SPIRIT-30

APPLICATION NOTES

Sonitech International Inc
14 Mica Lane, Suite 208
Wellesley, MA 02181 , USA

SPIRIT-30
APPLICATION NOTES

This packet contains the following SPIRIT-30 Application Notes:

_____ **Porting Application Program onto SPIRIT-30**

_____ **Developing Application Program using SPIRIT-30**

_____ **How to speed up applications using SPIRIT-30 and TMS320C30**

_____ **Serial I/O with AIB-2**

_____ **Parallel I/O interfacing with SPIRIT-30**

_____ **Other**

_____

_____

PORTING APPLICATION PROGRAM ONTO SPIRIT-30

This document describes my experiences of dealing with the SPIRIT-30 board while writing an application program. I have an application program, which creates fractal images. I have the program written in Microsoft C, and it works with a CGA or an EGA graphics adapter. The PC program is called IFC.EXE. I would like to port this program to SPIRIT-30 board. With this goal in mind, couple of ounces of some magic potion, and a positive attitude towards computers, I set to do the task. All the files referred to in this application note are available on the disk named 'APPLICATION NOTE'.

To begin with, modify your C:\AUTOEXEC.BAT file to contain the line

```
SET SPIRIT_DIR = C:\SPIRIT30
```

(SPIRIT30 is the base directory for the SPIRIT-30 software. If you choose not to have this name, give your own base directory name). Now, put the disk called APPLICATION NOTE in your disk drive A: and type A:INSTALL. This will install the necessary software in the directory C:\SPIRIT30\APPL, where C:\SPIRIT30 is the base directory as specified in the AUTOEXEC.BAT.

PRELIMINARIES

It is useful to have the following manuals at hand while writing the applications for the SPIRIT-30.

[1]. The SPIRIT-30 Run Time Library documentation
in the SPIRIT-30 Technical Reference Manual
[2]. The TMS320C30 Compiler User's Guide
[3]. The TMS320C30 Assembly Language Tools User's Guide

Go through these manuals at least once to be qualified as a C-30 application programmer.

Make sure that the installation of the TMS320C30 compiler and the other utilities is done right. In particular check to see if the PATH environment variable is updated to include the path of the directory containing the tools, and the C_DIR variable is appropriately set.

STEP 0:

Pray to your favorite spirit. STEPS 1 through 3 deal with writing the C code which will be compiled using the TMS320C30 compiler. STEPS 5 through 7 deal with creating the PC side of the software for this application.

STEP 1:

I have extracted the function ifc() from file ifc.c, and put it in the file difc.c. The prefix 'd' is added to indicate that this program would reside on the DSP.

STEP 2:

Later, I added the 'monitor' program to the difc.c. The result of this operation is in the file DIFC.C in the application note disk. This now is the entire source file for the DSP resident code. Take printout of this file. The main() does the following.

1. The DSP loops while checking the semaphor memory location '_wait' = 0.

2. If _wait = =IFC, a call to ifc() is made. When the ifc() is executed, the control comes back to main(). The semaphor _wait is asserted to 0. This assertion of _wait=0 by the DSP indicates to the host (PC) that the execution of ifc() is over. The DSP loops again while _wait = = 0. This is one way of writing the monitor program.

Any memory location on the board which needs I/O with the PC must be declared as a global variable. In the file DIFC.C, all such variables are declared as globals.

Read the file DIFC.C, especially the comments in the program, which are self-explanatory.

STEP 3:

Compile and link the program difc.c. Use the following commands:

```
c30c difc <enter>    /* COMPILE difc.c USING THE C-30*/
                          /* COMPILER */

lnk30 demo.cmd <enter>    /* LINK USING THE MEMORY */
                               /* MAP SPECIFIED IN demo.cmd*/
```

The file demo.cmd specifies that the executable DSP code is in the file difc.out. This step completes the creation of the DSP resident code. Read the file difc.cmd. The object modules linked along with difc.obj, and their contents are as follows

vectors.obj:	Source file - vectors.asm This file contains the code which resides at the location 0H in the memory. This code is executed at reset.
dspieee.obj:	Source file - dspieee.c Convert an array of numbers to/from TMS from/to IEEE floating point formats.
convert.obj:	Source file - convert.asm Convert formats of single numbers. Called by dsp30() and ieee30().
waitst.obj:	Source file - waitst.c Program the TMS320C30 for a given number of wait states. At reset, the chip runs with a default of 7 wait states.

STEP 4:

Create the PC side software. Let us divide the PC resident software into three modules. Module 1 is the highest level module, which contains main() on the PC side. This module handles user I/O, sets up the graphics and handles other DSP-independent functions. In our example, this module is called drawfern.c. Note that this module is really the program ifc.c, but it does not have the function ifc(), because we intend to have this function reside on the DSP. The module 1 contains the function calls initifc() and xifc(), which form Module 2 and Module 3 respectively.

STEP 5:

Create Module 2. This module is in the file initifc.c. The Module 2 contains the function initifc(), which does the following.

1. Download the code in the file difc.out to the SPIRIT board.

2. Find the addresses of the various variables in the program difc.c residing on the SPIRIT.

3. Start the DSP program.

STEP 6:

Create Module 3. This module is in the file xifc.c. The Module 3 contains the function xifc(), which is the driver for the DSP resident function ifc(). The function xifc() does the following.

1. Download the input arguments needed by the DSP resident function ifc() to the SPIRIT board.

2. Ask the DSP to execute ifc() by changing the memory location _wait =IFC.

3. Wait till ifc() execution is over by polling on _wait.

4. Upload the results of execution of ifc() from SPIRIT to PC.

STEP 7:

Compile and link the PC software. This is done by using the microsoft MAKE utility. The makefile is called FERNMAKE. This file compiles the modules 1,2, and 3 if necessary and links them together along with the run time library S30TOOLS.LIB. The file drawfern.exe containing the PC executable code is created.

DEVELOPING APPLICATION PROGRAM USING SPIRIT-30

This document describes the steps I followed in developing an application using SPIRIT-30 board. I would like to develop a fast floating point matrix multiplication program using SPIRIT-30. All the files referred to in this application note are included along with this note.

PRELIMINARIES

It is useful to have the following manuals at hand while writing the applications for the SPIRIT-30.

[1]. The SPIRIT-30 Run Time Library documentation in the SPIRIT-30 Technical Reference Manual.
[2]. The TMS320C30 Compiler User's Guide
[3]. The TMS320C30 Assembly Language Tools User's Guide

In the rest of this document these manuals will be referred to by their numbers.

I have to create two separate programs, one executing on PC, and one executing on SPIRIT-30. STEPS 1-2 deal with developing PC side software. STEPS 3-5 deal with developing software which resides on SPIRIT-30 (DSP program).

STEP 1:

I start writing the PC side software - "matrix.c". In order to interact with the SPIRIT-30, I need to use the SPIRIT-30 Run Time Library routines. So I include "s30tools.h", which contains function prototypes of the library routines. I have the basic algorithm: Download the DSP program, reset the TMS320C30, download the matrices, signal DSP program to start computation, wait for DSP program to complete multiplication, upload the resulting matrix and print the results. A simple two way handshaking protocol, isn't it? Given below are the details about how to perform these steps. Have a copy of the file "matrix.c" in hand while you go through the detailed description

(Step 1 of matrix.c)

First thing you need to do is to download the DSP program. What is so special about downloading DSP program compared to downloading data? The DSP program is nothing but an executable file, which can be executed by C30. This file is called 'COFF' file (refer [2]) and has several sections - text, data, etc. Each of these needs to be loaded at appropriate memory locations on SPIRIT-30. There is a library routine called "dsp_dl_exec()" (refer [1]) which will do this for you.

(Step 2 of matrix.c)

Now you have downloaded DSP program onto SPIRIT-30. But how to start executing it? Just reset the C30 using the "dsp_reset()" library routine. Resetting C30 will pass control to the program pointed by the vector in memory location 0H on SPIRIT-30. The DSP 'COFF' file contains a section called "vectors" with loading address 0, which contains the reset vector, and this is downloaded by dsp_dl_exec() routine.

(Steps 3,4 of matrix.c)

The DSP program has started executing, and is waiting for the PC side software to download the matrices which need to be multiplied. You can download data using the library routine "dsp_dl_long_array()" (refer [1]). But the question is, where (on SPIRIT-30 memory) to download the data? So the next step is to get the starting addresses of the matrices used by DSP program. This is possible only if the matrices are declared as global variables in DSP program (see STEP 3), in which case the information about these variables is given in the 'COFF' file. The dsp_dl_exec() routine not only loads the file onto SPIRIT-30 memory but also prepares a symbol table of global variables. We have a library routine called "get_laddr()" which, when given the global variable name, returns the starting address of this variable. So you need to know the variable names of the matrices used by DSP program. Use get_laddr() routine to get the starting address, and use this address to download the data.

(Step 5 of matrix.c)

Now signal the DSP program to start computation. This is similar to steps described above. Get the address of the variable 'flag', and set the flag by downloading a value (say) 1 in that location.

(Step 6 of matrix.c)

Wait for DSP program to complete the task. Keep polling the variable 'flag' by uploading and checking the value. The DSP program must reset the flag after completing the task (see STEP 3).

(Step 7 of matrix.c)

Get the starting address of the resulting matrix, and upload the values.

STEP 2:

In STEP 1 we created the PC side software source file matrix.c. Now we have to create the executable file. First choose the memory model, say X (L for large, M for medium, S for small). Compile using the command

```
cl /c /AX   matrix.c
```

where X in /AX is the first letter of the memory model. Then
link using the command

```
link   matrix,,,s30Xtool
```

where X is the first letter of memory model. s30Xtool is the SPIRIT-30 Run Time Library.

STEP 3:

My aim is to develop a DSP program which does fast matrix multiplication. So I organize DSP program into two modules, one written in 'C' which has the control software, and other written in C30 Assembly, which does the actual computation. Let us go through the steps in developing module 1, which I call mult.c. Have a copy of the file "mult.c" in hand before going through the details.

(Step 1 of mult.c)

In order to achive good performance, the first thing to do is to program C30 to 'zero wait state' for fast memory access. On reset, the C30 comes up with seven wait states, which is extremely slow. So we have to specifically program C30 to zero wait states. Use waitst() routine of waitst.obj.

(Step 2 of mult.c)

Before performing the computation, we need to get the values of the input matrices. In this case the values are downloaded by the PC side software. So wait for the PC to initialize the matrices and set the flag. Make sure to declare the flag and matrices as global variables, so that PC can get the starting addresses of these variables (see STEP 1) and work with these variables using download/upload library routines. If you declare them local, they would be allocated on stack, and PC software has no way of finding out the addresses. Also make sure to initialize the 'flag' variable to zero (reset).

(Step 3 of mult.c)

Now PC has given signal to start computation. Can we go ahead and start multiplicaton? NO! Note that we are working with floating point matrices. PC uses IEEE format to represent floating point numbers, whereas C30 uses it own format, which we call TI format. So before we do anything with the numbers, convert them to TI format. Use "dsp30()" routine from "convert.obj" to do the necessary conversion (refer [2]).

(Step 4 of mult.c)

Now you are all set to start the computation. This is the critical part of DSP program. Use an assembly routine compute() from module 2 to perform a fast multiplication. Make sure to pass proper parameters.

(Step 5 of mult.c)

We have the resulting matrix now. Can we signal the PC software that computation is done? NO! First convert the floating point numbers back to IEEE format using "ieee30()" routine from "dspiee.obj". Note that if you are working with integers you don't need to do any conversions, because both PC and C30 use the same representation for integers.

(Step 6 of mult.c)

Everything is done, so reset the 'flag' to signal PC software.

STEP 4:

The only thing remaining now is to develop module 2 of DSP program - the fast assembly routine "compute()", which performs matrix multiplication. Here are some clues for developing a 'C' callable assembly routine. First make sure to declare the function as global.

Note that the arguments to the function are pushed onto the stack in the reverse order. When control comes to your assembly routine, the stack pointer will be pointing to the return address. The one just below the return address would be the first parameter. Have a look at the file "compute.asm".

The code produced by 'C' compiler makes use of the registers AR3-AR7 and R4-R7. So if you need to use these as scratch registers, you must save them before you modify them, and restore them before you return. In "compute.asm", AR4 and AR5 are being used by the routine.

Refer to the note on "How to speed up Applications using SPIRIT-30" to get tips on developing efficient code.

STEP 5:

The source files for DSP program are ready now. Now we have to create the C30 executable 'COFF' file. Use the command

```
c30c    mult
```

to compile "mult.c". Link using the command

```
lnk30   mult.cmd
```

where mult.cmd is the command file for linking. Make sure to have the object file of module 2 (compute.obj) and object files convert.obj (dsp30() routine), dspieee.obj (ieee30() routine) and waitst.obj (to set zero wait state) in the command file. Also, check to see if ".bss" section is in ROM (SPIRIT-30 external memory) or not. You must have a line ".bss: {} > ROM" at the end of the command file mult.cmd. This is to make sure that the global variables reside in external memory, rather than internal memory.

```
/*************************************************************************/
/* matrix.c - program which demonstrates how to perform matrix           */
/*            multiplication with the SPIRIT-30 board.                   */
/*                                                                       */
/* Steps involved:                                                       */
/*      1. Download the C30 executable file "mult.out" which performs    */
/*         matrix multiplication. See mult.c, compute.asm files.         */
/*      2. Reset C30 to start execution of the file loaded in step 1.    */
/*      3. Get the addresses of global variables (addresses of a[][],b[][],*/
/*         c[][], and flag) used in "mult.out". These are the SPIRIT-30  */
/*         external memory addresses.                                    */
/*      4. Download the values of matrices a[][] and b[][] using the     */
/*         addresses obtained from step 3.                               */
/*      5. Set the flag (using the address obtained from step 3) to inform */
/*         DSP program (mult.out) that matrix multiplication can be started*/
/*      6. Wait for the flag to be reset by "mult.out", which indicates  */
/*         that the product has been computed - result is stored in c[][]. */
/*      7. Upload the resulting matrix c[][] using the address obtained in */
/*         step 3. Print the results.                                    */
/*                                                                       */
/* Files:                                                                */
/*      The PC source file is matrix.c                                   */
/*      The PC executable file matrix.exe                                */
/*      The C30 source files are 'mult.c' and 'compute.asm'              */
/*      The C30 executable file is 'mult.out'                            */
/*      See 'mult.c' file for instructions to create 'mult.out'          */
/*                                                                       */
/* To compile this program with Microsoft's C 5.1 compiler:              */
/*            cl /c matrix.c                                             */
/*            link matrix,,,s30stool                                     */
/*************************************************************************/

#include "s30tools.h"

float a[4][3]={                         /* matrix a[][] */
       { 1.0,  2.0,  3.0},
       {-1.0, -2.0, -3.0},
       { 5.0,  6.0,  7.0},
       {-5.0, -6.0, -7.0}
};
float b[3][4]={                         /* matrix b[][] */
       { 1.0, -1.0,  2.0, -2.0},
       { 1.0,  1.0,  1.0,  1.0},
       { 0.0,  0.0,  0.0,  0.0}
};
float c[4][4];                          /* matrix c[][] */


/* The following are set equal to the corresponding absolute address of the */
/* label in the DSP program by calling 'get_laddr()' with the label's name. */

long a_addr,b_addr,c_addr;
long flag_addr;
```

```
main()
{


/* STEP 1 */
        /* download DSP executable file 'mult.out' which performs matrix */
        /* multiplication                                                */

        if (dsp_dl_exec("mult.out") == -1) {
                printf("Error in downloading the file\n");
                exit(1);
        }

/* STEP 2 */
        dsp_reset();            /* RESET C30 to give control to mult.out */

/* STEP 3 */
        a_addr    = get_laddr("_a");    /* Find SPIRIT-30 address of a[][] */
        b_addr    = get_laddr("_b");    /* Find SPIRIT-30 address of b[][] */
        c_addr    = get_laddr("_c");    /* Find SPIRIT-30 address of c[][] */
        flag_addr = get_laddr("_flag"); /* Find SPIRIT-30 address of flag  */

        /* The above addresses are obtained from the symbol table prepared */
        /* by dsp_dl_exec() library routine while loading "mult.out"       */

        matrix(a,b,c);            /* Procedure to calculate product  */

        mat_print(a,b,c);         /* Print the results               */

        exit(0);
}

/*************************************************************************/
/* Procedure to compute to product of two matrices. c = a*b              */
/*************************************************************************/

matrix(a,b,c)
float a[4][3];
float b[3][4];
float c[4][4];
{
   int status;

/* Note: a 4x3 matrix consists of 12 floating pt. numbers */

/* STEP 4 */
        dsp_dl_long_array(a_addr,12,(long *)a);      /* Download a[][] matrix */
        dsp_dl_long_array(b_addr,12,(long *)b);      /* Download b[][] matrix */
```

```
/* A '1' is downloaded to SPIRIT-30 at address 'flag' which signals the C30   */
/*  that the data is ready and to perform a 4x4 product. When the C30 is      */
/*  done, it sets 'flag' = 0 to signal the PC that the data is ready.         */

        status = 1;
        printf("Waiting for DSP program to compute .. \n");

/* STEP 5 */
        dsp_dl_int_array(flag_addr,1,&status);          /* set the flag */

/* STEP 6 */
        while (1) {
                dsp_up_int_array(flag_addr,1,&status);  /* read the flag */
                if (status == 0) break;         /* check if it is reset */
                if (kbhit()) exit(0);
        }

/* STEP 7 */
        dsp_up_long_array(c_addr,16,(long *)c);    /* Upload c[][] matrix */
}

/*****************************************************************************/
/*  print the results                                                        */
/*****************************************************************************/

mat_print(a,b,c)
float a[4][3];
float b[3][4];
float c[4][4];
{
        register i,j;

        printf("\n\n");                 /* Print results */
        for (i=0; i<4; ++i)
        {
                for (j=0; j<3; ++j)
                        printf("%10.4f",a[i][j]);
                printf("\n");
        }
        printf("\ntimes\n");
        for (i=0; i<3; ++i)
        {
                for (j=0; j<4; ++j)
                        printf("%10.4f",b[i][j]);
                printf("\n");
        }
        printf("\nequals\n");
        for (i=0; i<4; ++i)
        {
                for (j=0; j<4; ++j)
                        printf("%10.4f",c[i][j]);
                printf("\n");
        }
}
```

```
/*************************************************************************/
/*  mult.c : C30 program which computes the product of two matrices      */
/*                                                                       */
/*  Steps involved:                                                      */
/*      1. Program C30 to 'zero wait state' for fast memory access (default */
/*         is seven wait states).                                        */
/*      2. Wait for the flag to be set by PC program (matrix.exe). PC pgm */
/*         sets this flag after initializing the matrices a[][] and b[][] */
/*      3. Convert the floating point values from IEEE format to TI format. */
/*         (PC uses IEEE format whereas C30 used TI format)              */
/*      4. Perform the matrix multiplication of a[][] & b[][], and place */
/*         the result in c[][]. (Use assembly routine for high performance) */
/*      5. The resulting matrix c[][] values are in TI format. Convert   */
/*         them into IEEE format.                                        */
/*      6. Reset the flag to inform PC program that results are ready    */
/*                                                                       */
/*  Files:                                                               */
/*      The PC source file is matrix.c                                   */
/*      The PC executable file is matrix.exe                             */
/*      The C30 source files are "mult.c" (this file) & "compute.asm"    */
/*      The C30 executable file is "mult.out".                           */
/*                                                                       */
/*  To create mult.out using TMS320C30 compiler -                        */
/*      c30c  mult       ('c' file)                                      */
/*      asm30 compute    (C30 assembly file)                             */
/*      lnk30 mult.cmd   (See mult.cmd file for details)                 */
/*************************************************************************/
float a[4][3];          /* matrix a[][] */
float b[3][4];          /* matrix b[][] */
float c[4][4];          /* matrix c[][] */
int flag = 0;           /* flag - initialize to zero (reset) */

main()
{

/* STEP 1 */
   waitst();            /* program C30 to zero wait state */

/* STEP 2 */
   while (flag != 1);   /* wait for the flag to be set by PC program - this */
                        /* flag is set by PC program after initializing     */
                        /* the matrices a[][] and b[][]                     */
/* STEP 3 */
   dsp30(a,12);         /* convert the floating point values of a[][] and   */
   dsp30(b,12);         /* b[][] from IEEE format to TI format              */

/* STEP 4 */
   compute(a,b,c,4,3,4);        /* call assembly routine which computes the */
                                /* product of a[][] & b[][], and puts the   */
                                /* results in c[][]                         */
/* STEP 5 */
   ieee30(c,16);        /* convert the floating point values of c[][]       */
                        /* from TI format to IEEE format                    */
/* STEP 6 */
   flag = 0;            /* reset the flag to inform PC program that the     */
                        /* results are ready                                */
}
```

```
;*****************************************************************************
; compute.asm:
;
; Synopsis:
;       compute(a,b,c,m,n,p)
;
;       float *a,*b,*c: pointers to two dimensional matrices a[][],b[][],c[][]
;       int    m, n, p: dimensions of the matrices ..
;                              a[][] is m X n, b[][] is n X p, c[][] m X p
; Description:
;       This function multiplies matrices a[][] and b[][], and puts the result
;       in matrix c[][]
;
; Stack structure after the call:
;               |-----------|
;      -fp(7)   |     p     |
;      -fp(6)   |     n     |
;      -fp(5)   |     m     |
;      -fp(4)   |     c     |
;      -fp(3)   |     b     |
;      -fp(2)   |     a     |
;      -fp(1)   |  ret addr |
;      -fp(0)-> |   old fp  |
;               |-----------|
; Registers modified:
;       r0, r1, r2, ar0, ar1, ar2, rs, re, rc
;
; Time taken (# cycles):
;       7m + 19mp + 2mnp   (2 for mpyf || addf instr latency)
;*****************************************************************************

fp      .set    ar3
        .global _compute
        .data

        .text
_compute
        push    fp              ; save old fp
        ldi     sp,fp           ; point to the top of the stack
        push    ar4             ; save ar4
        push    ar5             ; save ar5

        ldi     *-fp(2),ar0     ; ar0 = a (&a[0][0])
        ldi     *-fp(3),ar1     ; ar1 = b (&b[0][0])
        ldi     *-fp(4),ar2     ; ar2 = c (&c[0][0])
        ldi     *-fp(5),ar4     ; ar4 = m
        subi    1,ar4           ; ar4 = m-1
        ldi     *-fp(7),ar5     ; ar5 = p
        ldi     ar5,ir0         ; ir0 = p
        subi    1,ar5           ; ar5 = p-1
```

```
;
; for (i = 0; i <= m-1; i++)            [FOR LOOP 1]
;     for (j = 0; j <= p-1; j++)        [FOR LOOP 2]
;
; Start of the loop: ar0 = &a[i][0] and ar1 = &b[0][j]
loop
        ldi     ar1,r1          ; r1 = &b[0][j]
;
;        x = 0.0
;        for (k = 0; k <= n-1; k++)     [FOR LOOP 3]
;            x += a[i][k] * b[k][j];
;        c[i][j] = x;
;
        ldi     *-fp(6),rc      ; rc = n
        subi    1,rc            ; rc = n-1
        ldf     0.0,r0          ; r0 = 0.0
        ldf     0.0,r2          ; r2 = 0.0
        rpts    rc              ; repeat 'n' times
        mpyf    *ar0++,*ar1++(ir0),r0   ; | After this instruction :
    || addf     r0,r2           ;         |   ar0 = &a[i+1][0],
        addf    r0,r2           ;         |   ar1 = &b[n][j]
;
; end of [FOR LOOP 3]
;
        stf     r2,*ar2++       ; c[i][j] = x;
;
        dbd     ar5,loop        ; check index [FOR LOOP 2]
        subi    *-fp(6),ar0     ; ar0 = &a[i][0] (reposition to i th row)
        ldi     r1,ar1          ; ar1 = &b[0][j] (reposition to j th column)
        addi    1,ar1           ; ar1 = &b[0][j+1](reposition to j+1 th column)
;
;     end of [FOR LOOP 2]
;
;  i th row of c[][] has been computed .. so move on to next row
;
        addi    *-fp(6),ar0     ; ar0 = &a[i+1][0] (position to next row)
        dbd     ar4,loop        ; check index [FOR LOOP 1]
        ldi     *-fp(7),ar5     ; ar5 = p (index [FOR LOOP 2])
        subi    1,ar5           ; ar5 = p-1
        ldi     *-fp(3),ar1     ; ar1 = &b[0][0]
;
; end of [FOR LOOP 1]
;
        pop     ar5             ; restore ar5
        pop     ar4             ; restore ar4
        pop     fp              ; pop the old frame pointer
        retsu                   ; return to the calling program
;
;
        .end
```

```
/*************************************************************************/
/* MULT.CMD -  v1.10   COMMAND FILE FOR LINKING C30 C PROGRAMS           */
/*                                                                       */
/*    Usage:       lnk30 <obj files...>  -o <out file> -m <map file> c.cmd */
/*                                                                       */
/*    Description: This file is a sample command file that can be used   */
/*                 for linking programs built with the TMS320C30 C       */
/*                 Compiler.   Use it a guideline; you may want to change */
/*                 the allocation scheme according to the size of your   */
/*                 program and the memory layout of your target system.  */
/*                                                                       */
/*    Notes: (1)   You must specify the directory in which rts.lib is    */
/*                 located.  Either add a "-i<directory>" line to this   */
/*                 file, or use the system environment variable C_DIR to */
/*                 specify a search path for libraries.                  */
/*                                                                       */
/*           (2)   When using the small (default) memory model, be sure  */
/*                 that the ENTIRE .bss section fits within a single page. */
/*                 To satisfy this, .bss must be smaller than 64K words and */
/*                 must not cross any 64K boundaries.                    */
/*************************************************************************/
mult.obj
compute.obj
vectors.obj     /* TAKES CARE OF JUMP TO _c_int00 */
convert.obj     /* IEEE TO DSP CONVERSIONS      */
dspieee.obj     /* CONVERTS ARRAYS OF NUMBERS DSP_IEEE */
waitst.obj      /* PROGRAMS THE CPU FOR DESIRED WAITSTATES */
-c                                      /* LINK USING C CONVENTIONS      */
-o mult.out
-m mult.map
-lrts.lib                               /* GET RUN-TIME SUPPORT          */
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   VECS:   org = 0        len = 0xc0
   ROM:    org = 0x400    len = 0x7b00
   RAM0:   org = 0x809800 len = 0x400     /* RAM BLOCK 0                  */
   RAM1:   org = 0x809c00 len = 0x400    /* RAM BLOCK 1, PLUS 4K OF EXT   */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
/* A SIMPLE MINDED MAP WHICH PUTS EVERYTHING IN THE EXTERNAL MEMORY */
SECTIONS
{
   .text:   {} > ROM                /* CODE                               */
   .cinit:  {} > ROM                /* INITIALIZATION TABLES              */
   .stack:  {} > ROM                /* SYSTEM STACK                       */
   .data:   {} > ROM                /* DATA                               */
   .bss:    {} > ROM                /* GLOBAL & STATIC VARS (SEE NOTE 2)  */
   .sysmem: {} > ROM                /* DYNAMIC MEMORY - DELETE IF NOT USED */
}
```

HOW TO SPEED UP APPLICATIONS USING SPIRIT-30 AND TMS320C30

This document gives step by step procedure on how I speeded up an application executing on SPIRIT-30. I have a sample 'C' program. My goal is to reduce the execution time of this program on C30 as much as possible. All the text files ('.c','.asm' and '.cmd') referred to in this document are enclosed in this packet. This document expects the reader to have good knowledge about the TMS320C30 processor, Compiler, and Assembler.

PRELIMINARIES

It is useful to have the following manuals at hand while going through this document.

[1]. The TMS320C30 Compiler User's Guide
[2]. The TMS320C30 Assembly Language Tools User's Guide
[3]. The TMS320C30 User's Guide

In the rest of the document, these manuals are referred to by their numbers.

STEP 1:

First, I wrote down a sample program "dsp1.c" with some integer and floating point multiplications. Note that there is an outer 'for' loop which runs 10000 times. I have this so as to get the execution time in the order of seconds, so that the error percentage is less.

I compiled it using the command "c30c dsp1" and linked using the command "lnk30 dsp1.cmd", 'dsp1.cmd' being the command file. This program took **47.79** seconds.

From next step onwards, for step 'i' I plan to create 'dsp*i*.c' by modifying 'dsp*i-1*.c, and 'dsp*i*.cmd' by modifying 'dsp*i-1*.cmd' (if necessary).

STEP 2:

How to speed up this application? The first thing which stuck me is that, on reset the C30 processor by default assumes a 'seven wait state' external memory. The SPIRIT-30 board supports high speed on board memory, which can be used with zero wait states. So by using memory in zero wait states, I expect to get about 7 times speed up.

But how to program C30 to zero wait states? Simple! Use the 'waitst()' routine of 'waitst.obj' (included in Application Diskette). I made two changes: (1) Inserted a call to waitst() in the very beginning of the dsp1.c -> dsp2.c (2) Modified 'dsp1.cmd' to include 'waitst.obj' during linking -> 'dsp2.cmd'

This program took **7.08** seconds, almost 7 times improvement, as expected.

STEP 3:

Now I am using the fastest memory. So can there be further scope for improvement? I am using the fastest possible on board memory, but it is still slower than the on-chip memory, which is internal to the C30 processor. Since there are a lot of data accesses in this program, I could do better if I use internal memory to store data.

Note that the variables declared in this program are local, and hence the space is allocated on the stack. If you look at the 'dsp2.cmd' file, the stack is placed in external memory - observe the 'SECTIONS' part at the end of the file. The line ".stack: {} > ROM" (ROM means external memory) makes the linker place the stack in ROM. Modify this line to ".stack: {} > RAM0" (RAM0 means internal memory Bank-0) to have stack in internal memory -> dsp3.cmd. 'dsp3.c' is same as 'dsp2.c'.

This program took **5.22** seconds, a significant improvement!

STEP 4:

Is there anything faster than internal memory, for data accesses? CPU Registers, ofcourse! So next step is to use register variables. Unfortunately, TMSC30 complier allows only two integer and two floating point register variables. So we have to carefully choose which of the variables deserve to be placed in registers. To be specific, we must choose the most heavily used variables. 'dsp4.c' is same as 'dsp3.c' with some register variables. 'dsp4.cmd' is same as 'dsp3.cmd'.

This program took **5.00** seconds, not much improvement. Why? Because there are still quite a few memory variables, and also, the CPU registers are not drastically faster than internal memory.

STEP 5:

In all the above steps we have not looked at the actual code. Any 'C' compiler has its own limitations, and it cannot produce 100% optimized code. Also, C30 processor supports repeat and parallel instructions, which are not exploited by the 'C' compiler. So in this step we can try to use optimized assembly code in critical regions.

Step 5.1:

Have a look at 'dsp4.c'. There are three 'for loops', which are numbered. At this stage you must know how TMSC30 compiler generates code. After preprocessing, it converts the '.c' file into the equivalent assembly file ('.asm'), which in turn is assembled by the TMSC30 assembler, and finally linked by the TMS linker. Have a look at 'dsp4.asm' file, which is the file generated by the complier. Block 1 is the code for the 'for loop (1)'. Observe that it takes more than 1200 cycles to execute this part of the code. How to optimize this?

First of all, note that the 'for loop' resembles a simple 'do loop' with one index running from 0 to 99. So we can use a repeat (RPTS/RPTB) instruction. Also observe that the array indices increment by constant amount every iteration of the loop. Both the instructions can be transformed to two assembly 'STI' instructions, which can be executed using a 'parallel STI' instruction. Look at the optimized code for Block 1 in the file 'dsp5.asm'. It takes only about 100 cycles! I modified dsp4.asm by replacing the 'for loop 1' with equivalent optimized assembly code. 'dsp5.cmd' is same as 'dsp4.cmd'.

This program took **4.12** seconds, expected improvement.

Step 5.2:

I did a similar optimization for 'for loop 2', and the program executed in **3.08** seconds.

Step 5.3:

The last step is to optimize 'for loop 3', which is very tricky. I took advantage of the fact that the first two instructions of the loop are independent of the last two instructions. So I can split up the loop into two for loops. Also the first two instructions can be combined into a 'parallel MPYI || ADDI' instruction, and the last two can be combined into a 'parallel MPYF || ADDF' instruction. As a result, Block 3 can be optimized using two RTPS parallel instructions.

Looks as though everything is done, but there is a small problem! Once you convert 'for loop 3' also into assembly, there is absolutely no 'C' code within the outer 'for loop' (which runs 10000 times). The compiler doesn't care what you give within asm("") instruction. It assumes that it is proper assembly code, and dumps the given instructions in the proper place. It is the job of assembler to detect errors in this case. Have a closer look at 'dsp5.asm'. The 'C' compiler uses the register R0 as a scratch register to store index for the outer for loop. As far as the complier is concerned, there is no body for the loop, so it assumes that R0 remains unchanged. Unfortunately we are corrupting R0 in the optimized assembly code. I got over this problem by adding one more assembly instruction at the end of the loop, which restores the correct value of the index (see dsp5.c).

Now everything is set, and I executed the program. The outcome is unbelievable - **0.28** seconds! For a moment it looks suspicious, but if you take a closer look at the 'dsp5.asm' file, you will know the truth. Now, most of the program is in assembly, and bulk of the time is taken by the four repeat instructions, which come to about 400 cycles. Adding the remaining instructions, the body of the outer 'for loop' must take about 450 cycles, which at 60ns/cycle comes to 27 micro seconds. Considering the outer loop which runs 10000 times, the total time is 0.27 seconds, which is the close to the observed value.

CONCLUSION

So, from step 1 to step 5 we got a speed up of about 170 times! Here is the summary of the steps taken to speed up an application using SPIRIT-30: (Note that you cannot expect this much speed up for all the programs. It depends on how much of the code you can optimize using assembly, and how much of the data you can have in internal memory).

SUMMARY: HOW TO SPEED UP APPLICATIONS

Step 1: First write the program.

Step 2: Program C30 to zero wait states.

Step 3: Place data in internal memory.

Step 4: Have heavily used variables in registers.

Step 5: Program the critical sections in assembly.

To optimize assembly code -

* Make use of repeat (RPTS/RPTB) instructions.
* Try to use Parallel instructions.
* Use delayed branches.
* Avoid register conflicts for latency- you may have to rearrange some instructions (refer to 'section 10.2: Pipeline conflicts' of [3]).
* Avoid memory conflicts - distribute data in different memories (refer to 'section 10.3: Memory conflicts' of [3], refer to Appendix of this document).

APPENDIX

EXAMPLE: SPEEDING UP APPLICATION BY REMOVING MEMORY CONFLICTS

This appendix explains how I speeded up matrix multiplication program by removing memory conflicts. The matrix multiplication program is discussed in the document on "Developing Application Program using SPIRIT-30", and you must go through that document before reading this section.

Have a look at 'compute.asm' file. In the beginning of first page, an approximate timing equation is given in terms of m,n,p, the parameters to the function. For large m,n,p, the time is proportional to 'mnp'. This is due to the inner most loop, coded using RPTS instruction. Observe that there is a constant factor of '2' to this product. Have a closer look at the RPTS instruction - there are accesses to two different memory locations (pointed by AR0, AR1) in external memory, and hence there is memory conflict! An extra cycle is needed to fetch the data. If we can avoid this conflict, we can remove this factor, and speed up the computation 2 times.

How do avoid the memory conflict? A good solution is to place the data in internal memory (refer to section 10.3 of [3]). This means we must have the matrices in internal memory. In STEP 3 of this document we discussed how to place data in internal memory by declaring local variables and having stack in internal memory. It works fine for the example discussed (dsp1.c), since there is no interaction between PC and SPIRIT-30. But in matrix multiplication, data is initialized by the PC, and so the matrices must be declared as global variables (they must be in external memory).

How to get over this problem? The only solution seems to be to replicate data. So we retain the matrices as global variables, but declare exactly same set of variables (with different names, ofcourse) local, which can be placed in internal memory. Before calling the compute() function, copy the data from external memory to internal memory and call the function with local variables. When you return back, you have to copy the resulting matrix from internal to external memory so that PC can read back the results. Have a look at 'fmult.c', which is the modified version of mult.c. Also, 'fmult.cmd' has stack placed in RAM0. The new PC side software is 'fmatrix.c', which is exactly same as matrix.c, except that it downloads 'fmult.out' (modified DSP software) instead of 'mult.out'.

There are limitations to this method. There are only 2K words of internal memory, and if data exceeds 2K, you have to follow different strategy to distribute data, for e.g., using memory on peripheral bus (refer to section 10.3 of [3]).

```
/*****************************************************************/
/*                        File: dsp1.c                           */
/*****************************************************************/


main()
{
  int i;
  int index;
  int int1, int2, int3;
  int array1[100],array2[100];
  float f1, f2;
  float farray1[100],farray2[100];


 for (i=0;i<10000;i++) {


  for (index=0;index<100;index++) {
        array1[index] = 10;
        array2[index] = -12;
  }
  for (index=0;index<100;index++) {
        farray1[index] = 1.0;
        farray2[index] = 2.0;
  }

  f2 = 0.0;
  int2 = -1;
  int1 = 0;
  int3 = 0;

  for (index=0;index<100;index++) {
        int1 += int3;
        int3 = array1[index] * array2[index];
        f1 = farray1[index] * farray2[index];
        f2 += f1;
  }

 }

}
```

```
/*************************************************************************/
/* DSP1.CMD -  v1.10   COMMAND FILE FOR LINKING C30 C PROGRAMS           */
/*                                                                       */
/*    Usage:       lnk30 <obj files...>  -o <out file> -m <map file> c.cmd */
/*                                                                       */
/*    Description: This file is a sample command file that can be used   */
/*                 for linking programs built with the TMS320C30 C       */
/*                 Compiler.  Use it a guideline; you may want to change */
/*                 the allocation scheme according to the size of your   */
/*                 program and the memory layout of your target system.  */
/*                                                                       */
/*    Notes: (1)   You must specify the directory in which rts.lib is    */
/*                 located.  Either add a "-i<directory>" line to this   */
/*                 file, or use the system environment variable C_DIR to */
/*                 specify a search path for libraries.                  */
/*                                                                       */
/*           (2)   When using the small (default) memory model, be sure  */
/*                 that the ENTIRE .bss section fits within a single page. */
/*                 To satisfy this, .bss must be smaller than 64K words and */
/*                 must not cross any 64K boundaries.                    */
/*************************************************************************/
dsp1.obj
vectors.obj     /* TAKES CARE OF JUMP TO _c_int00 */
-c                                      /* LINK USING C CONVENTIONS      */
-o dsp1.out
-m dsp1.map
-lrts.lib                               /* GET RUN-TIME SUPPORT          */
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   VECS:   org = 0        len = 0xc0
   ROM:    org = 0x400    len = 0x7b00
   RAM0:   org = 0x809800 len = 0x400      /* RAM BLOCK 0                 */
   RAM1:   org = 0x809c00 len = 0x400     /* RAM BLOCK 1, PLUS 4K OF EXT   */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
/* A SIMPLE MINDED MAP WHICH PUTS EVERYTHING IN THE EXTERNAL MEMORY */
SECTIONS
{
   .text:   {} > ROM                /* CODE                               */
   .cinit:  {} > ROM                /* INITIALIZATION TABLES              */
   .stack:  {} > ROM                /* SYSTEM STACK                       */
   .data:   {} > ROM                /* DATA                               */
   .bss:    {} > ROM                /* GLOBAL & STATIC VARS (SEE NOTE 2)  */
   .sysmem: {} > ROM                /* DYNAMIC MEMORY - DELETE IF NOT USED */
}
```

```
/*****************************************************************/
/*                        File: dsp2.c                           */
/*****************************************************************/

main()
{
  int i;
  int index;
  int int1, int2, int3;
  int array1[100],array2[100];
  float f1, f2;
  float farray1[100],farray2[100];


  waitst(0);

 for (i=0;i<10000;i++) {


  for (index=0;index<100;index++) {
        array1[index] = 10;
        array2[index] = -12;
  }
  for (index=0;index<100;index++) {
        farray1[index] = 1.0;
        farray2[index] = 2.0;
  }

  f2 = 0.0;
  int2 = -1;
  int1 = 0;
  int3 = 0;

  for (index=0;index<100;index++) {
        int1 += int3;
        int3 = array1[index] * array2[index];
        f1 = farray1[index] * farray2[index];
        f2 += f1;
  }

 }

}
```

```
/*************************************************************************/
/* DSP2.CMD -  v1.10   COMMAND FILE FOR LINKING C30 C PROGRAMS           */
/*                                                                       */
/*    Usage:       lnk30 <obj files...>  -o <out file> -m <map file> c.cmd */
/*                                                                       */
/*    Description: This file is a sample command file that can be used   */
/*                 for linking programs built with the TMS320C30 C       */
/*                 Compiler.   Use it a guideline; you may want to change */
/*                 the allocation scheme according to the size of your   */
/*                 program and the memory layout of your target system.  */
/*                                                                       */
/*    Notes: (1)   You must specify the directory in which rts.lib is    */
/*                 located.  Either add a "-i<directory>" line to this   */
/*                 file, or use the system environment variable C_DIR to */
/*                 specify a search path for libraries.                  */
/*                                                                       */
/*           (2)   When using the small (default) memory model, be sure  */
/*                 that the ENTIRE .bss section fits within a single page. */
/*                 To satisfy this, .bss must be smaller than 64K words and */
/*                 must not cross any 64K boundaries.                    */
/*************************************************************************/
dsp2.obj
vectors.obj     /* TAKES CARE OF JUMP TO _c_int00 */
waitst.obj      /* PROGRAMS THE CPU FOR DESIRED WAITSTATES */
-c                                      /* LINK USING C CONVENTIONS      */
-o dsp2.out
-m dsp2.map
-lrts.lib                               /* GET RUN-TIME SUPPORT          */
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   VECS:   org = 0        len = 0xc0
   ROM:    org = 0x400    len = 0x7b00
   RAM0:   org = 0x809800 len = 0x400      /* RAM BLOCK 0                 */
   RAM1:   org = 0x809c00 len = 0x400     /* RAM BLOCK 1, PLUS 4K OF EXT  */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
/* A SIMPLE MINDED MAP WHICH PUTS EVERYTHING IN THE EXTERNAL MEMORY */
SECTIONS
{
   .text:   {} > ROM                /* CODE                               */
   .cinit:  {} > ROM                /* INITIALIZATION TABLES              */
   .stack:  {} > ROM                /* SYSTEM STACK                       */
   .data:   {} > ROM                /* DATA                               */
   .bss:    {} > ROM                /* GLOBAL & STATIC VARS (SEE NOTE 2)  */
   .sysmem: {} > ROM                /* DYNAMIC MEMORY - DELETE IF NOT USED */
}
```

```
/*************************************************************************/
/* DSP3.CMD -  v1.10   COMMAND FILE FOR LINKING C30 C PROGRAMS           */
/*                                                                       */
/*    Usage:       lnk30 <obj files...>  -o <out file> -m <map file> c.cmd */
/*                                                                       */
/*    Description: This file is a sample command file that can be used   */
/*                 for linking programs built with the TMS320C30 C       */
/*                 Compiler.  Use it a guideline; you may want to change */
/*                 the allocation scheme according to the size of your   */
/*                 program and the memory layout of your target system.  */
/*                                                                       */
/*    Notes: (1)   You must specify the directory in which rts.lib is    */
/*                 located.  Either add a "-i<directory>" line to this   */
/*                 file, or use the system environment variable C_DIR to */
/*                 specify a search path for libraries.                  */
/*                                                                       */
/*           (2)   When using the small (default) memory model, be sure  */
/*                 that the ENTIRE .bss section fits within a single page. */
/*                 To satisfy this, .bss must be smaller than 64K words and */
/*                 must not cross any 64K boundaries.                    */
/*************************************************************************/
dsp3.obj
vectors.obj     /* TAKES CARE OF JUMP TO _c_int00 */
waitst.obj      /* PROGRAMS THE CPU FOR DESIRED WAITSTATES */
-c                                      /* LINK USING C CONVENTIONS      */
-o dsp3.out
-m dsp3.map
-lrts.lib                               /* GET RUN-TIME SUPPORT          */
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   VECS:   org = 0        len = 0xc0
   ROM:    org = 0x400    len = 0x7b00
   RAM0:   org = 0x809800 len = 0x400      /* RAM BLOCK 0                 */
   RAM1:   org = 0x809c00 len = 0x400     /* RAM BLOCK 1, PLUS 4K OF EXT   */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
/* A SIMPLE MINDED MAP WHICH PUTS EVERYTHING IN THE EXTERNAL MEMORY */
SECTIONS
{
   .text:   {} > ROM                /* CODE                               */
   .cinit:  {} > ROM                /* INITIALIZATION TABLES              */
   .stack:  {} > RAM0               /* SYSTEM STACK                       */
   .data:   {} > ROM                /* DATA                               */
   .bss:    {} > ROM                /* GLOBAL & STATIC VARS (SEE NOTE 2)  */
   .sysmem: {} > ROM                /* DYNAMIC MEMORY - DELETE IF NOT USED */
}
```

```
/*****************************************************************/
/*                          File: dsp4.c                         */
/*****************************************************************/

main()
{
  int i;
  int index;
  int int2;
  register int int1, int3;
  int array1[100],array2[100];
  register float f1, f2;
  float farray1[100],farray2[100];

  waitst(0);

 for (i=0;i<10000;i++) {


  for (index=0;index<100;index++) {
        array1[index] = 10;
        array2[index] = -12;
  }
  for (index=0;index<100;index++) {
        farray1[index] = 1.0;
        farray2[index] = 2.0;
  }

  f2 = 0.0;
  int2 = -1;
  int1 = 0;
  int3 = 0;

  for (index=0;index<100;index++) {
        int1 += int3;
        int3 = array1[index] * array2[index];
        f1 = farray1[index] * farray2[index];
        f2 += f1;
  }

 }

}
```

(1)

(2)

(3)

```
*******************************************************
*    TMS320C30 C COMPILER     Version 1.10
*******************************************************
FP      .set    AR3

        .sect   ".cinit"
        .word   1,_flag+0
        .word   2
        .globl  _flag
        .bss    _flag,1
        .globl  _main
        .file   "dsp4.c"
        .text
*******************************************************
* FUNCTION DEF : _main
*******************************************************
_main:
        PUSH    FP
        LDI     SP,FP
        ADDI    403,SP
        PUSH    R4
        PUSH    R5
        PUSHF   R6
        PUSHF   R7
        LDI     0,R0
        PUSH    R0
        CALL    _waitst
        SUBI    1,SP
L2:
        LDI     @_flag,R0
        CMPI    1,R0
        BNZ     L2
        LDI     0,R1
        STI     R1,*+FP(1)
L5:
        LDI     *+FP(1),R0
        CMPI    10000,R0
        BGE     L4
        LDI     0,R1
        STI     R1,*+FP(2)
L8:
        LDI     *+FP(2),R0
        CMPI    100,R0
        BGE     L7
        ADDI    R0,FP,AR0
        LDI     10,R1
        STI     R1,*+AR0(4)
        ADDI    R0,FP,AR0
        LDI     -12,R2
        BD      L8
        STI     R2,*+AR0(104)
        ADDI    1,R0
        STI     R0,*+FP(2)
***     B       L8      ;BRANCH OCCURS
```

Blck 1

```
L7:
        LDI     0,R0
        STI     R0,*+FP(2)
L11:
        LDI     *+FP(2),R0
        CMPI    100,R0
        BGE     L10
        ADDI    R0,FP,AR0
        LDF     @CONST+0,R1
        STF     R1,*+AR0(204)
        ADDI    R0,FP,AR0
        LDF     @CONST+1,R2
        LDI     304,IR0
        BD      L11
        STF     R2,*+AR0(IR0)
        ADDI    1,R0
        STI     R0,*+FP(2)
***     B       L11     ;BRANCH OCCURS
L10:
        LDF     @CONST+2,R7
        LDI     -1,R1
        STI     R1,*+FP(3)
        LDI     0,R4
        LDI     0,R5
        LDI     0,R0
        STI     R0,*+FP(2)
L14:
        LDI     *+FP(2),R0
        CMPI    100,R0
        BGE     L13
        ADDI    R5,R4
        ADDI    R0,FP,AR0
        ADDI    R0,FP,AR1
        LDI     *+AR0(4),R0
        LDI     *+AR1(104),R1
        CALL    MPY_I
        LDI     R0,R5
        LDI     *+FP(2),R1
        ADDI    R1,FP,AR0
        ADDI    R1,FP,AR1
        LDF     *+AR0(204),R6
        LDI     304,IR0
        MPYF    *+AR1(IR0),R6
        BD      L14
        ADDF    R6,R7
        ADDI    1,R1
        STI     R1,*+FP(2)
***     B       L14     ;BRANCH OCCURS
L13:
        BD      L5
        LDI     *+FP(1),R1
        ADDI    1,R1
        STI     R1,*+FP(1)
***     B       L5      ;BRANCH OCCURS
```

Block 2

Block 3

```
L4:
        LDI     0,R1
        STI     R1,@_flag
L16:
        B       L16
*********************************************************
* DEFINE CONSTANTS                                      *
*********************************************************
        .bss    CONST,3
        .sect   ".cinit"
        .word   3,CONST
        .float  1.              ;0
        .float  2.              ;1
        .float  0.              ;2
*********************************************************
* UNDEFINED REFERENCES                                  *
*********************************************************
        .globl  _waitst
        .globl  MPY_I
        .end
```

```
/*************************************************************/
/*                        File: dsp5.c                       */
/*************************************************************/

main()
{
  int i;
  int index;
  int int2;
  register int int1, int3;      /* Register R4 for int1, R5 for int3 */
  int array1[100],array2[100];
  register float f1, f2;        /* Register R6 for f1, R7 for f2 */
  float farray1[100],farray2[100];

  waitst(0);
  while (flag != 1);

 for (i=0;i<10000;i++) {

/*
  for (index=0;index<100;index++) {
        array1[index] = 10;
        array2[index] = -12;
  }
*/

  asm(" LDI     10, R0");       /* constant 10 -> R0 */
  asm(" LDI     -12,R1");       /* constant -12 -> R1 */
  asm(" LDI     FP, AR0");      /* frame pointer -> AR0 */
  asm(" ADDI    4, AR0");       /* Address of array1 -> AR0 */
  asm(" LDI     FP, AR1");      /* frame pointer -> AR1 */
  asm(" ADDI    104,AR1");      /* Address of array2 -> AR1 */
  asm(" RPTS    99");           /* Do the following parallel instruction 100 times */
  asm(" STI     R0, *AR0++");           /* 10 -> array1[index++] */
  asm(" || STI      R1, *AR1++");       /* -12 -> array2[index++] */

/*
  for (index=0;index<100;index++) {
        farray1[index] = 1.0;
        farray2[index] = 2.0;
  }
*/

  asm(" LDF     1.0, R0");      /* constant 1.0 -> R0 */
  asm(" LDF     2.0,R1");       /* constant 2.0 -> R1 */
  asm(" LDI     FP, AR0");      /* frame pointer -> AR0 */
  asm(" ADDI    204, AR0");     /* Address of farray1 -> AR0 */
  asm(" LDI     FP, AR1");      /* frame pointer -> AR1 */
  asm(" ADDI    304,AR1");      /* Address of farray2 -> AR1 */
  asm(" RPTS    99");           /* Do the following parallel instruction 100 times */
  asm(" STF     R0, *AR0++");           /* 1.0 -> farray1[index++] */
  asm(" || STF      R1, *AR1++");       /* 2.0 -> farray2[index++] */
```

```
  f2 = 0.0;
  int2 = -1;
  int1 = 0;
  int3 = 0;

/*
  for (index=0;index<100;index++) {
        int1 += int3;
        int3 = array1[index] * array2[index];
        f1 = farray1[index] * farray2[index];
        f2 += f1;
  }
*/
  asm(" LDI     FP, AR0");      /* frame pointer -> AR0 */
  asm(" ADDI    4, AR0");       /* Address of array1 -> AR0 */
  asm(" LDI     FP, AR1");      /* frame pointer -> AR0 */
  asm(" ADDI    104,AR1");      /* Address of array2 -> AR1 */
  asm(" LDI     R4, R3");       /* int1 (=0) -> R3 */
  asm(" LDI     R5, R1");       /* int3 (=0) -> R1 */
  asm(" RPTS    99");           /* Do the following parallel instruction 100 times */
  asm(" MPYI    *AR0++, *AR1++, R1");   /* array1[index++] * array2[index++] -> int1 */
  asm(" || ADDI      R1, R3, R3");      /* int1 += int3 */
  asm(" LDI     R3, R4");       /* R3 -> int1 */
  asm(" LDI     R1, R5");       /* R1 -> int3 */
                                /* R4,R5 cannot be used in the above parallel */
                                /* instruction, so R1,R3 are being used */

  asm(" LDI     FP, AR0");      /* frame pointer -> AR0 */
  asm(" ADDI    204, AR0");     /* Address of farray1 -> AR0 */
  asm(" LDI     FP, AR1");      /* frame pointer -> AR1 */
  asm(" ADDI    304,AR1");      /* Address of farray2 -> AR1 */
  asm(" LDF     R6, R0");       /* f1 (?) -> R0 */
  asm(" LDF     R7, R2");       /* f2 (0.0) -> R1 */
  asm(" MPYF    *AR0++, *AR1++, R0");   /* farray1[index++] * farray2[index++] -> f1*/
  asm(" RPTS    99");           /* Do the following parallel instruction 100 times */
  asm(" MPYF    *AR0++, *AR1++, R0");   /* farray1[index++] * farray2[index++] -> f1 */
  asm(" || ADDF      R0, R2");          /* f2 += f1 */
  asm(" LDF     R0, R6");       /* R0 -> f1 */
  asm(" LDF     R2, R7");       /* R2 -> f2 */
                                /* R6,R7 cannot be used in the above parallel */
                                /* instruction, so R0,R2 are being used */

  asm(" LDI     *+FP(1), R0");  /* 'C' compiler uses R0 to store the index for */
                                /* the outer loop, and assumes that it is not  */
                                /* corrupted. But we are corrupting it - so    */
                                /* restore back its value from variable'index' */

 }

}
```

```
*******************************************************
*    TMS320C30 C COMPILER     Version 1.10
*******************************************************
FP      .set    AR3

        .sect   ".cinit"
        .word   1,_flag+0
        .word   2
        .globl  _flag
        .bss    _flag,1
        .globl  _main
        .file   "dsp5.c"
        .text
*******************************************************
* FUNCTION DEF : _main
*******************************************************
_main:
        PUSH    FP
        LDI     SP,FP
        ADDI    403,SP
        PUSH    R4
        PUSH    R5
        PUSHF   R6
        PUSHF   R7
        LDI     0,R0
        PUSH    R0
        CALL    _waitst
        SUBI    1,SP
 LDI 800H, ST
L2:
        LDI     @_flag,R0
        CMPI    1,R0
        BNZ     L2
        LDI     0,R1
        STI     R1,*+FP(1)
L5:
        LDI     *+FP(1),R0
        CMPI    10000,R0
        BGE     L4
```

```
        LDI     10, R0
        LDI     -12,R1
        LDI     FP, AR0
        ADDI    4, AR0
        LDI     FP, AR1
        ADDI    104,AR1
        RPTS    99
        STI     R0, *AR0++
        || STI      R1, *AR1++
        LDF     1.0, R0
        LDF     2.0,R1
        LDI     FP, AR0
        ADDI    204, AR0
        LDI     FP, AR1
        ADDI    304,AR1
        RPTS    99
        STF     R0, *AR0++
        || STF      R1, *AR1++
        LDF     @CONST+0,R7
        LDI     -1,R1
        STI     R1,*+FP(3)
        LDI     0,R4
        LDI     0,R5
        LDI     FP, AR0
        ADDI    4, AR0
        LDI     FP, AR1
        ADDI    104,AR1
        LDI     R4, R3
        LDI     R5, R1
        RPTS    99
        MPYI    *AR0++, *AR1++, R1
        || ADDI     R1, R3, R3
        LDI     R3, R4
        LDI     R1, R5
        LDI     FP, AR0
        ADDI    204, AR0
        LDI     FP, AR1
        ADDI    304,AR1
        LDF     R6, R0
        LDF     R7, R2
        MPYF    *AR0++, *AR1++, R0
        RPTS    99
        MPYF    *AR0++, *AR1++, R0
        || ADDF     R0, R2
        LDF     R0, R6
        LDF     R2, R7
        LDI     *+FP(1), R0
        BD      L5
        ADDI    1,R0
        NOP
        STI     R0,*+FP(1)
***     B       L5      ;BRANCH OCCURS
```

Block 1

Block 2

Block 3

```
L4:
        LDI     0,R1
        STI     R1,@_flag
L7:
        B       L7
*********************************************************
* DEFINE CONSTANTS                                      *
*********************************************************
        .bss    CONST,1
        .sect   ".cinit"
        .word   1,CONST
        .float  0.              ;0
*********************************************************
* UNDEFINED REFERENCES                                  *
*********************************************************
        .globl  _waitst
        .end
```

```
/*************************************************************************/
/*  fmult.c : C30 program which computes the product of two matrices     */
/*          - avoids memory conflicts by having data in internal memory -*/
/*                                                                       */
/*  Steps involved:                                                      */
/*      1. Program C30 to 'zero wait state' for fast memory access (default */
/*         is seven wait states).                                        */
/*      2. Wait for the flag to be set by PC program (matrix.exe). PC pgm */
/*         sets this flag after initializing the matrices a[][] and b[][] */
/*      3.(a) Copy the input data into internal memory (arr1[][],arr2[][]) */
/*      3.(b) Convert the floating point values from IEEE format to TI   */
/*            format (PC uses IEEE format whereas C30 used TI format)    */
/*      4. Perform the matrix multiplication of arr1[][] & arr2[][], and */
/*         place the result in arr3[][]. (Use assembly routine for high  */
/*         performance)                                                  */
/*      5.(a) The resulting matrix arr3[][] values are in TI format.     */
/*            Convert them into IEEE format.                             */
/*      5.(b) Copy the results (arr3[][]) back to external memory (c[][]) */
/*      6. Reset the flag to inform PC program that results are ready    */
/*                                                                       */
/*  Files:                                                               */
/*      The PC source file is fmatrix.c                                  */
/*      The PC executable file is fmatrix.exe                            */
/*      The C30 source files are "fmult.c" (this file) & "compute.asm"   */
/*      The C30 executable file is "fmult.out".                          */
/*                                                                       */
/*  To create fmult.out using TMS320C30 compiler -                       */
/*      c30c  fmult       ('c' file)                                     */
/*      asm30 compute    (C30 assembly file)                             */
/*      lnk30 fmult.cmd   (See fmult.cmd file for details)               */
/*************************************************************************/

float a[4][3];          /* matrix a[][] */
float b[3][4];          /* matrix b[][] */
float c[4][4];          /* matrix c[][] */

int flag = 0;           /* flag - initialize to zero (reset) */

main()
{
/* Declare exactly same matrices (a,b,c) with different names. These are */
/* local variables, and will be allocated on stack (internal memory).    */

   float arr1[4][3], arr2[3][4], arr3[4][4];

   int i,j;

/* STEP 1 */
   waitst();            /* program C30 to zero wait state */

/* STEP 2 */
   while (flag != 1);   /* wait for the flag to be set by PC program - this */
                        /* flag is set by PC program after initializing     */
                        /* the matrices a[][] and b[][]                     */
```

```
/* STEP 3.(a) */
/* Replicate data: copy data from external to internal memory */

   for (i=0;i<4;i++) for (j=0;j<3;j++) arr1[i][j] = a[i][j];
   for (i=0;i<3;i++) for (j=0;j<4;j++) arr2[i][j] = b[i][j];

/* STEP 3.(b) */
   dsp30(arr1,12);      /* convert the floating point values of arr1[][] and */
   dsp30(arr2,12);      /* arr2[][] from IEEE format to TI format            */

/* STEP 4 */
   compute(arr1,arr2,arr3,4,3,4);/* call assembly routine which computes the */
                            /* product of arr1[][] & arr2[][], and puts  */
                            /* the results in arr3[][]                   */


/* STEP 5.(a) */
   ieee30(arr3,16);     /* convert the floating point values of arr3[][]    */
                        /* from TI format to IEEE format                    */

/* STEP 5.(b) */
/* copy back the results into matrix c[][] - external memory */
   for (i=0;i<4;i++) for (j=0;j<4;j++) c[i][j] = arr3[i][j];

/* STEP 6 */
   flag = 0;            /* reset the flag to inform PC program that the     */
                        /* results are ready                                */

}
```

```
/*************************************************************************/
/* FMULT.CMD -  v1.10   COMMAND FILE FOR LINKING C30 C PROGRAMS          */
/*                                                                       */
/*    Usage:       lnk30 <obj files...>  -o <out file> -m <map file> c.cmd */
/*                                                                       */
/*    Description: This file is a sample command file that can be used  */
/*                 for linking programs built with the TMS320C30 C       */
/*                 Compiler.  Use it a guideline; you may want to change */
/*                 the allocation scheme according to the size of your   */
/*                 program and the memory layout of your target system.  */
/*                                                                       */
/*    Notes: (1)   You must specify the directory in which rts.lib is    */
/*                 located.  Either add a "-i<directory>" line to this   */
/*                 file, or use the system environment variable C_DIR to */
/*                 specify a search path for libraries.                  */
/*                                                                       */
/*           (2)   When using the small (default) memory model, be sure  */
/*                 that the ENTIRE .bss section fits within a single page. */
/*                 To satisfy this, .bss must be smaller than 64K words and */
/*                 must not cross any 64K boundaries.                    */
/*************************************************************************/
fmult.obj
compute.obj
vectors.obj     /* TAKES CARE OF JUMP TO _c_int00 */
convert.obj     /* IEEE TO DSP CONVERSIONS      */
dspieee.obj     /* CONVERTS ARRAYS OF NUMBERS DSP_IEEE */
waitst.obj      /* PROGRAMS THE CPU FOR DESIRED WAITSTATES */
-c                                      /* LINK USING C CONVENTIONS      */
-o fmult.out
-m fmult.map
-lrts.lib                               /* GET RUN-TIME SUPPORT          */
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   VECS:   org = 0        len = 0xc0
   ROM:    org = 0x400    len = 0x7b00
   RAM0:   org = 0x809800 len = 0x400      /* RAM BLOCK 0                  */
   RAM1:   org = 0x809c00 len = 0x400     /* RAM BLOCK 1, PLUS 4K OF EXT   */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
/* A SIMPLE MINDED MAP WHICH PUTS EVERYTHING IN THE EXTERNAL MEMORY */
SECTIONS
{
   .text:   {} > ROM               /* CODE                                */
   .cinit:  {} > ROM               /* INITIALIZATION TABLES               */
   .stack:  {} > RAM0              /* SYSTEM STACK                        */
   .data:   {} > ROM               /* DATA                                */
   .bss:    {} > ROM               /* GLOBAL & STATIC VARS (SEE NOTE 2)   */
   .sysmem: {} > ROM               /* DYNAMIC MEMORY - DELETE IF NOT USED */
}
```

SPIRIT-30
PARALLEL I/O INTERFACE

SPIRIT-30 has a 50 pin bi-directional 16 bit parallel I/O interface for high speed data acquisition, frame grabber, and memory interface. Using this interface, the data transfer can take place directly from an external device to the TMS320C30 processor memory. Typical transfer rates of upto 5 Million 16 bit words/sec can be achieved.

Signal Description:

The 50 pin connector layout and a description is given in the attached table (Table 2.2). The direction of the signals are from the viewpoint of the SPIRIT-30. The attached Figure 1.0 shows the P1 connector and IC's U1, U10, and U18 on the SPIRIT-30. U1 and U10 are 74ACT245 trancievers for the data signal. U18 is the buffer for the port addresses (A0, A1, A2, A3) and control signals. Other buffered signals available on the I/O bus are : (Note: 'X' indicates SPIRIT-30 parallel I/O bus or TMS320C30's Expansion bus)

*X1.RESET** (O) - When low, the device is put in RESET along with the SPIRIT-30 board.
X1.XF0 (O) - External flag pin which needs to be formatted as an output from the C30. This control signal can be used for synchronization of the external board with the SPIRIT-30, triggering of acquisition, interlocked operation for multiprocessing configuration, and other control tasks.
X1.INT1 & IACK* (O) - External interrupt
*X1.RDY** (I) - Ready signal from the external board to SPIRIT-30 to indicate that the device is ready
*X1.IOSTRB** - I/O strobe to access the external board (indicates valid address)
*X1.R/W** - Read/Write on the parallel I/O port. When this pin is high a Read is done by the SPIRIT-30.
X1.H1, BUF.CLOCK, X1.H3 - BUF.CLOCK is the buffered 8 Mhz clock, H1 and H3 are 16 Mhz clocks from the DSP.

Board Interface:

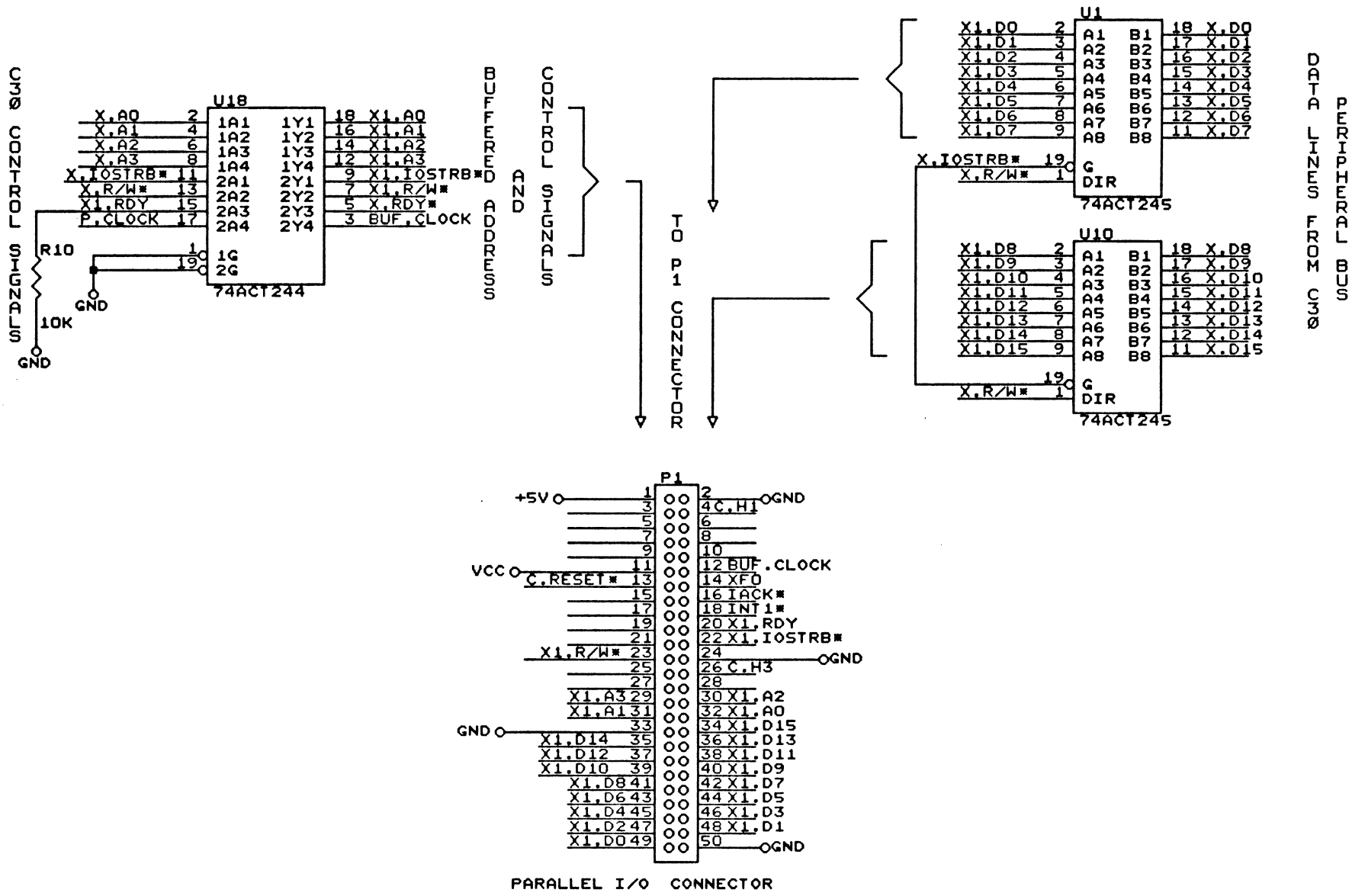
Using the buffered address, data, and control signals provided on the SPIRIT-30 parallel I/O, a simple hardware interface needs to be designed for an external board. The circuitry needed on the external board is shown in Figure 2.0. The two 74HCT245 buffers are not necessary unless you have several external boards. Address decoders (74HCT138's) are needed to decode the lower four addresses (A0..A4) of the SPIRIT-30 I/O port.

Physical Layout:

The SPIRIT-30 has a 50-pin male IDC connector with two rows of 25 pins. The cable length should be kept as short as possible to avoid problems due to transmission line effects (max length of 12 inches).

References:
TMS320C30 User's Guide pages 2-3,8-10..8-18, 13-17..13-20).
SPIRIT-30 Technical Reference Manual, pages 2-5..2-7.

From : Sonitech Int. Inc., 83 Fullerbrook Rd., Wellesly, MA.


C30 CONTROL SIGNALS
R10
10K
GND
U18
74ACT244
BUFFERED ADDRESS AND CONTROL SIGNALS
TO P1 CONNECTOR
U1
U10
74ACT245
DATA LINES FROM C30 PERIPHERAL BUS
P1
PARALLEL I/O CONNECTOR

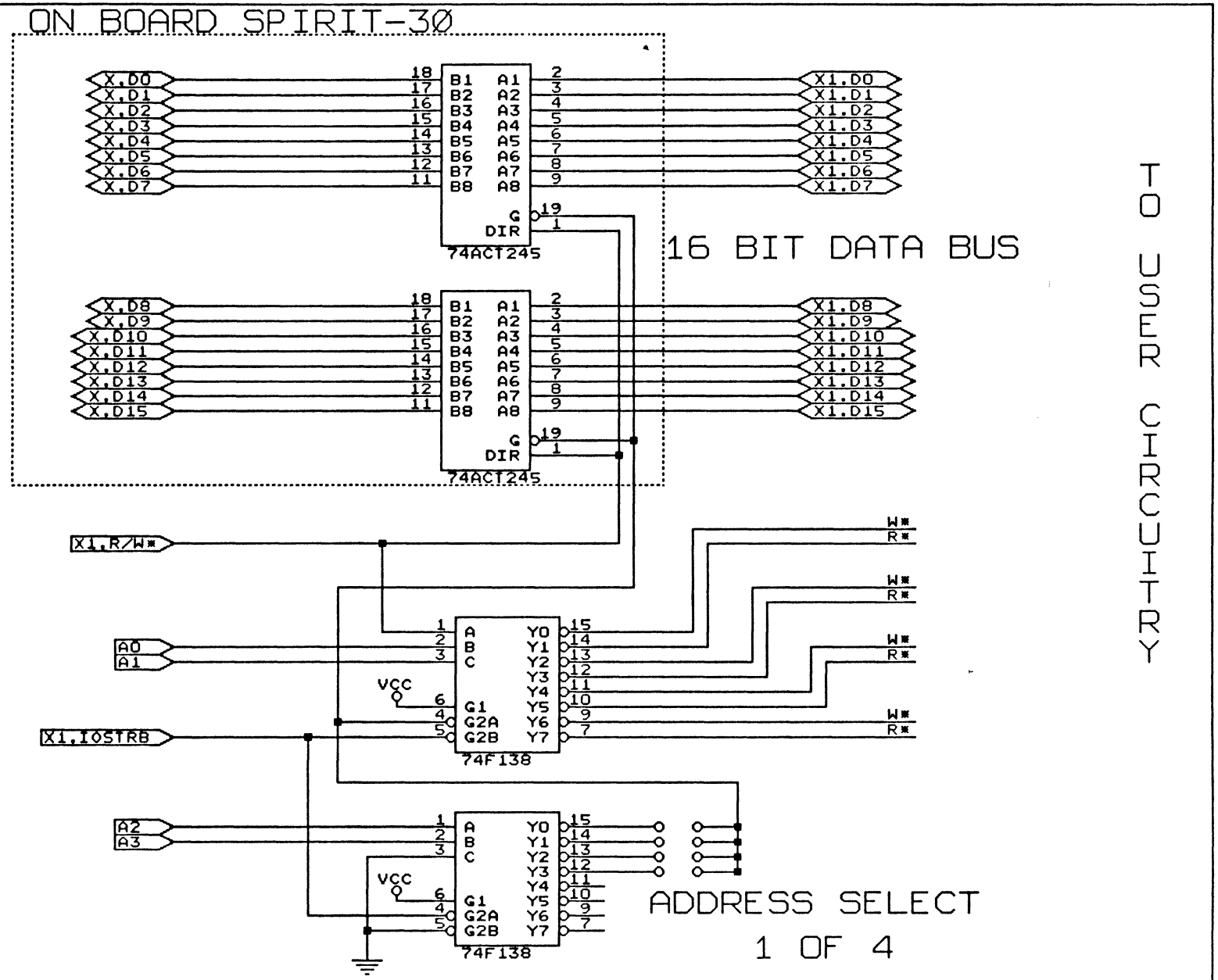


FIG 2.0 DECODER CIRCUITRY FOR EXTERNAL BOARD

8.2.2 Expansion Bus I/O Cycles

In contrast to primary bus and $\overline{\text{MSTRB}}$ cycles, $\overline{\text{IOSTRB}}$ reads and writes are both two cycles in duration (with no wait states) and exhibit the same timing. During these cycles, address always changes on the falling edge of H1, and $\overline{\text{IOSTRB}}$ is low from the rising edge of the first H1 cycle to the rising edge of the second H1 cycle. The $\overline{\text{IOSTRB}}$ signal always goes inactive (high) between cycles, and XR/$\overline{\text{W}}$ is high for reads and low for writes.

Figure 8-8 illustrates read and write cycles when $\overline{\text{IOSTRB}}$ is active and there are no wait states. For $\overline{\text{IOSTRB}}$ accesses, reads and writes require a minimum of two cycles. Some off-chip peripherals may change their status bits when read or written. Therefore, it is important that valid addresses be maintained when communicating with these peripherals. For reads and writes when $\overline{\text{IOSTRB}}$ is active, $\overline{\text{IOSTRB}}$ is completely framed by the address.

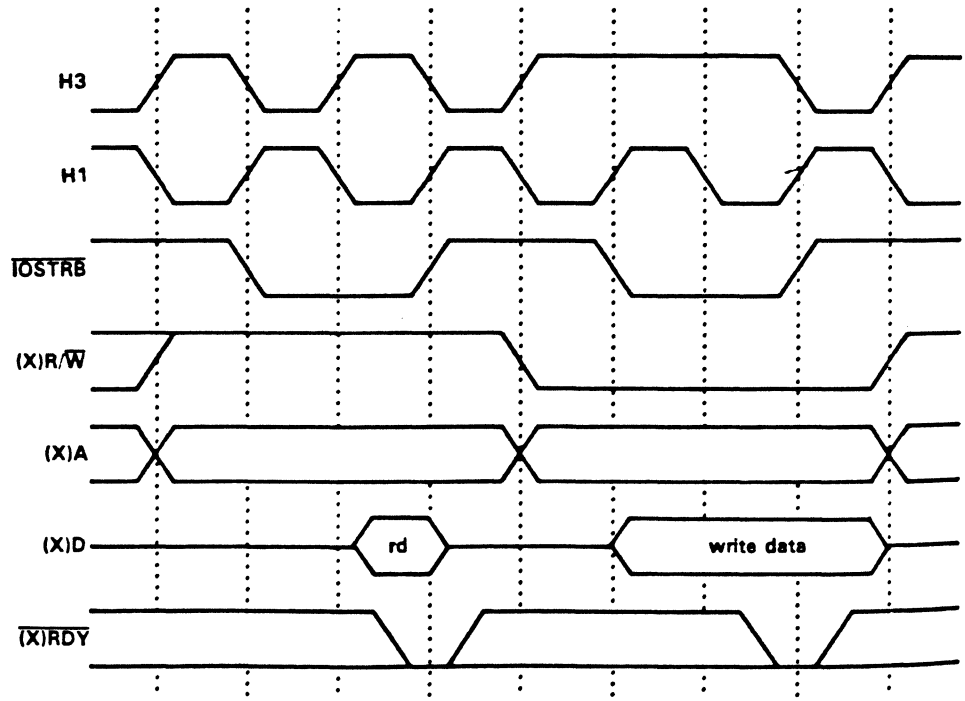


Figure 8-8. Read and Write for $\overline{\text{IOSTRB}}$ = 0

Figure 8-9 illustrates a read with one wait state when $\overline{\text{IOSTRB}}$ is active, and Figure 8-10 illustrates a write with one wait state when $\overline{\text{IOSTRB}}$ is active. For each wait state added, $\overline{\text{IOSTRB}}$, XR/$\overline{\text{W}}$, and XA are extended one clock cycle. Writes hold the data on the bus one additional cycle. The sampling of $\overline{\text{XRDY}}$ is repeated each cycle.

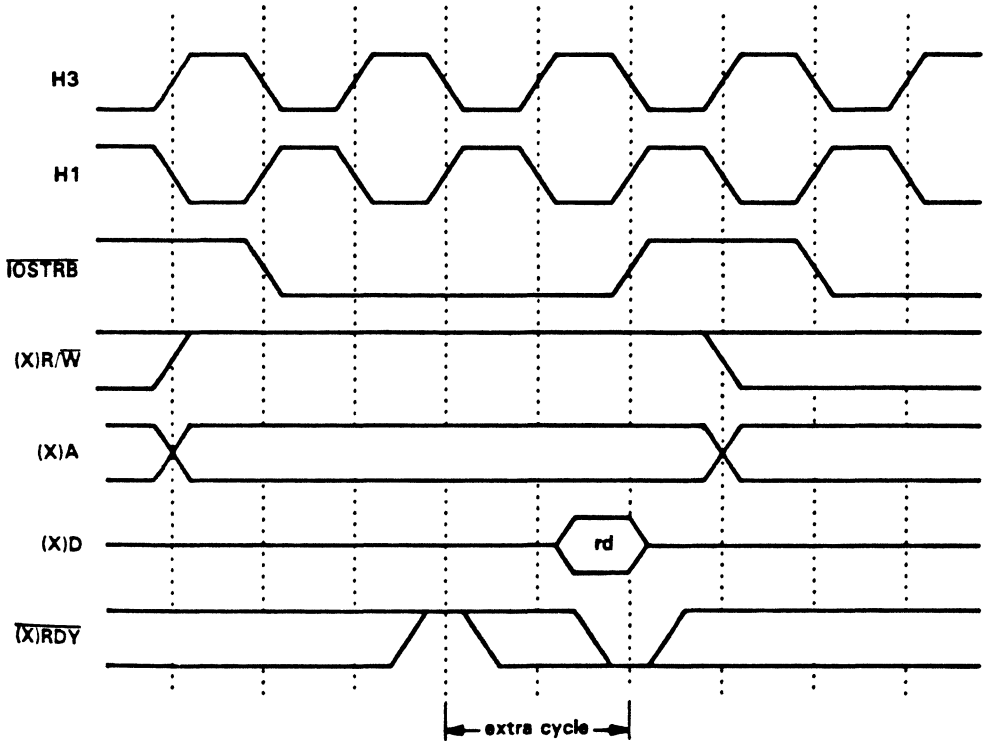


Figure 8-9. Read with One Wait-State for $\overline{\text{IOSTRB}}$ = 0

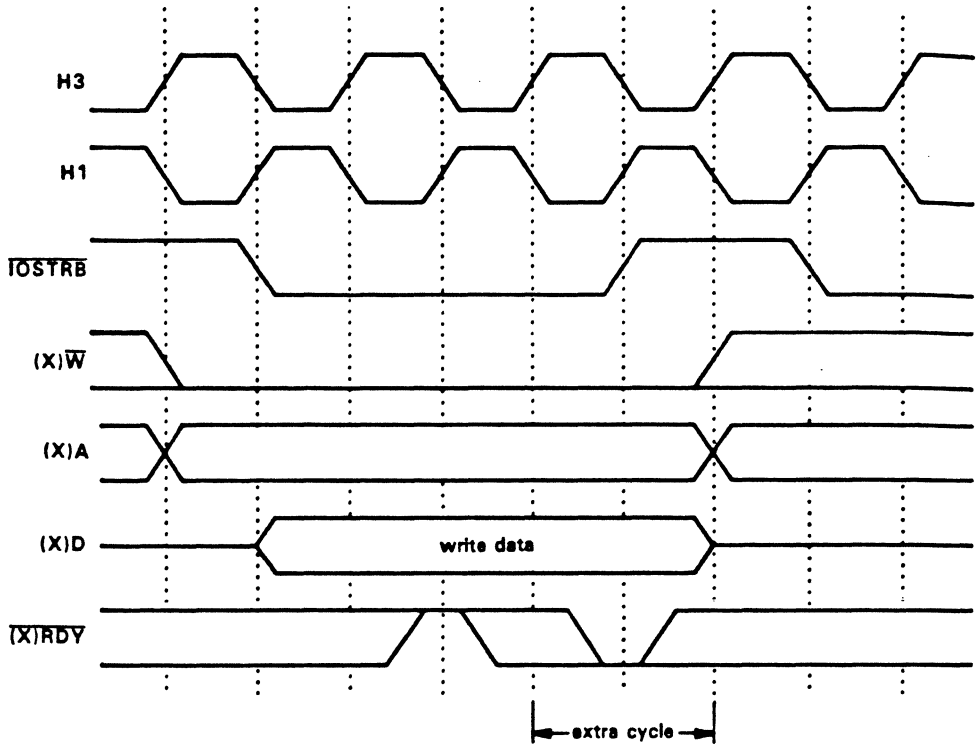


Figure 8-10. Write with One Wait-State for $\overline{\text{IOSTRB}}$ = 0

13.3 Expansion Bus Interface

The TMS320C30s expansion bus interface provides a second complete parallel bus which can be used to implement data transfers concurrently with, and independent of, operations on the primary bus. The expansion bus comprises two mutually exclusive interfaces controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals, respectively. These two signals are activated depending on what section of the memory space is accessed. This subsection discusses interface to the expansion bus using $\overline{\text{IOSTRB}}$; $\overline{\text{MSTRB}}$ cycles are identical in timing to primary bus cycles, and are discussed in Section 13.2.

Unlike thre primary bus, both read and write cycles on the I/O portion of the expansion bus are two H1 cycles in duration and exhibit the same timing. Thr XR/$\overline{\text{W}}$ signal is high for reads and low for writes. Since I/O accesses take two cycles, many peripherals that require wait states if interfaced either to the primary bus or using $\overline{\text{MSTRB}}$ may be used in a system without the need for wait states. Specifically, any devices with address access times greater than the 35 ns required by the primary bus but not less than 46 ns can be interfaced to the I/O bus without wait states.

A/D converters are one common DSP system component which often falls into this category. These devices are available in many speed ranges and with a variety of features, and while some may require one or more wait states on the I/O bus, others may be used at full speed.

One A/D converter that interfaces to the I/O bus without wait states and requires minimal additional logic is the ad 1332 from Analog Devices. Figure 13-12 illustrates an interface to this device.

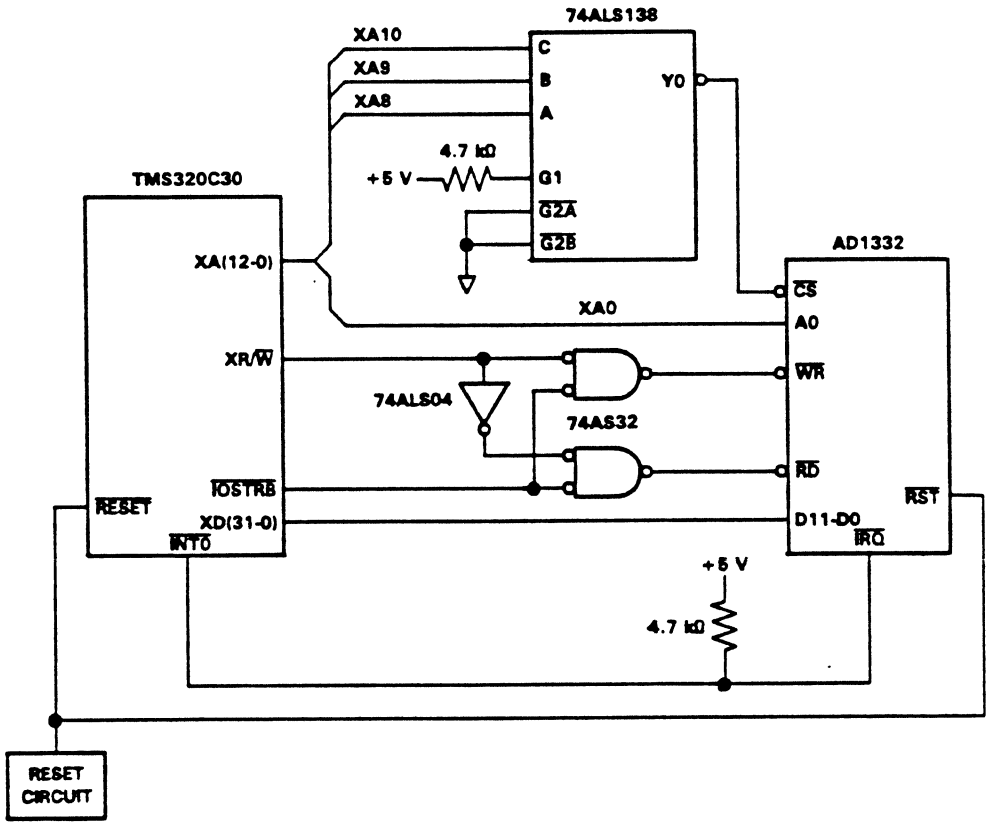


Figure 13-12. Expansion Bus Interface to A/D Converter

The interface uses a 74ALS138 to decode chip select for the converter. This configuration is shown assuming that other peripheral devices in the system also require chip select decodes. XA(8-10) are decoded to locate the converter at address 0804000h, which is the beginning of the I/O address space. Other peripherals may also use the outputs of the decoder, which generates chip selects in the I/O address space on 256 word boundaries.

XA0 is used to drive the single address line required in interfacing to the converter. This input selects between an internal 32-word FIFO buffer and the A/D's control/status register. Thus, the FIFO is located at address 0804000h and the control/status register is located at address 0804001h.

Since the converter requires $\overline{RD}$ and $\overline{WR}$ control signals rather than $\overline{WE}$ and $\overline{OE}$, random logic is used to generate these signals from $\overline{IOSTRB}$ and XR/$\overline{W}$. The converter's $\overline{IRQ}$ (Interrupt Request) output is used to alert the TMS320C30 to various conditions of converter status.

Figure 13-13 shows the timing for read and write operations between the TMS320C30 and the AD1332. Both operations are shown on the same timing diagram since, unlike the primary bus, only data bus timing and the state of XR/$\overline{W}$ differ between the two different types of cycles.

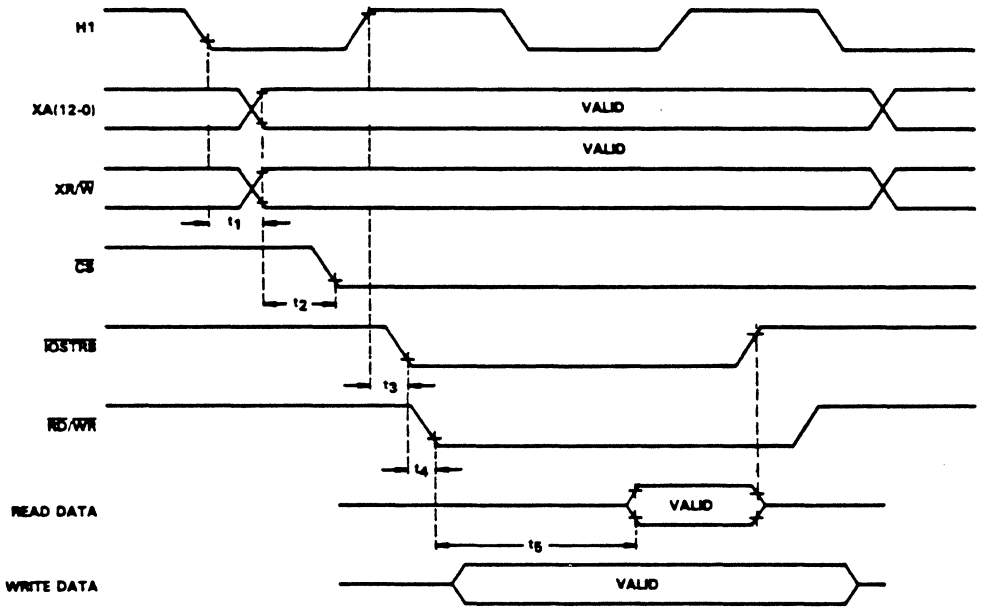


Figure 13-13. Timing of Expansion Bus Interface

In both cases, address and R/$\overline{W}$ are valid t1 = 10 ns after the falling edge of H1. After t2 = 17 ns, the propagation delay of the 74ALS138, the A/D converter's chip select goes low, selecting the device. Then, t3 = 10 ns after the rising edge of H1, $\overline{IOSTRB}$ goes low, and t4 = 5.8 ns following this, the $\overline{RD}$ or $\overline{WR}$ signal to the converter goes low, initiating either a read or write cycle, respectively.

For a read operation, the A/D converter provides data back to the TMS320C30 t4 + t5 = 30.8 ns after $\overline{RD}$ goes low. This satisfies the TMS320C30's requirement of having data valid 35 ns after $\overline{IOSTRB}$. For write operations, the A/D converter requires less than 5 ns of data setup and hold time with respect

to the rising edge of $\overline{WR}$. This is met with a high degree of margin by the TMS320C30.

It should be noted that for the AD1332's FIFO to be clocked properly, the $\overline{RD}$ signal must go high between accesses to the device. Therefore, although the AD1332 may be fast enough in some cases to be used at speeds approaching those of the primary bus, the $\overline{STRB}$ signal on the primary bus stays low for multiple consecutive read cycles. The I/O bus, therefore, is the preferable choice for interface to this device.