make_new_r0;
  | |       cmpf    r0,r1
  | +-----  dbgt    ar0,loop    ; loop until limit or eps
  |
  +---> done    ...
```

Figure 17. *while*-loop with count

Select maximum

The TMS320C30's conditional load instructions can sometimes eliminate the need for conditional branches. Since a (non-delayed) branch takes four cycles, this can be very effective. The code in Figure 18 selects the maximum of two numbers.

```
    if (r1 > r2)
        r0 = r1;
    else
        r0 = r2;
```

```
    ...
    ldf     r1,r0       ; preload r0 with r1
    cmpf    r1,r2       ; compare r2 with r1
    ldfge   r2,r0       ; load r2 if greater or eq
    ...
```

Figure 18. Selecting the maximum of two numbers

Calculating signum

Figure 19 uses conditional loads to calculate the signum of a number. Note that this code works only because the conditional loads do not change the condition flags.

```
if (r0 > 0)
    r0 = 1;
else if (r0 < 0)
    r0 = -1;
else
    r0 = 0;
```

```
...
cmpf    0,r0        ; test r0
ldip    1,r0        ; -> 1
ldin    -1,r0       ; -> -1
ldiz    0,r0        ; -> 0
...
```

Figure 19. alculating the signum of a number

Conditional call and return

When working in the C environment, conditional call and return instructions are mostly limited in use to simple functions. Because C-callable functions tend to require additional code before calling (to stack) and before returning (to restore registers and the frame pointer), only the most minimal C-callable function lend themselves to direct use of *call*cond and *rets*cond instructions.

However, in certain situations, good use can be made of these instructions. Figure 20 illustrates a conditional call dependent on the value of a bit in a register:

```
if (r0[14])
    process(r1);
```

```
...
pushf   r1          ; push argument
tstb    14,r0
callnz  _process    ; call this if bit set
subi    1,sp        ; adjust stack
...
```

Figure 20. Conditional call

More On Addressing

This chapter describes the addressing modes of the TMS320C30 in more depth. Some special topics (modulo and bit-reversed addressing) are left until Chapter .

Indirect addressing

As we have already seen, the TMS320C30 allows you to access memory via an address stored in one of *ar0* to *ar7*, analogously to a C pointer. In Figure 4 of Chapter 3, you can see that these registers have two arithmetic units—called *ARAUs*—independent of the main data ALU. The ARAUs give the TMS320C30 very powerful indirect addressing features.

Indexed addressing

An indirect memory access can be indexed relative to an address contained in an auxiliary register. For example, the instruction

```
ldi     *+ar0(3),r0
```

loads the word at (contents of *ar0*) + 3 into *r0*. The value 3 is referred to as the *displacement*: it is contained in eight bits of the instruction word, and can therefore be any value between 0 and 255.

The displacement can be subtracted from the auxiliary register as well as added:

```
addf    *-ar2(1),r2
```

The equivalent C constructions are:

```
x = *(p+3);
y += *(q-1);
```

where *x* and *y* are variables, and *p* and *q* are pointers.

Indexed addressing for structure access

Indexed addressing is an ideal match for access to C-style structures. For example, suppose you had the following C structure (defined in a header file):

```
typedef struct {
    int     size;
    float   seed;
    float   *data;
} MyStruct;
```

To read or modify this structure in an assembler routine, pass a pointer to the structure to the assembler routine:

```
MyStruct mydata;
...
myasmfunc( &mydata );
```

Now, the assembler routine can access the elements of the structure by using indexed addressing off the structure pointer. To ensure portability, assembler directives are used to define the *same* structure in assembler (see the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide*):

```
MyStruct    .struct         ; declare the structure name
size        .int            ; declare the fields
seed        .float
data        .word
sizeofMyStruct .endstruct   ; so we know its size
```

Note that the *data* field is defined as a *word*, not as a float, because it is a pointer, not a floating-point number. On the TMS320C30, the *.word* directive is the same as the *.int* and *.long* directives.

Now, suppose that the *myasmfunc* routine first loads the pointer to *mydata* into register *ar1* (see Chapter 6). Instructions in *myasmfunc* to access *mydata* might include:

```
...
ldi     *+ar1(MyStruct.size),ar2 ; load size into ar2
...
subrf   *+ar1(MyStruct.seed),r0  ; subtract r0 from seed
...
sti     ar0,*+ar1(MyStruct.data) ; update data pointer
...
```

(Note again that the *data* pointer is an *integer*, not a float.)

Note on displacements

The displacement is eight bits for two-operand instructions. Three-operand binary instructions (see Chapter 3) and parallel instructions (see Chapter 7) allow displacements of only zero or one. Thus, the following instruction is illegal:

```
addf    *+ar1(3),r0,r1
```

The index registers

The use of immediate displacements as described in the previous section has some limitations. In particular, displacements

i) are limited to 255,

ii) can only be zero or one for three-operand and parallel instructions, and

iii) must be constant.

These limitations can be overcome with the index registers, *ir0* and *ir1*. Instead of specifying an immediate value, specify one of the index registers. For example,

```
ldf     *+ar2(ir0),r2
addi    *-ar0(ir1),r3,r0
```

Obviously, *ir0* or *ir1* must already contain some meaningful value! In general, the index registers are used more often with post-modify and pre-modify addressing, as described in the following sections, than with indexed addressing.

Post-modify addressing

The TMS320C30 supports C-style pointer incrementing directly in hardware. For example, the pointer update in the C statement

```
x = *p++;
```

is done by the C30 instruction

```
ldf     *ar0++,r0
```

That is, the value at the location pointed to by *ar0* is loaded into *r0*, and *ar0* is then incremented by one (by one of the ARAUs). However, the TMS320C30 assembler can do more than C:

- The increment can be between 0 and 255[4]:

```
addf    *ar0++(16),r2
```

4 Three-operand and parallel instructions only allow an increment of one.

- The increment can be an index register:

```
        sti     r3,ar2++(ir0)
```

The increment can also be negative:

```
    and     *ar1--(2),r1
    stf     r4,*ar0--(ir1)
```

Pre-modify addressing

In C, the increment can also be done *before* the read from or write to the pointed-to memory location:

```
    x = *++p;
```

That is, the pointer is first updated, and the updated value is then used to address memory. The TMS320C30 supports pre-modify addressing in all the same modes as post-modify addressing. For example:

```
    ldf     *++ar4(2),ar0
    addf    *++ar0,r0,r1         ; note increment is one here
    stf     r1,*--ar3(ir0)
```

Usage of post-modify and pre-modify addressing

Much of the speed of the TMS320C30 for signal processing applications comes from effective use of addressing and index registers. For example, a common signal processing operation is to "window" a block of samples:

$$xs_i = xs_i \times w_i \; for \; 0 \leq i < n$$

where *xs* is the input vector and *w* is the window vector. This operation can be implemented using a decrement-and-branch instruction (see Chapter 3), as shown in Figure 21. In this code, it is assumed that *ar0* points to *xs*, *ar1* points to *w*, and *r0* contains the length of the vectors, *n*.

```
            ...
            ldi     r0,ar2       ; load counter
            subi    1,ar2        ;
        loop:
            mpyf    *ar0,*ar1++,r0  ; multiply..
            stf     r0,*ar0++       ; store..
            db      ar2,loop
            ...
```

Figure 21. Windowing using post-modify addressing

In this example, only one of the operands to *mpyf* is incremented; in general, however, both can be incremented, since there are two ARAUs. In Chapter 7, we will see how this loop can be made several times faster.

The memory map

Before we go any further, it would be helpful to learn a little more about the TMS320C30's memory map. Figure 22 shows how the 16-megaword address range is partitioned.

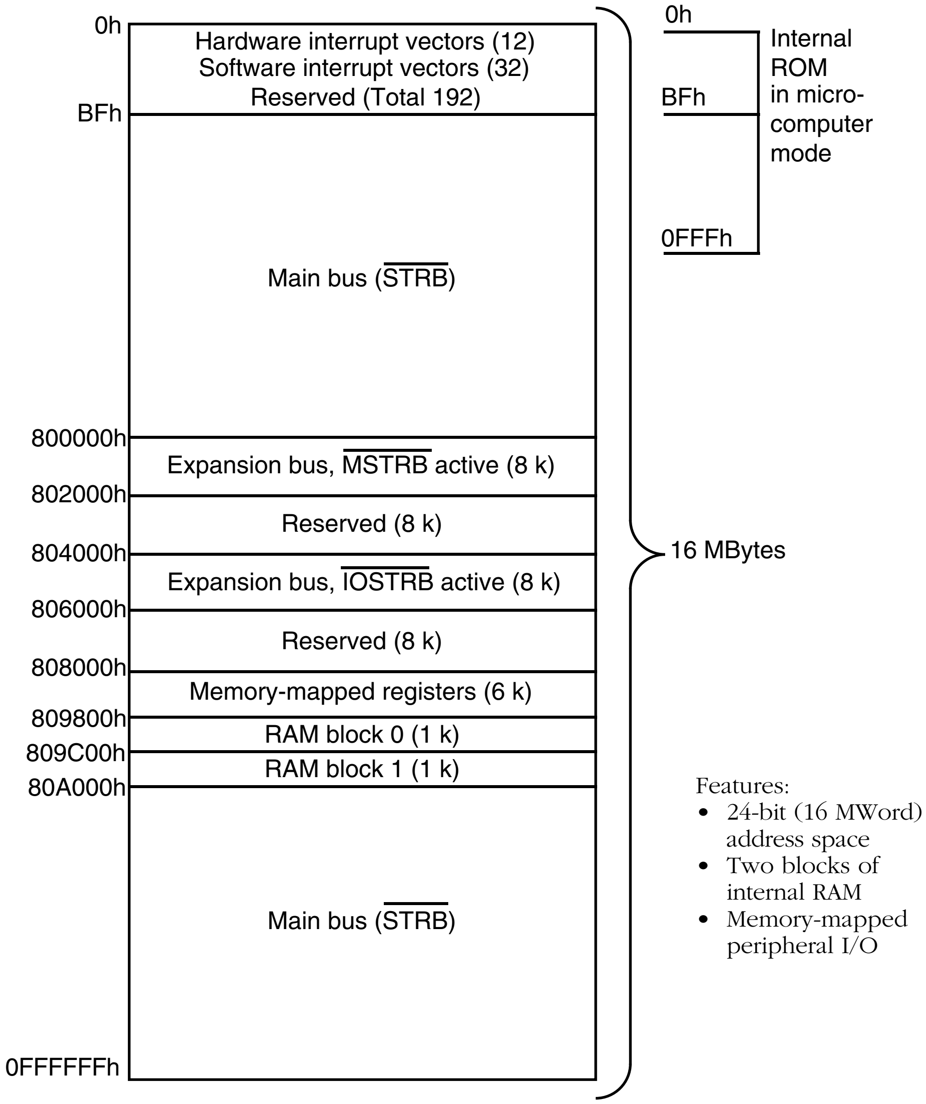


Figure 22. The TMS320C30 memory map

Most of the address space is assigned to the main off-chip bus, and is usually used for general-purpose external memory. Depending on the design of the particular board, this memory may be a combination of static and dynamic memory. Static memory usually has no wait states (that is, no speed penalty), while dynamic memory typically introduces one or two additional cycles per access. Of course, very few systems will actually use the whole 16 megaword address space—the TMS320C30 Evaluation Module, for example, uses only the lowest 16 kilowords.

The TMS320C30 also has the following special memory regions:

- The interrupt vectors (00h to 0BFh)

  These locations contain pointers to interrupt service routines, and are described in Chapter 10.

- The expansion bus (800000h to 802000h and 804000h to 806000h)

  The TMS320C30 (but not the TMS320C31) has a second set of data and address busses, called the expansion bus. This bus is used for external peripheral devices. Because its usage depends only on the particular hardware configuration, we will not deal further with this bus in this course.
- The on-chip peripherals (808000h to 809800h)

  The TMS320C30 has a number of on-chip peripherals, including two serial ports, two timers, and a DMA controller. The control registers for these peripherals, as well as register to control the operation of the main and expansion busses, are contained within this memory region. Programming the peripherals is dealt with in Chapter 9.
- The internal memory blocks (809800h to 80A000h)

  The TMS320C30 has two consecutive blocks of internal memory, of 1k words each. These memory blocks are important in achieving full performance, since they support an instruction read and two data accesses in a single cycle. Ways of allocating internal memory are covered in Chapter 11, while ways of maximising device performance are covered in Chapter 12.

Direct addressing

Direct addressing allows memory locations to be addressed by name. The following code illustrates access to a named memory location:

```
...
.int    count       ; declare the variable
...
ldi     @count,r0   ; read the variable
...
```

This form of addressing can also be used to access C variables. For example, suppose *current_ptr* is defined in a C file as follows:

```
float *current_ptr;
```

An assembler file can access this variable as follows:

```
...
.global _current_ptr    ; declare reference to variable
...
addi    @_current_ptr,r2    ; access variable
...
```

Note the additional underscore in front of the variable name: *all C variables and functions require a prepended underscore when referenced from assembler*.

The data page pointer

For direct addressing, the memory address is contained in the instruction word. Because of instruction encoding limitations, this address can only be 16 bits long, while the TMS320C30 has a 24-bit address range. As shown in Figure 5 of Chapter 3, the TMS320C30 solves this problem by using the lowest 8 bits of a special register, the *data page pointer*, or *dp* register, to specify bits 16 to 23 of the effective address. These eight bits specify one of 256 memory "pages," each of which is 64 kilowords long.

The effective address (that is, the actual address put onto the address bus) is therefore a combination of the *dp* register and the lower 16 bits of the variable's address. *dp* must of course be set to the appropriate value before-hand—this is done with the *ldp* (load data page) instruction:

```
ldp     _current_ptr
addi    @_current_ptr,r2
```

Unless it is known that all variables are contained within the same data page, each and every direct memory access must be preceded by a *ldp* instruction.

The small and large memory models

Obviously, loading *dp* before each direct memory access introduces a rather large overhead. In many systems, all variables can be contained within one data page; *dp* need be loaded only once at the beginning of the program. The TMS320C30 C compiler thus has two “memory models” (similar to the memory models used in compilers for 8086-series microprocessors): large and small.

The small memory model requires that all *directly-addressable* data fits into a single 64 kiloword page. This is the default mode assumed by the C compiler. Assembler code that is linked with C code compiled in the small memory model need not (in fact, must not!) load *dp*, since *dp* is set up by compiler-generated[5] initialisation code.

The large memory model allows directly-addressable data to be located anywhere in memory. The C compiler generates a *ldp* instruction before every direct memory access; all code linked with code compiled in the large memory model must be written and/or compiled for the large memory model. The large memory model is specified with the *–mb* option to the compiler or assembler.

How to cope with both memory models

Obviously, you do not want to write two different versions of every assembler function you write: one for the small memory model, and one for the large memory model. You can write code that caters for both models by using the *.BIGMODEL* assembler flag: this flag is set if code is assembled with the *–mb* option. For example:

```
.if     .BIGMODEL
ldp     _current_ptr
.endif
addi    @_current_ptr,r2
```

Note that many assembler routines will not need to insert these kinds of assembler directives: if a routine receives all its data via its arguments, it may not even need to use the direct memory addressing mode. This type of routine conforms more closely to the spirit of modular programming.

How to avoid the large memory model

Note that the 64 kiloword restriction of the small memory model applies only to directly addressable data—that is, data which is accessed using the direct addressing mode. You can have much more than 64 kwords of data in your program, by using *indirect addressing*. In other words, allocate large memory arrays using the C heap—the standard routines are *malloc()* and *free()*. The pointers to these large arrays will be directly addressable, but the data itself is accessed using indirect addressing. Note however that it is not sufficient that all variables (not including heap-allocated variables) fit into 64 kilowords—t*hey must all fit into one data page.*

See Chapter 11 for more detailed information on allocating and using memory.

---

5 Well, not strictly true: *dp* is set up by the initialisation function *boot.asm* contained in the C run-time support (RTS) library).

C-Assembler Interfacing

Although the TMS320C30 C compiler provides the convenience and high-level abstractions of the C language for real-time DSP, the compiler is less efficient than hand-coded assembler. In particular, time-critical sections of code—such as inner program loops and computationally-intensive algorithms—are usually coded in assembler. Thus, we need to be able to call assembler programs from C, and vice versa.

In most of this course, we will write all assembler functions to be "C-callable." In other words, they conform to the C compiler's function calling interface. C-callable functions are, of course, always produced by the C compiler. Thus, C-callable functions can be written in assembler or C, and called from assembler or C.

As our example, we will use the function *whither*, and show how to call this function from assembler, and how to write a C-callable version of it in assembler. *whither* has the C prototype:

```
float
whither( int size, float seed );
```

The stack

Since the C compiler's default argument-passing mechanism uses the run-time stack, some understanding of the C stack is required before dealing with function interfaces.

The TMS320C30 has a stack exactly like conventional microprocessors. The current position of the stack is indicated by the stack pointer register, *sp*. The stack grows upwards in memory, as illustrated in Figure 23 1.

In the C environment, the stack has four main uses:

i) For holding the return address of called functions

ii) For preserving registers across function calls

iii) For passing arguments to functions

iv) As storage for local variables within functions

In assembler programs, only (i) is mandatory. (ii) to (iv) can be and often are used, depending on the complexity of the program and on function-calling conventions. In this section, we deal with (ii) and (iii).

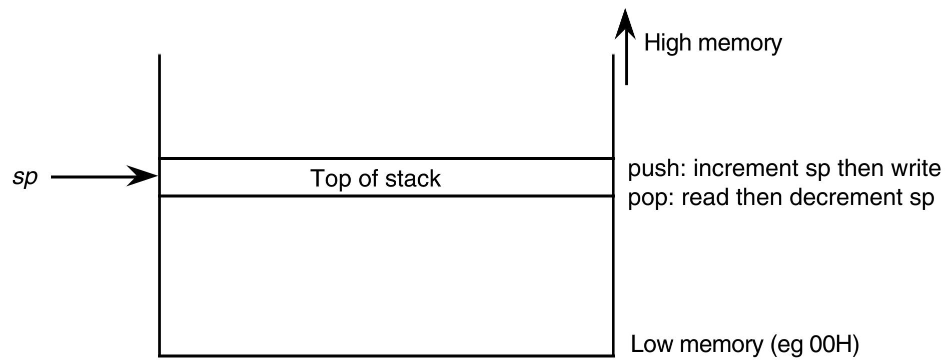


Figure 23. The stack

Push and pop instructions

There are four instructions for storing and loading data to and from the stack: *push*, *pop*, *pushf*, and *popf*. *push* increments the stack pointer, and then writes the specified 32-bit integer register to the top of the stack. *pop* loads a 32-bit integer register with the value at the top of the stack, and then decrements the stack pointer.

pushf and *popf* are exactly the same, except that they store and load 32-bit floating-point values from R0 to R7.

These four instructions are usually used for items (ii) and (iii) above—that is, to preserve registers across function calls, and to pass arguments on the stack.

Preserving registers

Figure 24 illustrates preserving registers. As we will see in the next section, the C compiler requires that all functions preserve certain registers. If, for example, R4 and R5 are used in the body of the function, then they must be saved on the stack on entry to the function, and restored before returning. (For the moment, ignore the code dealing with the *fp* register.)

```
whither:
    push    fp
    ldi     sp,fp
    push    r4      ; save r4 and r5 (as integer)
    push    r5
    ...
    ...             ; body uses r4 and r5
    ...
    pop     r5      ; restore r4 and r5
    pop     r4
    pop     fp
    rets
```

Figure 24. Preserving registers (1)

Stacking function arguments

Figure 25 illustrates how function arguments are stacked. The function *whither* has one integer argument and one floating-point argument—the code shown assumes that they are contained in registers R1 and R0 respectively. After the return from *whither*, the stack pointer is restored to its original value with a subtract instruction.

```
    ...
    pushf   r0      ; second arg
    push    r1      ; first arg
    call    whither ; call it
    subi    2,sp    ; restore stack pointer
    ...
```

Figure 25. Passing stack arguments

Saving 40-bit floats

The *pushf* instruction only saves a 32-bit float. However, R0 to R7 hold 40-bit floats. There are some situations in which the full 40 bits must be saved. In particular:

- The C compiler requires that R6 and R7 are preserved as 40-bit floats.
- Any of R0 to R7 used in an interrupt service routine must be saved as a 40-bit float.

A two-instruction sequence is required to push or pop a 40-bit float. To push R0:

```
    push    r0      ; push mantissa
    pushf   r0      ; push 32-bit float
```

To pop R0:

```
    popf    r0      ; pop 32-bit float
    pop     r0      ; overwrite mantissa
```

Figure 26 shows the code for *whither* if R6 is also used in the function body.

```
whither:
    push    fp
    ldi     sp,fp
    push    r4      ; save r4 and r5 (as integer)
    push    r5
    push    r6      ; save r6 (as float)
    pushf   r6
    ...
    ...             ; body uses r4 and r5
    ...
    popf    r6      ; restore r6
    pop     r6
    pop     r5      ; restore r4 and r5
    pop     r4
    pop     fp
    rets
```

Figure 26. Preserving registers (2)

Setting the stack size and location

The size and location of the memory allocated for use as the stack is set by the linker. The default size of the stack is 1024 words. It can be changed by supplying the "-stack" option to the linker (or after "-z" when invoking the *cl30* shell):

```
    lnk30 main.obj whither.obj c.cmd -o prog.out -stack 2048
```

Usually, however, it is easier to add a line to the linker command file. For example, add the line

```
    -stack = 512
```

to *c.cmd*.

The stack uses the special memory section called "*.stack*" The location of this section is set in the linker command file. If you examine the linker command file *c.cmd*, you will see a line like:

```
    .stack  >   RAM1
```

which puts the stack into RAM block 1. To put the stack into external RAM (which you have to do if you made the stack larger, for example), change this line to:

```
    .stack  >   SRAM
```

The C calling interface

Successfully interfacing C and assembler code is mainly a matter of understanding the function calling convention. The main points concerning the function interface are as follows:

i) The compiler passes arguments on the C stack. Arguments are pushed onto the stack in reverse order—that is, the last argument is pushed first.

   The compiler also has an option to pass arguments in registers, but discussion of this is deferred to a more advanced section.

ii) All arguments are passed as 32-bit values. Integers, floats, and pointers are passed as-is. Structures are pushed onto the stack element-by-element.

iii) All float, integer and pointer results are return in R0. *void* functions do not set R0 to any particular value. Structure results are covered in a later section.

Stack frames and the frame pointer

When a function is called, it creates a *stack frame*. This is simply the term for the area of the stack used for that function call. Register AR3 is reserved by the compiler as a pointer to this area. Understanding the structure of the frame is crucial to writing C-callable functions. To make code easier to read, a mnemonic is usually defined for AR3—the following line should appear new the beginning of your assembler source file:

```
fp  .set ar3    ; use ar3 as the frame pointer
```

The example code

Figure 27 shows the sequence of instructions when a function is called. The constructed stack frame is illustrated in Figure 28. This illustrates the full use of the stack during a function-call sequence, in which the stack frame has three parts:

- Function arguments
- Return address
- Saved frame pointer
- Local variables
- Saved registers

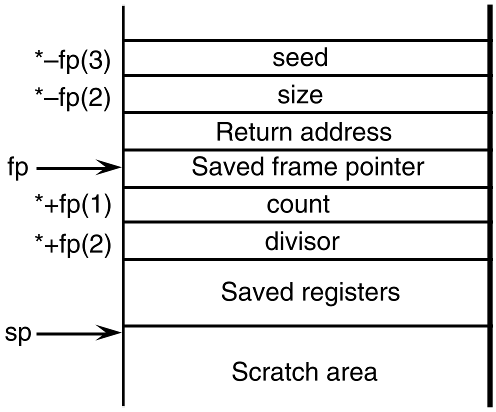
To illustrate the use of the local variable part, we will assume that *whither* has two local variables:

```
float
whither( int size, float seed )
{
    int count;
    float divisor;
    ...
}
```

Many assembler functions do not need all of these fields. A more minimal stack frame that can be used be many assembler functions is described later.

```
        Caller                        Callee
1    ...
2    +-> pushf   r0
3    |   push    r1
4    |   call    whither
5    |                          whither:
6    |                     +-----> push       fp
7    |                     |       ldi        sp,fp
8    |                     | +---> addi       2,sp
9    |                     | | +-> push       r4
10   |                     | | |   push       r5
11   |                     | | |   push       r6
12   |                     | | |   pushf      r6
13   |                     | | |   ...
14   |                     | | |   ldi        *-fp(2),ar0
15   |                     | | |   ldf        *-fp(3),r0
16   |                     | | |   ...
17   |                     | | |   sti        ar0,*+fp(1)
18   |                     | | |   stf        r0,*+fp(2)
19   |                     | | |   ...
20   |                     | | |   popf       r6
21   |                     | | |   pop        r6
22   |                     | | |   pop        r5
23   |                     | | +-> pop        r4
24   |                     | +---> subi       2,sp
25   |                     +-----> pop        fp
26   |                             rets
27   +-> subi    2,sp
28       ...
29       addf    r0,r2
30       ...
```

Figure 27 Calling a C-callable function




```
whither:
      push    fp
      ldi     fp,sp
      push    ...
      ..
      ..      *-fp(2)
      ..      *-fp(3)
      ..
      pop     ....
      pop     fp
      rets
```

Figure 28. The stack frame

The function call sequence

The steps in calling a function are as follows:

Line 1:	Initially, the stack contains the stack frame of the caller. As far as the call to *whither* goes, we only care about what goes on the stack above this point.
Lines 2 and 3:	The caller pushes the arguments onto the stack.
Line 4:	The *call* instruction pushes the return address onto the stack.
Line 6:	The callee saves *ar3* (the old frame pointer) on the stack.

Line 7: The callee copies the stack pointer into the frame pointer register. This is the current frame pointer, and is used to access any variable in the callee's stack frame.

Line 8: The callee reserves space in the local stack frame for local variables.

Lines 9 to 12: The callee pushes registers that must be preserved onto the stack.

Accessing arguments

The arguments to the function are accessed by using indirect addressing off the frame pointer. The first argument to the function is two locations below FP, the second is three locations below, and so on. In Figure 27, the arguments are just loaded into the AR0 and R0 registers:

Line 14: Load the first argument (*size*) into AR0.

Line 15: Load the second argument (*seed*) into R0.

Of course, the arguments can be used directly as operands to other instructions, and over-written with new values. For example, the following code sequence doubles the *size* argument, and stores the new value back into the *size* variable on the stack:

```
ldi     *-fp(2),r0      ; load size
addi    *-fp(2),r0      ; double it
sti     r0,*-fp(2)      ; save new value
```

Accessing local variables

The local variables are also access relative to the frame pointer. They start one location *above* the frame pointer:

Line 17: Store R1 into the first local variable (*count)*.

Line 18: Store R0 into the second local variable (*divisor)*.

As for the function arguments, the local variables can be read and written like any other memory locations by using the register offset addressing mode.

Return sequence

Once the assembler function has finished its processing, the stack frame is deconstructed in the reverse order to which it was built:

Lines 20 to 23: Preserved register are popped off the stack, *in the reverse order to which they were popped.*

Line 24: The stack pointer is adjusted to remove the local variables.

Line 25: The frame pointer register is restored to its original value.

Line 26: The return address is popped of the stack and loaded into the program counter. The next instruction executed is thus the one following the *call* instruction.

Line 27: The caller restores the stack pointer to its initial value.

After execution of line 27, the stack is back in the state it was in before the call to *whither* was initiated.

Important rule

The stack must always be restored to its original state. This is important enough to put it into a box:

Rule: The stack must always be restored to its initial value.

Figure 27 shows how stack operations always occur in pairs. Values are pushed onto the stack at one point, and later are popped off the stack or the stack pointer adjusted to discard them. These pairs must always be properly nested to maintain the stack correctly.

After the return, the caller can use the result of the called function (line 24). For a void function, the caller just ignores the value of R0.

Register usage

Although the example code for *whither* saved several register, many assembler functions will not need to save any registers at all (other than the frame pointer). The C compiler only requires that the registers listed in Table 5 are preserved across function calls. This has three ramifications on your assembler code:

1. A called function needs to save only these registers on the stack (and only if they are used, of course). Many assembler functions only need a few registers—for example, a function that uses R0 to R3, AR0 to AR2, and IR0, needs to save only the frame pointer (AR3) on the stack.
2. A caller can be sure that registers listed in Table 1 will not be corrupted by calls to other functions. Thus, variables that are used frequently throughout the life of a function are usually put into these registers.
3. If a caller wants any values in any other register to be preserved across a function call, then it must save the variable itself with a *push/pop* pair around the function call.

Note that registers R6 and R7 *must* be saved and restored with *push/pushf* and *popf/pop* instruction pairs, as described earlier.

Table 5. Registers preserved across function calls

Register	Compiler usage
r4, r5	Integer register variables
r6, r7	Floating-point register variables
ar3	Frame pointer (*fp*)
ar4 to ar7	Pointer register variables
sp	Stack pointer
dp	Data page pointer (small model only)

A minimal calling interface

The previous calling interface example described used the full stack frame created by the C compiler. When using assembler mainly as a means of ensuring fast execution of critical code, many functions do not use the full stack frame: In particular:

- Hand-coded assembler functions tend to use registers for storing temporary values in. The local variables part of the stack frame is therefore not required, and lines 8 and 24 of Figure 27 are omitted.
- Assembler functions often need only a few registers, and so do not need to preserve any registers other than the frame pointer. Lines 9—12 and 20—23 of Figure 27 are omitted.
- Functions with no arguments may not even need a frame pointer. Lines 6, 7, and 25 of Figure 27 are omitted.

Figure 29 shows *whither* with no local variables and no need to preserve any registers. Figure 30 shows a function with no arguments. This example simply increments the global variable *access_count*.

```
_whither:
    push    fp
    ldi     sp,fp
    ...
    ldi     *-fp(2),ar0
    ldf     *-fp(3),r0
    ...
    pop     fp
    rets
```

Figure 29. Minimal C-callable function

```
_access:
    ldp     _access_count
    ldi     @_access_count,r0
    addi    1,r0
    sti     r0,@_access_count
    rets
```

Figure 30. Even more minimal C-callable function

Advanced Instructions

Although the title of this chapter is “Advanced Instructions,” the instructions presented here are not necessarily any more difficult to use than those presented previously! Rather, these instructions are those that distinguish the instruction set of the TMS320C30 (and other DSP microprocessors) from conventional microprocessors.

The topics covered in this chapter are delayed branches and repeat instructions, which reduce the overhead of branching and looping; parallel instructions, which enable better utilisation of the TMS320C30’s internal resources, and the use of these features together.

Delayed branches

The TMS320C30 pipeline

Previously, we have seen how loops can be built using decrement-and-branch instructions or conditional branches. However, the standard branching instructions are quite expensive: four cycles, whether the branch is taken or not. This is because the TMS320C30 has a four-stage pipeline: Fetch, Decode, Read, and Execute. Each stage is operating on a different instruction—that is, while one instruction is being executed, the following one is having its operands read from registers of memory, and so on.

When a branch instruction is decoded, the TMS320C30 stops reading instructions, because it does not want to inadvertently execute any portion of the instruction following the branch instruction inadvertently. This is called “flushing” the pipeline. When the branch instruction is executed, the program counter is set the appropriate value, and the TMS320C30 starts filling the pipeline again from the instruction at that address. Hence the additional cycles taken to execute a standard branch (whether the branch is taken or not). See Chapter 10 of the TMS320C3X User’s Guide for detailed information on pipeline operation.

Delayed branches

To eliminate this branching overhead, the TMS320C30 has a set of special *delayed branch* instructions. Each of the three types of branching instruction has a delayed version:

- *brd*

  Delayed unconditional branch
- *b*cond*d*

  Delayed conditional branch
- *db*cond*d*

  Delayed decrement-and-branch with condition

Delayed branches execute the three instructions *following* the branch instruction, and *then* the branch is taken. The branch thus effectively takes only one cycle. Figure 31 illustrates the use of a delayed branch. The three instructions following the branch (*mpyf*, *addf*, and *fix*) are executed before the branch is taken.

```
            ...
            brd     proceed      ; delayed branch
            mpyf    r0,r1        ; execute three instructions
            addf    r2,r1
            fix     r1
    +-----  ...             <<- branch occurs here
    |       ...
    +-> proceed:
            ...
```

Figure 31. llustrating the use of a delayed branch

The three instructions following the delayed branch instruction are already in the pipeline when the delayed branch instruction is executed. While these three instructions are being executed, the TMS320C30 continues to fetch and decode instructions from the branch destination. Therefore, these three instructions cannot be any instruction that affects the program counter—that is:

- Any standard or delayed branch
- Subroutine call or return
- Interrupt return
- Repeat instruction
- *trap* or *idle*

Using delayed branches

For assembler coding by hand, the best way of utilising delayed branches may be to write your code using standard branches, then convert standard branches into delayed branches. To take advantage of the delayed branch, move instructions from before the branch instruction to after it; this must be done carefully! Figure 32 illustrates some example assembler code modified this way. Note the following:

- A conditional delayed branch uses the condition codes at the time the instruction is executed, *not the time the branch is actually taken*. When moving instructions to after the branch, you must not affect the condition code that the branch instruction depends on! In this case, instruction (5) must immediately precede the branch instruction.
- Instructions cannot be moved past instructions that use their result. In this case, instructions (1) and (2) cannot be moved; however, (3) and (4) can be, since (5) does not use their result.
- If there are not three instructions that can be moved after the branch instruction, then one or two *nop* instructions must be inserted to make sure that there are three instructions after the delayed branch. In this case, I have inserted one *nop*.

```
        ...
(1)     subf    *--ar1,r0       ; Original code
(2)     mpyf    r0,r1
(3)     addf    r1,r1
(4)     stf     r1,*ar2(3)
(5)     subf    @incr,r0
(6)     bnn     notneg          ; branch if not negative
        ...
```

```
        ...
(1)     subf    *--ar1,r0       ; Code with delayed branch
(2)     mpyf    r0,r1
(5)     subf    @incr,r0
(6')    bnnd    notneg          ; * branch if not negative
(3)     addf    r1,r1           ; |
(4)     stf     r1,*ar2(3)      ; |
(-)     nop                     ; *  (branch occurs)
        ...
```

Figure 32. Converting a standard branch into a delayed branch

Converting unconditional branches (*br*) into delayed branches (*brd*) is generally much easier than converting conditional branches, because there is no problem ensuring that the right condition is tested.

Repeat instructions

The TMS320C30 provides two instructions for repeating a single instruction or a block of instructions. The repeat instructions take four cycles to execute, but there are *no* cycles used for the branch back to the start of the loop. There are thus referred to as "zero-overhead looping" instructions. They can be particularly effective when used with parallel instructions—this topic will be covered later in this chapter.

The TMS320C30 uses three special registers for the repeat instructions:

- *rs*

  The address of the first instruction in the repeat block.
- *re*

  The address of the last instruction in the repeat block.
- *rc*

  The repeat counter.

All of these registers can be used as general-purpose integer registers when not being used for repeat instructions.

Single-instruction repeat

The *rpts* instruction executes the instruction following it the specified number of times. It can be used to sum an array of 32 integers as show in Figure 33.

```
        ...
        ldi     0,r0
        rpts    31              ; loop next instruction
    +-> addi    *ar0++,r0       ; repeat 32 times
        ...
```

Figure 33. Single-instruction repeat

The argument to *rpts* must be one less than the number of times the following instruction is to be repeated. In this example, the operand is an immediate value, but in general, it can use any of the operand addressing modes (immediate, register, direct, or indirect).

Block repeat

The *rptb* instruction repeats a block of instructions; the number of iterations is equal to the value in *rc* plus one. Its operand is the 24-bit address of the last instruction in the block, so the *rc* register must be set up explicitly before executing the *rptb*. The code in Figure 34 sums one complex vector into another. *ar0* and *ar1* point to the vectors; *r0* contains the length of the vectors.

```
            ...
            ldi     r0,rc           ; set up loop counter
            subi    1,rc

            rptb    loop            ; start loop
    +-----> addf    *ar0++,*ar1,r0  ; real part
    |       stf     r0,*ar1++
    |       addf    *ar0++,*ar1,r0  ; imaginary part
    |   loop:
    +-----  stf     r0,*ar1++
            ...
            ...
```

Figure 34. Block repeat

Parallel Instructions

To allow efficient utilisation of the TMS320C30's internal parallelism, its instruction set includes a number of "parallel" instructions, of which there are three groups:

- Parallel multiply and add/subtract
- Parallel load and/or store
- Parallel arithmetic and store

Parallel multiply and add/subtract

This instruction is provided to allow both the internal multiplier and ALU to be used simultaneously. This is in fact the only instruction that allows the TMS320C30 to achieve its peak floating-point rating of 33 MFLOPS (at 33 MHz clock rate).[6]

6 You can see that the claimed floating-point performance of the TMS320C30 in the manufacturer's literature is in fact a very unrealistic figure for almost all applications!

Most DSP chips provide a similar instruction, since a multiply-accumulate operation is the heart of most digital filters.

Figure 35 illustrates the format of the parallel multiply-add/subtract instruction. There are three points to note:

- There must be two register operands from *r0* to *r7* and two indirect operands.
- The multiply destination must be *r0* or *r1*, and the addition destination must be *r2* or *r3*.
- Auxiliary register updates can only be zero (no update), one, *ir0*, or *ir1*.

Here are some examples of this instruction:

```
    mpyf    r0,r1,r0
||  addf    *ar0++,*ar1++(ir0)%,r2

    mpyi    *ar0,r0,r1
||  subi    r1,*ar1,r2
```

In the second example, *r1* is used as both a source and a destination. This is perfectly legal, since all operands are read *before* any results are written. In this case, *r1* is read for use in the *subi* part of the instruction *before* the result of the *mpyi* part is written to *r1*. Thus, you cannot perform, for example, a multiply, and an addition which uses the result of the multiply, in one instruction.

Note that both parts of the instruction must be the same "type"—you cannot mix an integer multiply with a floating-point multiplication, for example.

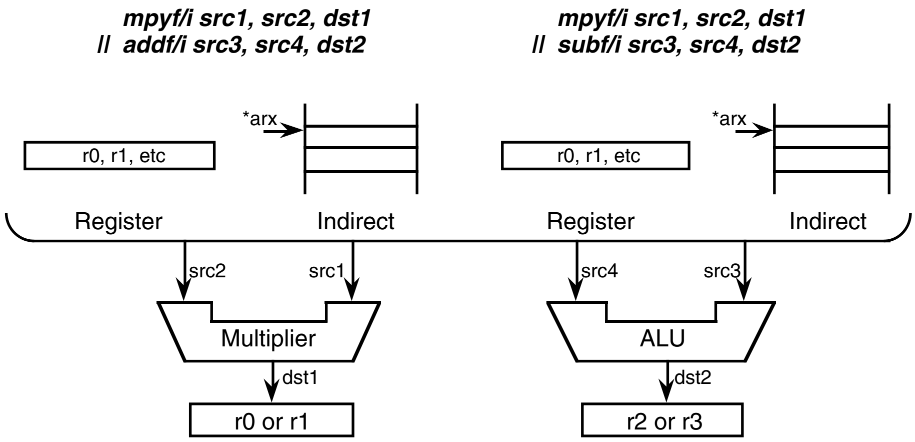


Figure 35. Parallel multiply-add instruction format

Parallel load/store instructions

The TMS320C30 can perform three bus accesses per cycle: one instruction read and two operand reads or writes. The parallel load/store instructions take advantage of this facility, by allowing two loads, two stores, or a load and a store to be performed in one instruction.

Figure 36 illustrates the format of these instructions. Note that:

- There are two indirect memory references and two register references from *r0* to *r7*.
- Auxiliary register updates can only be zero (no update), one, *ir0*, or *ir1*.

Here are some examples of these instructions:

```
    ldf     *ar0,r0          ; load two consecutive vals
||  ldf     *ar0++(ir0),r1

    ldf     *ar0++%,r0       ; copy across buffers
||  stf     r1,*ar1++%

    sti     r4,*ar0          ; store in adjacent posns
||  sti     r4,*+ar0
```

Again, source operands are loaded before destination operands are written, so you cannot load a value and store the same value in one instruction.

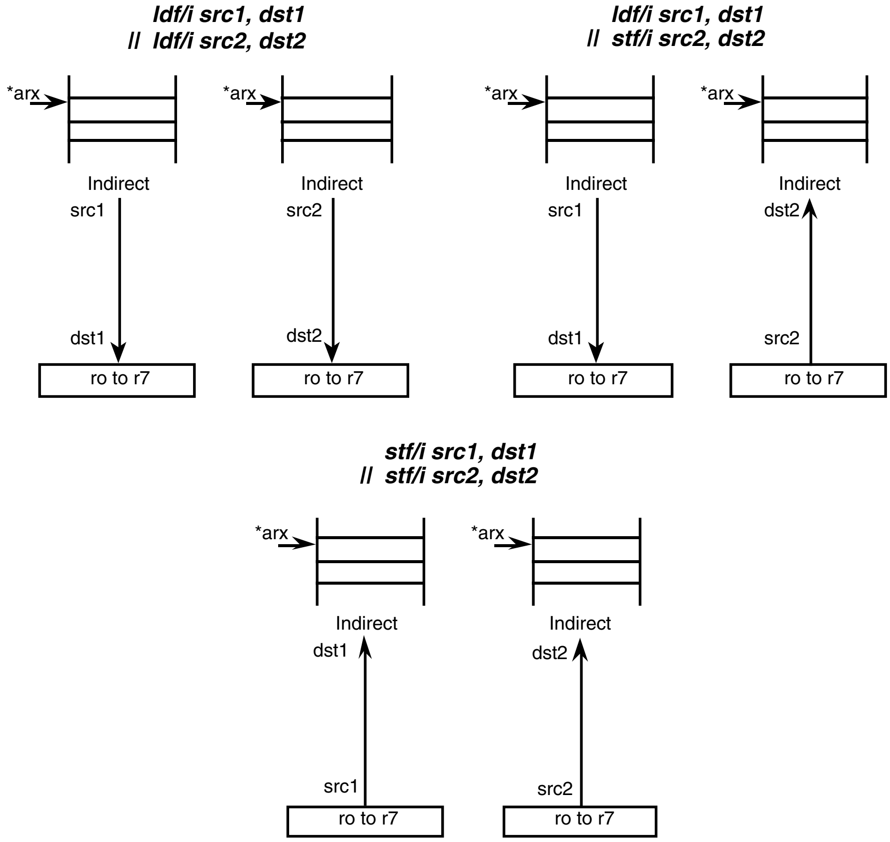


Figure 36. Parallel load/store instruction formats

Parallel arithmetic-store instructions

The TMS320C30 can perform a store in parallel with most of its arithmetic instructions: *absf/i*, *addf/i*, *and*, *ash*, *fix*, *float*, *mpyf/i*, *negf/i*, *not*, *or*, *subf/i*, *xor*.

Figure 37 and Figure 38 illustrate the formats of unary and binary operations in parallel with a store operation. All registers must be one of *r0* to *r7*. Note also that:

- Unary operations have an indirect operand.
- Binary operations have one indirect operand and one register operand.
- Store operations have an indirect destination.
- Auxiliary register updates can only be zero (no update), one, *ir0*, or *ir1*.

Again, the "types" of the instructions must be the same. (*fix* is paralleled with a *sti*, *float* is paralleled with a *stf*.) The result of the arithmetic operation cannot be stored into memory in the same instruction, since the source operands are read before the destination operands are written. Here are some examples of these instructions:

```
      mpyf    r1,*+ar1,r4
||    stf     r2,*ar0

      absi    *ar2++(ir0),r1
||    sti     r1,*ar4++(ir0)

      subf    *--ar2,r1
||    stf     r0,*ar0
```

Notes: (1) The source operands can be written in either order for commutative operations. (2) *r1* will be read before the new value is written. (3) The destination operand can be omitted if it is the same as the second source.

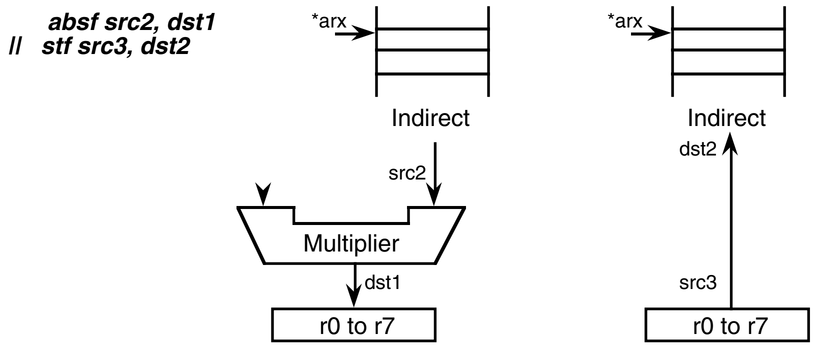


Figure 37 . Parallel unary arithmetic-and-store instruction

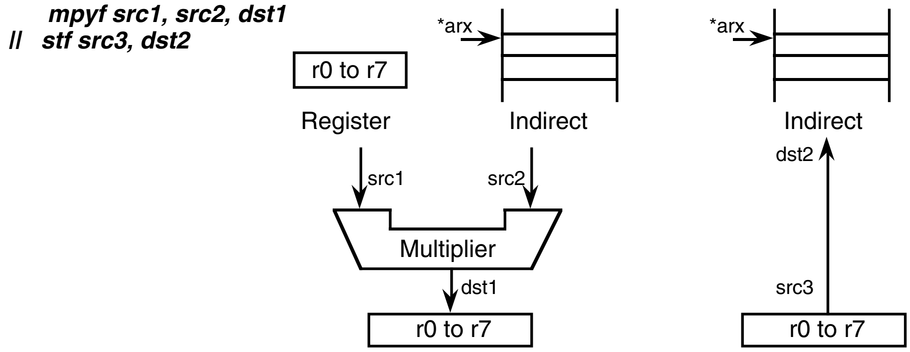


Figure 38. Parallel binary arithmetic-and-store instruction

Parallel instructions and loops

The repeat instructions and parallel instructions must be used together to achieve the performance of which the TMS320C30 is capable. However, there is a small subtlety in the use of parallel instructions in loops that must be understood.

To illustrate the problem, we will use a loop that simply copies data from one location to another. Here is an incorrect version of the loop; *ar0* points to the source, *ar1* to the destination, and *ar2* contains the loop count:

```
      subi    1,ar2        ; set count to 1 less
      rpts    ar2          ; execute loop
      ldi     *ar1++,r0
||    sti     r0,*ar0++
```

The problem is that the *sti* part of the instruction reads *r0* before the *ldi* write to it. Thus, it is storing the value of *r0* from the *previous* time through the loop! The solution is to "prime" the loop by loading the first value, executing the loop *n–1* times, and then storing the last value:

```
        subi    2,ar2           ; set count to 2 less
        ldi     *ar1++,r0       ; prime loop
        rpts    ar2             ; execute loop
        ldi     *ar1++,r0
    ||  sti     r0,*ar0++
        sti     r0,*ar0++       ; clean up loop
```

Note well that the loop counter has *two* subtracted from it before the loop, instead of just one. This means that, if this loop occurs in a situation in which the loop count *might* be one, then this case must be tested for. (To see why, suppose that *ar2* contained 1. Subtracting two gives $2^{31}-1$, since *ar2* is treated as an unsigned number. The loop will then execute 2^{32} times—not what was intended!)

Auxiliary register updates in parallel instructions

I have so far skipped a small point but important point about the use of parallel instructions. All have two indirect operands, which can be incremented or decremented by zero, one, *ir0*, or *ir1*. For example, the code

```
    fix     *ar0++,r0
||  sti     r0,*ar1++
```

could appear in a loop that converts an array of floats to integers. But in this case, the increment is on two different pointers—the integer array is in a different location to the float array. But suppose that we wanted to write the converted data back to its original location: then both parts of the above instruction then use the same auxiliary register, so where should we put the increment operator (++)?

In all parallel instructions, if both indirect operands use the same auxiliary register, the correct addresses for both will be generated, but only one of them updates it. In this particular case, the desired instruction is:

```
    fix     *ar0,r0
||  sti     r0,*ar0++
```

Note that the assembler will give a warning that both indirect operands use the same auxiliary register.

If spare registers are available, you could also do this:

```
    ldi     ar0,ar1
    ..
    ..
    fix     *ar0++,r0
||  sti     r0,*ar1++
```

The operand that updates the auxiliary register (if both are the same) is:

- For parallel arithmetic-and-store, and load-and-store

  The destination operand of the store. The above instruction is an example of this.

- For parallel load and parallel store

  The second indirect operand. For example,

  ```
      ldi     *+ar0,r1
  ||  ldi     *ar0++(ir0),r0
  ```

  which loads two consecutive locations into *r0* and *r1* , and increments the auxiliary register.

- For parallel multiply-and-add/subtract

  The second indirect operand, where the *mpyf/i* part is “before” the *addf/i* or *subf/i* part. For example,

  ```
      mpyf    *ar0,*ar0++,r1
  ||  addf    r0,r1,r2
  ```

which squares the pointed-to location and increments the pointer, as well as adding two registers.

Special Addressing Modes

The TMS320C30 has—in common with other modern DSP devices—two special addressing modes: circular addressing, and bit-reversed addressing. These addressing modes significantly increase device performance for a number of common signal processing algorithms.

Circular addressing

Overview

Circular addressing can be used in a surprising number of algorithms: convolution and correlation, input/output buffers, inter-process communication, Viterbi decoding, and column-wise matrix addressing.

Consider an FIR filter, as illustrated in Figure 39:

$$ys(n) = \sum_{i=0}^{k-1} coeffs(k-i-1)\, xs(n-i)$$

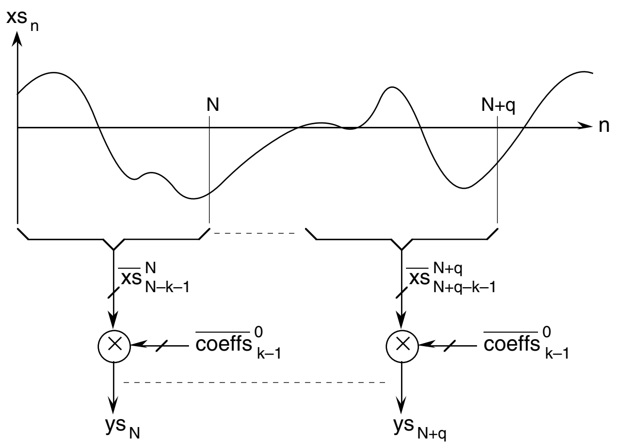


Figure 39. FIR filtering

The implementation of an FIR filter is often visualised as a shift register: each new input sample is shifted into one end of the buffer, and the oldest sample discarded from the other end.

In software, however, the data is stored in an array of memory, rather than in a shift register. One way to implement the shift register is to actually shift all data values down through an array:

```
for each input sample x do
    temp1 = x;
    for i = k-1 downto 0 do
        temp2 = buf[i];
        buf[i] = temp1;
        temp1 = temp2;
```

Figure 40 illustrates this process.

Earlier DSP devices, such as the TMS32010, had a special instruction that combined this data shift with a multiply-accumulate instruction (for implementing filters).

Modern devices, however, provided a more general means of achieving the same end:

```
curpos = 0;
for each input sample x do
    buf[curpos++] = x;
    if (curpos >= k) then
        curpos = 0;
```

Thus, each new input sample is just written into the next location in memory; when the end is reached; the index into the array “wraps” back to zero. It is this wrap-around of the array index (actually implemented as a pointer wrap-around) that is done for free by the circular addressing hardware of the TMS320C30). Figure 41 illustrates this process.

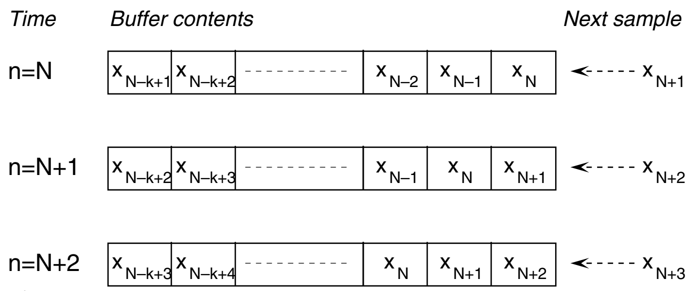


Figure 40 Implementing a shift register (1)

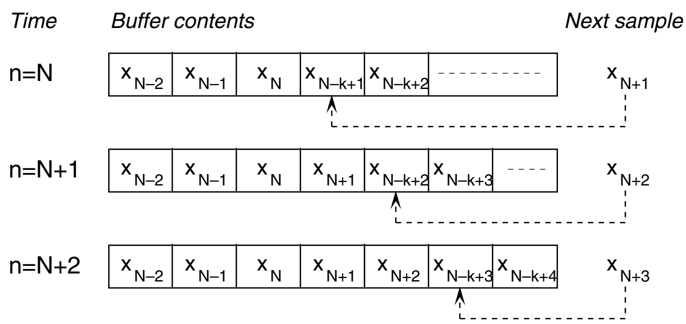


Figure 41. Implementing a shift register (2)

TMS320C30 implementation

To implement a circularly-addressed array of length N in TMS320C30 assembler, you need:

1. A block of memory, starting on a multiple of

   $2^{\left[\left(\log_2 N\right)+1\right]}$, if N is a power of 2

   $2^{\left[ceil\left(\log_2 N\right)\right]}$, otherwise

   For example, a circularly-addressed array of 25 elements must start on a multiple of 32. An array of 64 elements must start on a multiple of 128.[7]

---

7 This is a trap for the unwary: if the block size is a power of two, then it must start at an address that is aligned at *twice* its size.

2. An auxiliary register, which is initialised to point to somewhere within the array.
3. The block size register, *bk*, which must be initialised to the size of the array.

Supposing that the C variable *circbuf* contains the address of a suitable piece of memory for the array. TMS320C30 assembler to write (integer) input values into this array would be:

```
        ldi     @_circbuf,ar4   ; ar4 is circular pointer
loop:
        call    _readval        ; get input sample in r0
        ldi     25,bk           ; array has 25 elements8
        sti     r0,*ar4++%      ; write and circular modify
        b       loop
```

Note that the “%” following the post-increment “++” indicates that the auxiliary register modification is to take place in circular fashion. Post-decrement addressing can also be used, and register can be modified by a value between 0 and 255, or by either of the index registers. Thus, the following instruction is valid:

```
        addi    *ar4++(3)%,*ar5--(ir0)%,r2
```

(As it stands, of course, the above code is useless, since the input data is never processed! This is only an example to illustrate the addressing mode.)

Bit-reversed addressing

Overview

The common forms of the radix-2 FFT algorithm either requires its input data in “scrambled” order, or produces its output data in scrambled order. Figure 42 depicts the decimation-in-time FFT with scrambled output data.

8 *bk* is loaded each time through the loop because the call to *_readval* may corrupt it otherwise. If *_readval* is an assembler routine know not to corrupt *bk*, then *bk* can be loaded once before the loop.

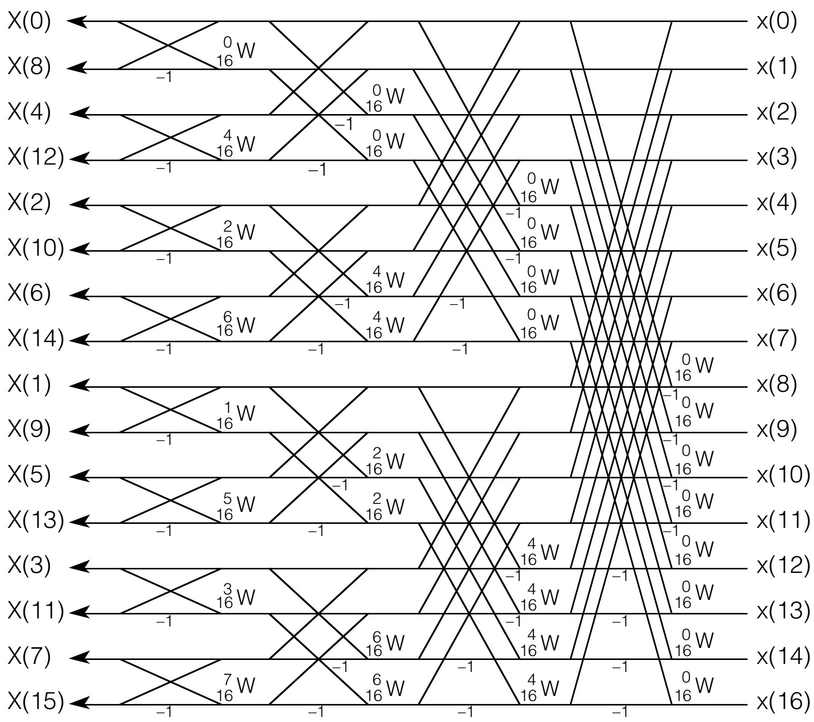


Figure 42. Decimation-in-time FFT Butterfly Diagram

The ordering of the output data is commonly referred to as "bit-reversed" order, because the indexes into the output array can be obtained by reversing the order of bits in each index. Thus, data items 0, 1, 2, 3 etc, are stored in locations 0, 8, 4, 12, and so on. Re-ordering the data array typically consumes as much as 30% of the total FFT time on a conventional processor.

Because the FFT is a very commonly-used signal processing algorithm—both in actuality and as a benchmark for comparing the performance of DSP devices—most DSP devices provide a special "bit-reversed addressing" mode just for implementing the data re-ordering.

TMS320C30 implementation

To implement bit-reversed addressing on an array of length $N=2^n$ in TMS320C30 assembler, you need:

1. A block of memory, starting on a multiple of N. Thus, bit-reversed addressing operates only with power-of-2 sized arrays.
2. An auxiliary register, which is initialised to point to the start of the array.
3. Index register 0, *ir0*, which must be initialised to half the size of the array.

The bit-reversed addressing mode is indicated by a "b" following the post-increment "++" symbol, as shown in the following code fragment from a bit-reversed data copying routine:

```
        rptb    loop
;
        ldf     *ar0++(ir0)b,r0 ; load real, store imaginary
||      stf     r1,*+ar1
;
loop:   ldf     *+ar0,r1        ; load next imag, store real
||      stf     r0,*ar1++(ir1)
```

Note that bit-reversed addressing operates only with *ir0*, and only in post-increment mode. For more information on bit-reversed addressing, see Chapters 5 and 11 of the *TMS320C3x User's Guide*.

Allocating memory arrays

Both the circular and bit-reversed addressing modes require that memory be allocated on power-of-2 boundaries in memory. There are a number of ways to do this:

- Use the linker to locate specially-named memory sections at a pre-determined location.
- Use the linker's *.align* directive to specify that a certain memory area be located on the appropriate boundary.
- Cast a C pointer to a hard-wired address. For example, the C declaration
  ```
  float *buffer = (float *)0x809800;
  ```
  will create a C pointer called buffer, pointing to the start of internal memory.
- Create buffers at the correct location using an appropriate dynamic memory allocator.

The last option is the most flexible. The Texas Instruments C Compiler is now provided with a suitable allocation routine, *balloc()*, with which you can specify the alignment of the block to be allocated.

The *DSPKit* libraries contain a faster and more flexible set of routines for memory allocation—see the *DSPMem* documenation of *DSPKit*. As a simple example of their use, the code to create a circular buffer of 25 elements within the TMS320C30's on-chip memory will include the following code:

```
main()
{
    float *buffer0;
    float *buffer;

    processor_init();                /* initialise system */
    buffer0 = mem_alloc( 25, FastHeap );    /* get memory */
    buffer = buffer0;

    /* use buffer, which is on a 32-word boundary */

    mem_free( buffer0, 25, FastHeap );      /* free mem */
}
```

On-chip Peripherals

The TMS320C30 contains several on-chip peripherals: two serial ports, two programmable timers, and a DMA (direct memory access) controller. This chapter introduces these peripherals and describes how they can be programmed in C. For detailed information on the peripherals, however, you will need to refer to chapter 8 of the *TMS320C3x User's Guide*.

Overview of peripherals

The peripherals are controlled by a number of on-chip registers, located from 808000H to 80807FH in the TMS320C30's address space (although addresses up to 809BFFH are reserved as peripheral address space). These registers can be read and written just like any other memory location. Each peripheral circuit has its own block of memory, as shown in Figure 43.

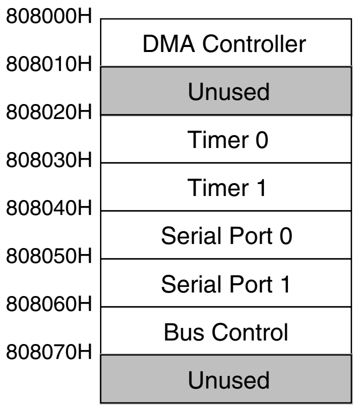


Figure 43. Peripheral Memory Map

In this chapter, we will look only at the timers—the other peripherals are more complex, and are described in more detail in following chapters.

The timers

Each timer control block in fact only contains three registers, as shown in Figure 44. The "global control" register contains mode select fields and status flags. The counter register contains the current timer count; it is incremented at half the clock rate: if the clock rate is 30 MHz, for example (as it is in the EVM board), the count register will be incremented at 15 MHz. The period register contains the timer period; a timer interrupt is generated and the timer counter reset to zero when the value in the counter register reaches the value in the period register.

See pages 8-2 to 8-11 of the TMS320C3x User's Guide for information on the operation of the timer control bits.

Timer 0	Timer 1	
808020H	808030H	Global control
808021H	808031H	Reserved
808022H	808032H	Reserved
808023H	808033H	Reserved
808024H	808034H	Counter
808025H	808035H	Reserved
808026H	808036H	Reserved
808027H	808037H	Reserved
808028H	808038H	Period
808029H	808039H	Reserved
80802AH	80803AH	Reserved
80802BH	80803BH	Reserved
80802CH	80803CH	Reserved
80802DH	80803DH	Reserved
80802EH	80803EH	Reserved
80802FH	80803FH	Reserved

Figure 44. Timer control block

Accessing hardware memory locations in C

There are a number of ways in which the timer registers can be accessed from C. Suppose we wish to write a variable, say *timer_period*, into the period register of Timer 1. Here is one way:

```
*(unsigned int *)0x808038 = timer_period;
```

This C statement creates a pointer to the timer register (which is, after all, just a memory location as far as the code is concerned), and uses the pointer to write to the register.

This method is both error-prone and difficult to read; it is easy to inadvertently type say 0x808083h as the address, for example. One could use macros to define memory locations; for example,

```
#define TIMER_1_PERIOD (*(unsigned int *)0x808038)
...
TIMER_1_PERIOD = timer_period;
```

However, you end up defining a lot of macros! A (better, I think) method is to define a C structure that mimics the memory layout of the timer control block. In the *DSPKit* file *periph.h*, for example, you will find the following structure definition:

```
typedef struct
{
    timer_control_t control;
    unsigned        rsv1[3];

    unsigned        counter;
    unsigned        rsv2[3];

    unsigned        period;
    unsigned        rsv3[7];
} timer_t;
```

(*timer_control_t* is defined elsewhere in that file.)

Now we can access the timers as follows:

```
#define TIMER0 ((timer_t *)0x808020)
#define TIMER1 ((timer_t *)0x808030)
    ...
    TIMER0->period = timer_period;
```

This can be taken a step further: we can define a single C structure to mimic the whole peripheral memory space! In *periph.h*, you can find the following structure definition:

```
typedef struct
{
    dma_t       dma;                /* DMA controller */
    unsigned    rsv1[16];
    timer_t     timers[2];          /* Two timers */
    port_t      ports[2];           /* Two serial ports */
    bus_t       bus;                /* Bus control */
    unsigned    rsv2[16];
} peripherals_t;
```

If you compare this structure with Figure 43, you will see the correspondence immediately. In order to access the peripheral memory space, the file *periph.c* actually defines a pointer to this memory (rather than just using a constant pointer as before):[9]

```
    volatile peripherals_t
    *peripherals_p = (volatile peripherals_t *)0x808000;
```

periph.h also defines this macro:

```
    #define peripherals (*peripherals_p)
```

The period register of timer 1 is thus accessed by

```
    peripherals.timers[1].period = timer_period;
```

Using the definitions in the file *periph.h*, you can read or write any peripheral register in this way.

9 The *volatile* keyword is used to prevent the optimiser from breaking your code. Suppose you were polling a flag in the peripheral address space, testing for whether it had been set yet. The optimiser would see this code as a good opportunity to remove unneeded code, and would load the flag in question into a register once, *before* the loop! As far as the optimiser can tell, the location you are repeatedly testing never changes, so there is no need to load it each time through the loop. The *volatile* keyword informs the compiler that the variable you are examining in fact *can* be changed by some external agent (a peripheral circuit, in this case), so it must be read afresh each time through the loop.

Accessing structure- and bit-fields

The previous section showed how to access whole registers in the peripheral memory space. However, many registers contain a large number of *bit-fields*. For example, Figure 45 shows the structure of one of the timer "global control" registers.

Bit	12—31	11	10	9	8	7	6	5	4	3	2	1	0
	XXXXXXXXXXXX	TSTAT	INV	CLKSRC	C/P	HLD	GO	xx	xx	DATIN	DATOUT	I/O	FUNC

Figure 45. Bit-fields in the timer global control register

Suppose we wished to set bit 6 (the *GO* bit), without changing any other bits. In assembler, we have to load the word from memory, set the bit, and write the word back again. For example,

```
    ldp     808030h
    ldi     @808030h,r0
    or      40h,r0
    sti     r0,@808030h
```

We could mimic this in C if we wanted:

```
    temp = peripherals.timers[1].control;
    temp |= 0x40;
    peripherals.timers[1].control = temp;
```

However, C also allows you to define *bit-fields* in structures—that is, structure elements that are only a given number of bits in size. To illustrate, *periph.h* defines the *timer_control_t* structure as follows:

```
typedef struct
{
    unsigned    func       : 1 ;
    unsigned    io         : 1 ;
    unsigned    datout     : 1 ;
    unsigned    datin      : 1 ;
    unsigned    unused1    : 2 ;
    unsigned    go         : 1 ;
    unsigned    hld        : 1 ;
    unsigned    cp         : 1 ;
    unsigned    clksrc     : 1 ;
    unsigned    inv        : 1 ;
    unsigned    tstat      : 1 ;
    unsigned    rsv1       : 4 ;
    unsigned    rsv2       : 16;
} timer_control_t;
```

If you compare this with Figure 45, you will see how the elements of this structure match the bit fields contained in the timer control register.[10] Now, the *go* bit can be set by:

```
    peripherals.timers[1].control.go = 1;
```

Clearly, any of the bits in any peripheral control register can be read or written this way. To close this chapter, the following code fragment initialises and starts timer 1:

[10] Note: The relative order of the declarations in *timer_control_t* and the control bits is completely machine-dependent.

```
/*
 *  Set all bits of control register to zero.
 */
*(unsigned *)&peripherals.timers[1].control = 0;

/*
 *  Set the period register and control bits
 */
peripherals.timers[1].period = timer_period;

peripherals.timers[1].control.clksrc = 1;   /* internal clk */
peripherals.timers[1].control.hld = 1;      /* release */
peripherals.timers[1].control.func = 1;     /* TCLK is o/p */
peripherals.timers[1].control.go = 1;       /* start */
```

Interrupts

Interrupts are used to allow a processor to respond to external events; in the TMS320C30 environment, these events can include a timer time-out, data being received or transmitted by a serial port, completion of a DMA transfer, or a command or data from a controlling host computer.

This chapter describes the TMS320C30 interrupt structure, and how to set up and use interrupts using first the *ints* module of the *DSPKit* library, and then assembler code.

This chapter is still incomplete—contact the author for the current version.

C30 interrupt structure

Interrupt sources

The TMS320C30 has twelve hardware interrupt sources, including a hardware reset. The names of these interrupts defined by the *DSPKit* library are:

Reset

Device reset interrupt. This occurs when the TMS320C30 is powered up. When you start running a program in the TMS320C30 Source Debugger, the program executes as though a hardware reset had occurred.

External0, *External1*, *External2*, and *External3*

External interrupt sources. These are triggered by four pins of the TMS320C30, and are used for communication with the host computer in the EVM board. In other systems, the use of these interrupts is determined by the design of the external circuitry.

Transmit0, *Receive0*

Serial port 0 interrupts. these interrupts are triggered when serial port 0 has finished transmitting a word or has received a word.

Transmit1, *Receive1*

Serial port 1 interrupts.

Timer0, *Timer1*

Timer interrupts. Each of the on-board timers can generate an interrupt when it times out.

DMA

DMA (direct memory access) controller interrupt. This is triggered by completion of a DMA transfer.

Interrupt vectors

The first twelve locations of memory are occupied by the *hardware interrupt vectors*. These location contain the address of an *interrupt service routine* (ISR), which is to process the corresponding interrupt when it occurs. In the TMS320C3x User's Guide, these locations are shown as in Figure 46:

00h	RESET
01h	INT0
02h	INT1
03h	INT2
04h	INT3
05h	XINT0
06h	RINT0
07h	XINT1
08h	RINT1
09h	TINT0
0Ah	TINT1
00Bh	DINT

Figure 46. TMS320C30 Interrupt Vectors

(Note that the names of these interrupts defined by *DSPKit* are different to the names in the *User's Guide*.)

Interrupts and polling

Basic interrupt operation

Interrupts occur *asynchronously* of the program being executed by the CPU—in other words, they can occur at any time. Let us suppose that the CPU is currently performing some task, such as (say) calculating the power spectrum of a portion of a signal. While it is doing so, another input sample from the A/D converter arrives at serial port 0 (which we assume is connected to the converter).

Since serial port 0 has just received a data word, it triggers the *Receive0* interrupt. Assuming that interrupts are currently enabled, the TMS320C30 performs the following:

1. It disables any further interrupts.

   Although we haven't come to it yet, the *GIE* bit in the status register, *ST*, is cleared.

2. It pushes the current program counter plus one onto the stack.

   For example, suppose that the TMS320C30 is executing the *ldi* instruction in the following code fragment:

   ```
   addi    *ar0++,r0       ; address 0158H
   ldi     *ar1,r1         ; address 0159H
   mpyi    r0,r1,r2        ; address 015AH
   ```

   The address 015A (hexadecimal) is loaded onto the stack.

3. It loads the value from the corresponding interrupt vector into the program counter.

   In this case, it loads the word at memory address 06H into the program counter, so that execution continues from that address. Since the location 06H should contain the address of the interrupt service routine (ISR) for receive interrupt 0, this ISR will now be executed.

4. The ISR must terminate with a *reti* instruction, instead of a *rets* instruction. This instruction pops the top of the stack into the program counter, and sets the *GIE* bit.

   In our example, the CPU will continue executing from the *mpyi* instruction, as though nothing had ever happened between the *ldi* and the *mpyi* instructions

Characteristics of interrupt service routines.

As you can see, calling an ISR is very similar to calling a normal subroutine. There are two key differences:

- An ISR must preserve *every* register, *including the status register (ST)*. This ensures that the main program functions correctly even though an ISR can be executed between any two instructions. You will recall that normal subroutines only need to preserve certain registers (see *C-Assembler Interfacing*).
- An ISR must end with a *reti* instruction, so that interrupts are re-enabled after the ISR completes.

Polling

Polling is an alternative method of responding to external events, which is simpler to program than interrupts, and can sometimes even be more efficient. With interrupts, the CPU executes its main program without regard for external events; the external events force the CPU to respond to them by interrupting it. With polling, however, the CPU must "poll" for external events.

In the case of the data received on serial port 0, the CPU will repeatedly test a bit in the serial port control registers[11] to check whether a word has been received. If it has, the CPU takes appropriate action. If it has not, the CPU either does something useful for a short while, or just waits a short while (perhaps as little as one instruction) before trying again.

A silly analogy may help understand the difference. (Acting out the following scenarios might help appreciate the analogy!)

Suppose I decided that one of my students has to make me coffee periodically in order to pass this course. Using an interrupt-driven protocol, the student would interrupt me when the coffee was ready; I would handle the interrupt by taking the coffee and drinking it. The sequence of events looks like this:

Me	Student
"Make me a cup of coffee!"	
	Starts making coffee
(Busy working)	(Busy making coffee)
(Some time passes....)	
	"Your coffee is ready"
Stops working, and takes coffee from student.	Hands me coffee
Drinks coffee, and then goes back to work	Sits around waiting for next command.

11 Or the interrupt flag (*if*) register.

Using a polling protocol, however, I would periodically ask the student whether the coffee was ready yet. Finally, it would be, and I would then be able to drink it. The sequence of events looks like this:

Me	**Student**
"Make me a cup of coffee!"	
	Starts making coffee
(Does a little work)	(Busy making coffee)
"Is the coffee ready yet?"	"No"
(Does a little work)	(Busy making coffee)
"Is the coffee ready yet?"	"No"
(Does a little work)	(Busy making coffee)
"Is the coffee ready yet?"	"Yes"
Stops working, and takes coffee from student.	Hands me coffee
Drinks coffee, and goes back to work	Sits around waiting for next command.

Accessing and controlling interrupts in C

To simplify the use of interrupts, *DSPKit* provides a utility module called *ints*, which contains a number of C-callable routines.

Initialising interrupts

The code sequence below will ensure that all interrupts are reset and all interrupts vectors cleared. It is contained in the default *DSPKit* system initialisation routine, *processor_init()*. If your program does not start by calling *processor_init()*, it should execute the following sequence early on.

```
disable_interrupts();
clear_pending_interrupts();
disable_all_interrupts();
reset_isrs();
```

Installing and removing interrupts

Interrupt service routines are easily installed at run-time, as illustrated by the following code fragment from the *DSPKit* library *evmlib*, which installs an ISR for one of the serial ports.[12]

```
/*
 *  Install the data ISR and abort if error
 */
if ( ! install_isr( Receive0, c_int50 ) )
{
    fatal( "AIC interrupt installation error" );
}
```

When closing down a peripheral such as a serial port, the corresponding ISR should be removed. For example,

```
uninstall_isr( Receive0 );
```

Coding interrupt service routines

Interrupt service routines can be coded in C, simply by giving the ISR function a name of the format *c_intxx*, where *xx* is a number in the range 00 to 99. For example, the ISR used in the above examples starts as follows:

```
static void
c_int50( void )
{
    integer item;

    /*
     *  Read from serial port, convert to 14-bit integer.
     */
    item = (peripherals.ports[MYPORT].receive << 16) >> 18;
    ...
}
```

12 In fact, this is a simplified version of the code.

Memory Management

A typical TMS320C30 system has several regions of memory, of varying sizes and speeds. Of course, all have at least the on-chip memory, which is capable of up to three accesses per instruction cycle. Most will have some form of external memory—perhaps fast zero-wait-state static RAM, or maybe an external program ROM. More expensive systems may also have much larger memory built with dynamic RAMs, with one or two wait states. In multi-processor systems, there may also be shared memory accessible by more than one processor.

Effective use and management of these different memory areas—especially the precious internal memory—is crucial, and this chapter describes how memory can be allocated and used effectively. Detailed discussion of the ins and outs of *why* internal memory is so precious is left until Chapter 12.

This chapter is incomplete: contact the author for the current version.

Pipeline Conflicts

To achieve peak performance from the TMS320C30, it is necessary to understand how the device's instruction pipeline works and how instruction ordering and memory usage interact with the pipeline.

This chapter is not yet complete—contact the author for the current version.