Ring Array Processor (RAP): Software User's Manual Version 1.0

P. Kohn† and J. Bilmes†

Abstract

The RAP machine is a high performance parallel processor developed at ICSI as described in previous technical reports. This report documents the RAP software environment. It is intended for the moderately experienced C programmer who wishes to program the RAP. The RAP software environment is very similar to the UNIX C programming environment. However, there are some differences arising from the hardware that the programmer must keep in mind. Also described is the RAP library which contains hand-optimized matrix, vector and inter-processor communications routines. SIMD programs can be developed under UNIX with a simulated RAP library and then recompiled to run on the RAP. Other parallel programming styles are also described.

†International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704-1105, USA

1. Introduction

This software user's manual is designed for the relatively experienced C programmer who wants to begin programming on the Ring Array Processor (RAP). Separate reports describe the RAP hardware [1] [2], the RAP software internals [3], RAP algorithms [4] and speech research supported by the RAP [5] [6]. Also relevant to the RAP user are the manuals from Texas Instruments for their C compiler [8], assembler and linker [9], simulator [10] and the TMS320C3x User's Guide [11].

The goals of the RAP software design were:

- Make it efficient for computational tasks of interest (backpropagation and speech recognition algorithms).
- Make it easy to learn and use by speech scientists who are often not familar with object-oriented languages.
- Make it as much like the standard UNIX environment as possible (allow program debugging under UNIX).
- Get the RAP system software operational as soon as possible.

2. The RAP: overview and current status

The RAP is a high performance parallel processor designed to train layered backpropagation networks [12]. Each processing element is a TMS320C30 digital signal processor that can sustain 32 million floating point operations per second. The prototype 8 processor (2 board) system runs at 256 MFLOPS and can pass data at up to 512 Mb/sec around its communication ring. A system with 64 processors (16 boards) is feasible in a single VMEbus card cage. The RAP is currently being used to train backpropagation networks for continuous speech recognition research.

Although the RAP hardware is capable of MIMD operation (Multiple Instruction streams controlling Multiple Data streams), the software and communications ring were designed for the SIMD (Single Instruction stream controlling Multiple Data streams) style of programming. In SIMD programming the same program is loaded into all of the processors. Usually, the processors will all be doing the same operations on different parts of the data. For example, to multiply a matrix by a vector, each processor would have its own subset of the matrix rows that must be multiplied. This is equivalent to partitioning the output vector elements among the processors. If the complete output vector is needed, the ring broadcast routine is called to redistribute the part of the output vector from each processor to all the other processors. This is described in more detail in the section on parallelism.

There is no shared memory between processing nodes. All inter-processor communication is handled by the ring. The hardware does not automatically keep the processors in lock step; for example, they may become out of sync by using the processor's node number in an "if" statement. However, when the processors must communicate with each other through the ring, synchronization automatically occurs. A node that attempts to read before data is ready or write when there is already data waiting will stop executing until the data can be moved.

3. Differences between UNIX and RAP C environments

This section documents the most important differences between the standard C environment and the RAP C environment. Many of these differences are related to the hardware. For instance, each processor has four banks of memory with different sizes and speeds. The processor architecture is not like most UNIX machines in that each memory address holds a 32 bit word instead of an 8 bit byte. Other differences relate to the lack of a memory management unit and virtual memory.

The basic procedure for compiling and linking your program is the same as in standard UNIX: the make utility does the work for you. There is a prototype makefile in the example subdirectory of the RAP software tree. See the documentation that comes with the software distribution tape for more details. Simply add the names of your object files (ending in ".o" instead of ".c" or ".asm") to the OBJ_FILES list.

Running the program is a little different. The RAPMC debugger is used to download, execute and monitor programs on the RAP processors. RAPMC is described in section 7.

3.1. Include rap.h

All C programs running on the RAP should start with the preprocessor directive:

```
#include "rap.h"
```

This one include replaces most of the usual C includes such as: math.h, stdio.h, stdlib.h and many others. There are actually two rap.h files: one for the RAP and one for the SUN. This allows the same source code to be compiled for either machine depending on the include directory specified in the makefile.

3.2. Stack is fixed size: no large local arrays

The RAP has no memory mapping or virtual addressing. The stack is a fixed region of memory that has a size defined in the link.cmd file (described in more detail in the Linker section 6.2.5). There is no stack overflow detection, making it all too easy to write garbage on nearby memory. The current default stack size is 4k words located in SRAM. To avoid overflow, do not declare non-static arrays inside a function with a size much over the stack size divided by the maximum number of nested function calls.

There are three possible ways to avoid the problems caused by a small fixed size stack:

1. The stack size can be increased as described in the Linker command file section. This is an option if there is sufficient free SRAM.
2. Declare a pointer to the array and use malloc to allocate the actual space for the data. Be careful to free pointers returned by malloc when they are no longer needed.
3. An array inside a function that is declared as static will be permanently allocated in SRAM. Again, this is an option if there is SRAM to spare.

Also, beware of functions with large recursion depths. In this case, the stack may have to be increased in size to handle the maximum number of nested calls.

When RAP programs start, all the stack memory is initialized to 0x12345678 (hex) by the boot function. This allows the exit system call to often detect stack overflows after the fact (assuming things are not so messed up that exit is never called). Also, the RAPMC debugger can be used to check for stack overflows.

3.3. Small memory model: no large global arrays

The direct addressing mode of the TMS320C30 can only access 64k words without changing the Data Base Pointer (DP) register. By default, the C compiler assumes that the DP register is set correctly for all accesses to global variables (or static variables inside a function). There is a compiler option (-b) that will cause the compiler to reload the DP register before every memory access, but this is extremely inefficient; the RAP library would have to be recompiled with this option as well.

Since the default linker command file puts all variables into SRAM and SRAM is only 64k words, users do not have to worry about this limitation unless the linker command file is modified. If the user attempts to declare large global arrays the linker will complain that it can not allocate enough SRAM. The easiest solution is to change the array to a pointer and use malloc.

Because of these limitations, care must be taken when modifying the linker command file that all variables are assigned to one memory region no larger than 64k. For example, one should never attempt to move selected variables into the on-chip memory since there is a huge address gap between SRAM and the internal RAM. Instead, use rap_malloc to allocate pointers to memory in internal RAM.

3.4. Memory spaces

Each processing node has four memory blocks:

Memory types and their characteristics				
Name	Description	Speed	Size	Primary Usage
RAM0	Inside processor chip	No wait states	1k words	library and user code
RAM1	Inside processor chip	No wait states	1k words	more user code
SRAM	Static memory on board	No wait states	64k words	code, data, stack
DRAM	Dynamic memory on board	3 wait states	1-4M words	large data arrays

Although internal and static RAM both have no wait states, the number of memory fetches or stores that can happen per processor cycle is different. The CPU can do up to three memory accesses to RAM0 and/or RAM1 in every processor cycle as summarized in table 10-2 of the *TMS320C3x User's Guide* [11]. At most one external memory read can happen per processor cycle. The CPU can only write one external word every two processor cycle.

The user can allocate data structures using the standard malloc function. In this case the allocation will occur in the memory block specified by the global variable DEFAULT_RAM. The initial value of this variable is SRAM. It can be changed by the user at any time.

The free function can be passed a pointer to any type of memory. It is an error to call free with anything other than a pointer returned by malloc that has not yet been passed to free. The free function attempts to report these errors.

To allocate memory in a specified memory block, use the function rap_malloc:

```
void *
rap_malloc(number_of_words_to_allocate, memory_block_name)
```

See the RAP libraries section for more details.

3.5. Word addressable instead of byte addressable

On the RAP each address or pointer can only be used to access a 32 bit word. Declaring a variable as double on the RAP is exactly the same as declaring it as float; the same is true for long and int. Characters are stored in the least significant 8 bits of the 32 bit word; the rest of the bits are normally zero. The sizeof operator on all of the basic data types (int, char, float and double) will return one. For portablility, always use the sizeof operator when passing system routines (such as read, write or malloc) the size of a memory region.

3.6. Opening files for binary or character I/O

Because of the 32 bit representation for characters, the standard input/output routines need to know if a file's data should be interpreted as 32 bit binary numbers or as 8 bit characters that must be expanded to 32 bit words. The fopen function has been extended to include the mode "b" for 32 bit direct binary reading. The "b" must be used in addition to "r" for reading or "w" for writing, e.g., "rb" or "wb". Use fopen the standard way to read or write files as streams of characters.

3.7. Floating point precision

There are floating point rounding bugs in the TI math routines. For example, on the RAP (int)pow(2.0, 9) is 511 instead of 512. When converting a floating point number to an integer, be sure to add 0.5 to it first in order to catch near misses caused by truncating a string of nines after the decimal point. Use the = = and != comparison operators very carefully on floating point numbers. Most often one uses (fabs(x-y) < EPSILON) instead of (x = = y) with

EPSILON typically smaller than 1e-6. There is no double precision arithmetic; all floating point numbers are 32 bits in memory or 40 bits when in a register.

3.8. Floating point

Floating point numbers on the TMS320C30 have a different internal representation than on most other machines; it is not IEEE compatible. To write floating point numbers to a file for use by programs on another machine, the simplest approach is to use alphanumeric (ASCII) format (fprintf).

If the same file is being read repeatedly, it is often useful to use the RAP to read the data in ASCII and write a "digested" file (using fwrite) that has the binary floating point numbers. This digested file can then be read more quickly by the RAP using fread. Do not try to create or read this digested file with any other machine without checking the floating point encoding.

There are routines for converting between IEEE and TMS320C30 floating point formats that can be found in the Software Applications section of the *TMS320C3x User's Guide* [11]. These routines are written in TMS320C30 assembly language and would be difficult to port because they rely on the floating point instructions.

3.9. Compiler bugs

Several TI C compiler bugs have been found during the RAP development. Since new compiler releases happen on a regular basis, these will not be documented here. Call the TI DSP Hotline for more information at 1-713-274-2320.

3.10. DSP does not stop for bad instructions or invalid pointers

Digital signal processors tend to lack some of the nice features of regular microprocessors. Most microprocessors have a special error trap for bad instructions, memory pointers out of range, division by zero or writing data over the program's own instructions. These features make it likely that programs that corrupt code or data will stop eventually. However, on the RAP, a program can write to any invalid address and the data may end up somewhere else (the hardware ignores the unused upper address bits and behaves as if they are zeros). Moreover, invalid instructions do not cause a special interrupt. Especially when the stack overflows, a jump to a random address can cause the bug to seem to come from somewhere unrelated in the program. In general, once the processor is off the track it will settle into an infinite loop or hit something that causes it to call exit. Since the RAP board can not become a VMEbus master, the damage to the rest of the system and network files is somewhat limited.

On the TMS320C30 the result of division by zero is zero. It is a good habit to test the denominator for zero before dividing.

3.11. Global variables

Two special global variables are provided to support parallel programming. The N_NODE variable is set to the number of processors in the RAP system. The NODE_ID variable is set to a number from 0 to N_NODE-1, indicating which processor the program is on. The ring is unidirectional, allowing data to flow only in NODE_ID order: from node zero to node one, ..., from node N_NODE-1 to node zero.

4. How to use the RAP for parallel processing

There is no general problem-independent technique for utilizing parallel computation on the RAP. However, there are several basic schemes that work for many types of problems.

4.1. Processor farm

If a problem can be broken into totally independent subproblems, then each subproblem can run on its own processor. This is a near trivial form of parallel processing since it involves little or no modification to the uniprocessor program. Results from each processor may need to be combined to produce the final output. For example, to run a series of experiments with different parameter values, one processor is assigned to each experiment. Another example is dynamic programming for speech recognition experiments; in this case the database of sentences is divided among the processors.

The RAPMC debugger supports this approach by allowing a RAPMC process to be started for each processor on the RAP. Window system users can open a separate window for each processor.

RAPMC can also be used to setup a separate batch queue of programs to run on each processor. A RAPMC process for each processor is run in the background with a different script file. The script file can change the processor number with the "node" command and then repeat the "load", "run" and "wait" commands for each program to be run.

4.2. SIMD

Data structures for the most time consuming parts of the problem are divided among the processors. For example, each processor calculates a part of a vector. The communications ring may be used to redistribute the results from each processor (see ring_broadcast).

The code below demonstrates this by multiplying a matrix and a vector. The rows of the matrix are divided into N_NODE sections, each with n_row_per_node rows. If the number of rows in the matrix does not divide the number of processing nodes evenly, up to N_NODE-1 rows of zeros can be added to fill out the matrix with little efficiency loss.

```
/*
**   Distributed matrix multiply
*/
rap_mul_mv_v(
   int n_row_per_node,      /* number of matrix rows per processor */
   int n_col,               /* number of columns in the matrix */
   float *matrix_per_node,  /* rows of matrix for this processor */
   float *in_vector         /* input vector */
   float *out_vector        /* output vector */
)
{
float *out_vector_per_node;

   /* make pointer to start of this processor's part of output vector */
   out_vector_per_node = out_vector + (NODE_ID * n_row_per_node);

   /* multiply this processor's rows by the in_vector */
   mul_mv_v(n_row_per_node, n_col,
            matrix_per_node, in_vector, out_vector_per_node);

   /* redistribute the out_vector_per_node from each processor
   ** to produce the complete output vector
   */
   ring_distribute(n_row_per_node, out_vector_per_node, out_vector);
}
```

4.3. Pipeline

In this scheme, each processor operates on the stream of data flowing through the ring. For example, processor 0 reads from an analog to digital converter and does a FFT; processor 1 sends the spectrum to processor 2 using the ring_put and ring_get routines described in section 6.2.4.3.6. Processor 1 preprocesses the spectrum for processor 2. Processor 2 runs the forward pass of a "neural" net that was trained to categorize phonemes. Processor 3 takes the phoneme probabilities and runs a dynamic programming step to recognize words. In this signal processing example the ring was used as a linear pipeline; processor 3 never sends to processor 0.

4.4. MIMD

This is the most general and also the most difficult form of parallelism as there are no restrictions on what each processor is doing. The global variables NODE_ID and N_NODE can be used to determine on which processor the program is running and how many processors are available. The RAP library provides routines for inter-processor communication on the ring (ring_read, ring_write, ring_put, ring_get, ring_sync, ring_broadcast). The RAPMC debugger can be used to load different programs on each processor and redirect each ones standard output stream to a different file. As an alternative, multiple RAPMC processes can be started, one for each processor.

Each processor also has two high speed serial ports that can interconnected to create more complex communication topologies. Currently, there is no support in the RAP library for the serial ports. They are described in detail in the TMS320C3x User's Guide [11].

5. Quick start: Running a simple program on the RAP

This section contains step by step instructions to get a simple SIMD style example program to run on the RAP. The setup used for this example should be easy to be extended to larger programs with multiple C and assembler source files. Simply add your code to the main.c file, or make new source files and add their names to the file list inside the makefile.

5.1. Set up PATH

Change your shell startup script file ("~/.cshrc" if you are using csh) so that the path of the bin subdirectory of the RAP software tree is included in the "setenv PATH" command. Then source the startup script to set your current shell's search path.

5.2. Set up .rapmcrc

The .rapmcrc file in your current working directory is automatically run when rapmc starts up. To set up rapmc to use all of its nodes as one SIMD machine, the following 3 lines should be in your .rapmcrc file:

```
node *
* miss
0 catch
```

The first command tells RAPMC to use all the processors of the RAP as one SIMD machine; each RAPMC command will go to all nodes of the RAP (unless a node number prefix is used). The second two commands cause standard output from all but processor 0 to be ignored. In SIMD mode all processors usually output the same thing at the same time.

5.3. Copy files from example directory

Copy all the files from the example subdirectory of the RAP root directory into your own RAP working directory. These files are:

```
main.c - a simple program that prints some stuff
makefile - makefile for yo
link.cmd - linker command file for building yo
```

A listing of these files is given in appendix A.

5.4. Make and Run

Do a "make" command. Type "rapmc" to enter the RAP master commander. RAPMC allows programs to be loaded, run and debugged. Give the following commands to RAPMC:

```
load yo
run These are some arguments
quit
```

Always exit RAPMC (using the "quit" command) when you are not using the RAP since other users may be queued up to use it. So long as any of your RAPMC processes exist, other users will be held off.

6. RAP programming environment

This section describes the C level programming environment on the RAP in more detail.

6.1. The main function

As in standard C, the first user function called on startup is main(). The command line arguments are the same as in standard C. The first argument to main (argc) is the number of command line arguments plus one. The second argument (argv) is a pointer to an array of character pointers to each argument; the name of the program itself as the first (zeroth index) argument.

6.2. RAP libraries This section describes the functions available in the RAP library. Many of these are the same as the standard C library. A few of these have been extended, such as malloc and fopen. Some functions are RAP specific, such as inter-processor communication and matrix/vector operations.

6.2.1. Standard C functions

Many standard C functions are supported by the RAP environment. Appendix B is a listing of the standard C functions currently in the library.

6.2.2. Differences in memory allocation

The RAP malloc function needs to know what type of memory to allocate. The memory types recognized by rap_malloc include: RAM0, RAM1, SRAM, DRAM and FASTEST. RAM0 and RAM1 are each 1k word blocks of very fast on-chip memory. SRAM is a 64k word block of fast off-chip memory. DRAM is a 1M word or 4M word block of slow off-chip memory (depending on the DRAM chips installed). FASTEST is a special keyword that tells malloc to try allocating the memory in the following priority: RAM1, RAM0, SRAM and DRAM. When FASTEST is used, early calls to rap_malloc will get faster RAM than later calls when only slower RAM remains.

The user can also allocate data structures using the standard malloc function. In this case the allocation will occur in the memory block specified by the global variable DEFAULT_RAM. The initial value of this variable is SRAM. It can be changed by the user at any time.

To allocate a specific type of memory, the rap_malloc function has a second argument containing the memory type.

6.2.2.1. rap_malloc

```
void *rap_malloc(int size, int memory_type)
```

Allocate size words from memory block memory_type and return a pointer to it. Returns NULL if this can not be accomplished. It is very important to ALWAYS check for a NULL return, since memory corruption can be hard to trace later. The ckmalloc routine described below does this checking for you.

Note that datatype void* can be cast into a pointer to any other datatype.

6.2.2.2. malloc_usage

```
void malloc_usage()
```

Print a summary of memory used and available in each memory block.

6.2.2.3. ckmalloc

```
void *ckmalloc(int size, int memory_type, char *where)
```

This does the same thing as malloc except that it prints a message and stops if no memory can be allocated. The last argument is a string that is inserted in the error message to identify where in the program it stopped.

6.2.2.4. MALLOC macro

To make allocating a little cleaner, the MALLOC macro is provided. This macro eliminates the need to explicitly cast the result of malloc to the correct data type and also the need to multiply the size of an array by the sizeof an element in the array.

```
float_pointer = MALLOC(float, n_element, SRAM, "weight matrix");
```

The above is translated into:

```
float_pointer =
   (float *) ckmalloc(n_element * sizeof(float), SRAM,
                      "weight matrix");
```

6.2.3. Differences in stdio

There are some important differences between the standard stdio functions and the ones provided by the RAP library.

6.2.3.1. BINARY vs. ASCII files

Because the TI320C30 stores each char in a 32 bit word, the host software that services file requests must know if each 32 bit word in the file corresponds to 1 or to 4 words on the RAP. When doing character input or output (e.g. fprintf or fscanf), use fopen as in the standard (e.g. "r" to read, "w" to write, "a" to append). If the data is a direct copy of memory that contains 32 bit quantities then the fopen must have the added mode letter "b" (e.g. "rb", "wb", "ab"). Also keep in mind that the TI320C30 floating point binary representation is not directly compatible with workstations.

6.2.3.2. SIMD mode file reading

When a RAP program reads a file from all its processors, a host request is generated for each processor. This inefficiency causes file reading overhead not to scale very well to large numbers of nodes. A solution is to only read the file from one node and use the communications ring to broadcast the file buffer to the other nodes. Since reading files is fairly common, this solution was implemented inside the standard library.

In order to read a file in SIMD mode, use "rs" as the second argument to the fopen call. This causes node 0 to read the file; all other nodes get the data from the ring. When using this mode, all nodes must be reading the same file with the same stdio library calls (fread, fseek, fclose, fgetw, fscanf, etc.) executed in the same order.

6.2.3.3. Changing size and memory type of file buffers

The file buffers used by the stdio package are located in the memory block specified by the global variable STDIO_RAM. The initial value of this variable is SRAM. It can be changed by the user at any time.

The default block size is currently 1024 words. This is specified by the global variable called BUFSIZ. A larger value will increase I/O throughput at the cost of using more memory. This variable can also be changed by the user at any time.

6.2.3.4. Multi-processor file writing

Writing files from the RAP can be tricky. The simplest approach is for each processor to write to a different file name. Things get more complex when all the processors need to write data into the same file. In general, bad things will happen if multiple processors even open the same file for writing. One simple solution is to use the ring to send all the data to processor 0. In this scheme, only processor 0 opens and writes the file.

6.2.4. RAP specific routines

This section describes the hand optimized RAP routines that were written to support the backpropagation algorithm. Many of these routines should also be useful for other applications.

6.2.4.1. Matrix and vector functions

While this is far from a complete set of matrix and vector routines, it is sufficient to run backpropagation algorithms efficiently. More routines are being added, so this list is unlikely to be complete.

These routines have a naming and argument order convention. The general form for names is "operation_xx_x". The first x's in the name describe the inputs and the last x describe the output type of the operation. These x's can be one of: "s" for scalar, "v" for vector or "m" for matrix. There is also "b" for a binary vector (elements are 0 or 1) that is encoded by a list of indexes to the elements that are 1.

Arguments are always in the following order: number of rows, number of columns, inputs, output. Sometimes there will be only one vector size; in this case the first two arguments are replaced by a single argument for the number of elements.

6.2.4.1.1. mul_mv_v

```
mul_mv_v(int n_row, int n_col, float *matrix,
   float *in_vector, float *out_vector)
```

Multiply matrix [n_row by n_col] by in_vector [n_col] to produce out_vector [n_row].

6.2.4.1.2. muladd_mv_v

```
muladd_mv_v(int n_row, int n_col, float *matrix,
   float *in_vector, float *out_vector)
```

Multiply matrix [n_row by n_col] by in_vector [n_col]. Add this product to out_vector and store the result in out_vector [n_row].

6.2.4.1.3. mul_vm_v

```
mul_vm_v(int n_row, int n_col, float *in_vector,
   float *matrix, float *out_vector)
```

Multiply row in_vector [n_row] by matrix [n_row by n_col] to produce out_vector [n_col].

6.2.4.1.4. muladd_vm_v

```
muladd_vm_v(int n_row, int n_col, float *in_vector,
   float *matrix, float *out_vector)
```

Multiply row in_vector [n_row] by matrix [n_row by n_col] and accumulate the result into out_vector [n_col].

6.2.4.1.5. add_vv_v, sub_vv_v, mul_vv_v

```
add_vv_v(int n_ele, float *in_vector_1, float *in_vector_2,
   float *out_vector)
sub_vv_v(int n_ele, float *in_vector_1, float *in_vector_2,
   float *out_vector)
mul_vv_v(int n_ele, float *in_vector_1, float *in_vector_2,
   float *out_vector)
```

Arithmetic on corresponding elements of two vectors.

6.2.4.1.6. mul_svv_m

```
mul_svv_m(int n_row, int n_col, float const,
   float *in_vector_1, float *in_vector_2, float *out_matrix)
```

Out_matrix is set to the outer product of in_vector_1 and in_vector_2 scaled by const.

6.2.4.1.7. muladd_svv_m

```
muladd_svv_m(int n_row, int n_col, float const,
   float *in_vector_1, float *in_vector_2, float *out_matrix)
```

Add the outer product of in_vector_1 and in_vector_2 scaled by const to out_matrix.

6.2.4.1.8. mul_vv_s

```
double mul_vv_s(int n_ele, float *in_vector_1,
```

```
   float *in_vector_2)
```

Inner product of two vectors is returned.

6.2.4.1.9. muladd_sv_v

```
muladd_sv_v(int n_ele, float const, float *in_vector,
   float *out_vector)
```

The elements of in_vector are multipled by const and accumulated into out_vector.

6.2.4.1.10. set_s_v

```
set_s_v(int n_ele, float const, float *out_vector)
```

Set all elements of out_vector to const.

6.2.4.1.11. max_v

```
int max_v(int n_ele, float *in_vector, float *max_element)
```

The largest element of in_vector is placed in max_element. The index of the largest element is returned.

6.2.4.1.12. copy_v_v

```
copy_v_v(int n_ele, float *in_vector, float *out_vector)
```

The in_vector is copied to the out_vector.

6.2.4.1.13. muladd_mb_v

```
muladd_mb_v(int n_row, int n_col, int n_one, float *matrix,
   int *binary_in_vector, float *out_vector)
```

Multiply a matrix by a binary vector with n_one elements that are 1 and (n_col - n_one) that are zero. The binary vector is a list of the element indexes that are 1. The rows of the matrix indicated by the binary_in_vector are added together and accumulated (added) into out_vector.

6.2.4.1.14. muladd_svb_m

```
muladd_svb_m(int n_row, int n_col, int n_one, float const,
   float *in_vector, int *binary_in_vector, float *out_matrix)
```

Sum the outer product of in_vector and a binary_in_vector into out_matrix. The binary vector is a list of the element indexes that are 1 [n_one].

6.2.4.1.15. dsigmoid_vv_v

```
dsigmoid(int n_ele, float *in_vector_1, float *in_vector_2,
   float *out_vector)
```

This function is used in the backpropagation algorithm. The following equation describes its function:

```
out_vector[i] =
```

```
(1.0 - in_vector_1[i]) * in_vector_1[i] * in_vector_2[i]
```

6.2.4.2. Table lookup

These routines support table lookup approximations. They use the following structure to represent the table:

```
struct Table_struct {
  int size;    /* number of floats in table data */
  float scale; /* multiply requested number by scale, and */
  float offset;/* then add offset and truncate to an index */
  float *data; /* into the actual table data itself */
};
```

6.2.4.2.1. make_table

```
struct Table_struct *
make_table( float (*function)(float), float lower_limit,
   float upper_limit, int table_size)
```

Create a lookup table with table_size entries approximating the function from lower_limit to upper_limit.

6.2.4.2.2. make_sigmoid

```
struct Table_struct *
make_sigmoid(float resolution, int table_size)
```

Create a sigmoid lookup table of size table_size. The smallest value in the table is set by resolution.

6.2.4.2.3. table_v_v

```
table_v_v(int n_ele, struct Table_struct *table,
   float *in_vector, float *out_vector)
```

Look up n_ele floating point numbers in table starting at in_vector and put the results in out_vector.

6.2.4.2.4. table_s_s

```
float table_s_s(struct Table_struct *table, float in_number)
```

Lookup in_number in table and return the result.

6.2.4.2.5. sigmoid_v_v

```
sigmoid_v_v(int n_ele, float *in_vector, float *out_vector)
```

Take the sigmoid of each of the n_ele elements in in_vector and store it in the corresponding element of out_vector.

6.2.4.2.6. sigmoid

```
float sigmoid(float x)
```

Return the sigmoid function of x.

6.2.4.3. Ring functions

These functions allow arrays of data to be communicated between processors on the RAP machine. All of these functions must be executing on all processors simultaneously to function properly. The ring will automatically pause a processor that starts one of these functions before all processors are ready to communicate.

6.2.4.3.1. ring_distribute

```
void ring_distribute(int size_of_my_array, void *my_array,
   void *whole_array)
```

The array my_array of size size_of_my_array words will be sent to all other processors. The array whole_array must be at least of size N_NODE*size_of_my_array. After this function is run, the whole_array will contain the concatenation of the my_array from processor 0, followed by the my_array from processor 1, etc.

6.2.4.3.2. ring_write and ring_read

```
void ring_write(int size_of_array, void *array)
int ring_read(int size_of_array, void *array)
```

These routines work together to allow one processor to broadcast data to all of the other processors. To work properly, one processor calls ring_write and while all the others must call ring_read. The ring_read array is filled with the data from the ring_write array. Ring_read returns the number of words read. If the size_of_array for ring_read is smaller then the size_of_array for ring_write then the ring_read array will contain the first part of the ring_write array.

6.2.4.3.3. ring_sync

```
int ring_sync(int code)
```

Ring_sync causes the processor to wait until all processors have called ring_sync. If the code argument is the same for all processors, then 1 is returned. Otherwise, 0 is returned.

6.2.4.3.4. ring_sum

```
void ring_sum(int n_ele_per_node, void *in_vector,
   void *out_vector)
```

This routine is used in the backpropagation step to accumulate the partial errors for each unit. The in_vector is of size N_NODE*n_ele_per_node. The in_vectors from each processor are summed together and out_vector is set to the section of the resulting vector from index NODE_ID * n_ele_per_node to, but not including index (NODE_ID+1) * n_ele_per_node.

6.2.4.3.5. Lowest level ring macros

```
void ring_put_int(int)
int ring_get_int()
int ring_shift_int()
int ring_get_int_nowait()
```

```
int ring_shift_int_nowait()

void ring_put_float(float)
float ring_get_float()
float ring_shift_float()
float ring_get_float_nowait()
float ring_shift_float_nowait()
```

These low level macros allow single words to be written, read and shifted through the communications ring. Using these routines requires considerable care to avoid deadlocks.

The ring_put_int macro sends a single integer word to the outgoing neighbor; the neighbor must do a ring_get_int or ring_shift_int for each word put.

The ring_get_int macro receives a single word from the incoming neighbor.

The ring_shift_int macro receives a single word from the incoming neighbor exactly like ring_get_int except that it also sends the word on to the outgoing neighbor as if a ring_get_int were followed by a ring_put_int.

The ring_get_int_nowait and ring_shift_int_nowait do the same thing as ring_get_int and ring_shift_int except that the processor is not stopped in the case where there is no word ready to be read. In this case, the word from the previous ring_get_int or ring_shift_int is returned.

6.2.4.3.6. ring_get, ring_put

```
void ring_put(int size_of_array, void *array)
int ring_get(int size_of_array, void *array)
```

These routines are designed for communication from a node to its neighboring node. Unlike ring_read and ring_write these routines only require that node (N) runs ring_put and node ((N+1) modulo N_NODE) run ring_get; all other nodes are unconstrained and may compute or communicate using ring_put and ring_get. These routines are useful for pipelined parallelism where each processor has data flowing in and out at independent rates and times. As noted before, there is no buffering of data between nodes, processors will wait until data can be sent and received before continuing.

6.2.4.4. Random functions: Numerical Algorithms

Routines from the book *Numerical Recipes in C* [13] are easily ported to the RAP. The only ones currently in the library are the random functions.

6.2.4.4.1. rand_func

```
void rand_func(int generator_number)
```

The rand_func routine sets which pseudo-random number generator is used for the other functions below. Currently there are three generators numbered 0 through 2.

6.2.4.4.2. seed

```
seed(int seed_value)
```

Set seed for pseudo-random number sequence.

6.2.4.4.3. frandom

```
float frandom()
```

Return the next pseudo-random number from generator.

6.2.4.4.4. random_v

```
random_v(int n_ele, float *out_vector)
```

Set the elements of out_vector to pseudo-random numbers.

6.2.4.5. Panic: error exit

The panic function is used to print fatal error messages.

```
panic("message with printf style formats", arguments, ...)
```

The message is printed to standard error to insure that the user gets it even if RAPMC is not set to catch standard output from that processor. A stack backtrace is also printed for use in debugging.

6.2.5. Linker command file

The linker command file tells the linker where to allocate each section of each object file in memory. Each object file produced by the compiler has three sections: ".text" is the instructions and constants, ".data" is for all global and static local variables that are declared with an initial value, ".bss" is for all global and static local variables that are declared without an initial value.

6.2.5.1. Description

The linker command file is described in detail in the *Texas Instrument Assembly Language Tools* manual [9]. The linker command file described here is provided in the example subdirectory mentioned in the quick start section. There are three parts of the linker command file: arguments to the link command (map file name), the memory block declarations (e.g. SRAM, DRAM, RAM0, and RAM1), and the files and sections to load into each memory block.

The example linker file is listed in Appendix A at the end of this manual. Comments in the file describe each of the above sections in more detail.

6.2.5.2. Moving code into internal RAM or DRAM

It is efficient to move the functions that consume most of the run time into the fastest memory bank. Since it is often the case that over 90 percent of the time is spent on less than 10 percent of the instructions, the small size of the internal on-chip RAM (2048 instructions) is not as limiting as one might expect. The example link.cmd file puts selected library routines into RAM0.

The linker always loads each object file into the same three memory banks. For example, to move the most time critical functions into internal RAM, first make a new source file containing just those functions. Then edit the link.cmd file and add the following line to the RAM0 section group, following the example in the comment there.

```
fast.o(.text)
```

Then add your new file name to the list of object files at the beginning of the make-file.

Do not attempt to try this with (.data) or (.bss) instead of (.text) without understanding all the warnings in the link.cmd file.

6.2.5.3. Changing the stack size

Follow the comments in link.cmd and change value of the STACK_SIZE variable.

6.2.5.4. Map file

The first section of the link.cmd file specifies a map output file name (-m option). The map file contains the address of all global symbols. See the TI linker manual for more details.

7. RAP debugger: RAPMC

7.1. Introduction

RAPMC is a command interpreter that acts as a controller and debugger for a RAP machine. A user may control a set of RAP boards in the aggregate, or may individually control RAP nodes. A user may use multiple RAPMC's, each one controlling a different RAP node. In this section, the computer's output `will be in a Courier font` whereas *what you type will be in an Italics font.*

7.2. Starting up RAPMC

To start up RAPMC, make sure the directory containing `rapmc` is in your path and type:

```
% rapmc
```

One of two things will happen. If you see:

```
The RAP is currently in use
RAP control will be granted when you reach the front of the RAP user queue
Hit <ret> to show RAP user queue or 'q' to quit
>
```

then someone else is using the rap. A queue is maintained which you may look at at by hitting <*return*>. When you reach the front of the queue, you will automatically get RAP control (i.e. any startup script will automatically execute and afterwards you will get the RAPMC prompt). If you do not wish to wait to reach the front of the queue, hit *q*. If the user queue is empty, or if you wait to reach the front of the queue, after the startup script has completed, you will see:

```
RAP Master Commander. Version
Copyright (C) 1990 International Computer Science Institute
Connected to host (rap_host_name).
RAP ready with 4 nodes.
Type "help" for a list of commands.
0>
```

If a file named `.rapmcrc` exists in the current directory, RAPMC will use it as a startup script (the commands listed in that file will be executed). Once done, you may start entering RAPMC commands on the RAPMC command line.

A user may start up multiple RAPMC's. Once the user reaches the front of the user queue, all RAPMCs started by this user will get RAP control. Each RAPMC will become a different

RAPMC session that can be seen by looking at the user queue. RAPMCs need not all be started from the same UNIX host. With this feature, a user may start up many different RAPMC's and have each one affect a different RAP node. This can be useful for logging in from home to check work in progress.

7.3. RAPMC command line arguments

The usage of RAPMC is given by:

```
rapmc [-f <startup script file>] [<Host Name>]
```

Where the optional `-f <startup script file>` specifies the name a file to be used in place of `.rapmcrc` and `<Host Name>` is the host name that supports a RAP (the default host is currently `<really.berkeley.edu>`). If you give the `-f` option to RAPMC (or have a `.rapmcrc` in the current directory), but another user is currently using the RAP, your script will be executed when you reach the front of the user queue (even if you background RAPMC with `^Z` and `bg`). This is analogous to submitting a batch job.

7.4. RAPMC Scripts

RAPMC may accept commands from a file. The contents of a file must be valid RAPMC commands or comments. Comments in the file begin with the # character and last until the end of the line. RAPMC script files are `.rapmcrc`, a file given with the `-f` option, or a file given with the `source` command. Currently, scripts may not be nested.

7.5. RAPMC Simulated Batch Job Queue

A user may simulate a batch job by entering commands into a script and backgrounding RAPMC when run. Given the following script called `my_script`:

```
# this is a RAPMC script
*           # change to affect all nodes
load yo     # load the yo program
0 > out0    # send each node's standard output to a different file
1 > out1
2 > out2
3 > out3
# run yo with yo's argv[1] = "yo_param1" and argv[2] = "yo_param2"
run yo_param1 yo_param2
wait        # wait for all nodes to finish running
quit        # quit this rapmc so someone else can get their batch job done.
```

you may run RAPMC from the unix command line as:

```
% rapmc -f my_script &
```

To redirect standard error output to a file named `error_output_file`, you may run RAPMC as:

```
% rapmc -f my_script >& error_output_file &
```

At this point, your RAP job will run to completion (even if you log off). When the script is done, the RAP will be free for the next user in the queue.

7.6. RAPMC commands

The RAPMC prompt `0>` shows the default node (zero in this example) followed by `>`. The default node specifies which node the command will affect. A default node of `*` implies all

nodes. Commands may be prefixed with a node number (or *) to select which node(s) to affect. If a command is not prefixed with a node number, the default node is used. If a command is prefixed with a node number, the prefixed node number becomes the command's current node. For example:

```
0>node 3          # change to node 3
3>4               # change to node 4
4>reset           # this resets node 4, current node is 4, default node is 4
4>3reset          # this resets node 3, current node is 3, default node is 4
4> *reset         # this resets all nodes, current node is *, default node is 4
```

Only enough characters to make a command unique need be specified on a command line. For example:

```
0>ru              # run
0>ex              # examine
```

Two special cases are `l` for `load` and `r` for `run`.

7.7. RAPMC RAP Manipulation Commands

7.7.1. load

```
load [file]
l [file]
```

Load `file` to current RAP node(s). `file` must exist in the current directory, or a full path must be specified. It must be a valid TMS320C30 executable file. If `file` is not given, the file specified at the previous load command will be used.

7.7.2. run

```
run [arguments]
go [arguments]
r [arguments]
```

Start running current node. If [arguments] are given, use them, otherwise, use previously given arguments.

7.7.3. Load & Run command

```
TMS_executable_file_name [arguments]
```

Load `TMS_executable_file_name` to current RAP node(s) and start running with given arguments. This is a shorthand for:

```
0> load TMS_executable_file_name
0> run [arguments]
```

Note that this only works if the executable does not have the same name as a RAPMC command.

7.7.4. examine

```
examine [[format] [Node_address]]
```

`examine` allows you to peek and poke at a rap node's memory. You may only examine one node at a time, thus the current node number can't be star. `format` specifies the available forms with which you may peek at memory. They are `x` for hex, `d` for decimal, `f` for floating point, `c` for character, and `i` for TMS320C30 instructions. `Node_address` is the starting node address to be examined. If `Node_address` isn't given, the previous node address will be used with the given format. If both `format` and

`Node_address` aren't given, the previous format and node address will be used. `examine` will place you into a loop where you may modify the given address location by entering in a hex, decimal, or floating point value, go on to the next memory location by hitting <*return*>, or leave by typing `q`. Hex values are specified by a preceding "`0x`". Floating point values are specified by an embedded "`.`" or `e` (e.g. 34., 2.3, 2e1). For example:

```
2>ex x 0x0
0x0 : 0x2FA <ret>
0x1 : 0x3 0x30
0x2 : 0x403 q
2>ex    # use previous format and address
0x0 : 0x2FA <ret>
0x1 : 0x30 <ret>
0x2 : 0x403 q
2>ex i      # use new format (instructions) but previous address
0x0 : ABSF R0,R0 <ret>
0x1 : MPYF3 R1,R2,R3 q
2>
```

7.7.5. quit

```
quit
```

Leave RAPMC. If this is the last RAPMC session owned by you, a `reset` will occur and the next person in the user queue will get the RAP. Otherwise, `quit` will not affect any other RAPMC sessions.

7.7.6. kill

```
kill
```

This will leave RAPMC like `quit` but all other RAPMC sessions owned by you on any host will be terminated. After a `kill`, the next person in the user queue will get the RAP. `kill` will automatically reset the RAP with a `reset` command.

7.7.7. reset

```
reset
```

`reset` will reset the current node(s) to a known state. It will destroy the current running job. Data in RAP memory may be wiped out.

7.7.8. shelve

```
shelve
```

Quit RAPMC but keep the RAP job running. All output redirected by this RAPMC session will stop. The next person in the user queue will **not** get the RAP. No other RAPMC sessions owned by you will be affected. After a `shelve`, you can get back by running RAPMC again.

7.8. RAPMC I/O Commands

7.8.1. redirect, >

```
> fileName
redirect fileName
```

This redirects the current node's standard output to the file given by `fileName`.

7.8.2. append, >>

```
>> fileName
append fileName
```

This appends the current node's standard output to the file given by `fileName`.

7.8.3. pipe, |

```
| command [command_args]
pipe command [command_args]
```

`pipe` will pipe the current node's standard output to the UNIX program given by command. If `command_args` are given, they will be given as arguments to the program.

7.8.4. getback, <

```
getback
<
```

This will return the standard output of the current node to RAPMC. If the nodes standard output was redirected to a file, the file will be closed; if it was piped to a program, the program will receive an EOF.

7.8.5. miss

```
miss
```

`miss` will cause all standard output from the current node to be missed. If the node's standard output has been redirected to a file or piped to a program, the file or program will stop receiving the data until a `catch` command for the current node is performed.

7.8.6. catch

```
catch
```

This will catch all output from the current node.

7.8.7. echo

```
echo string
```

`echo` sends its string to the terminal. It is most useful in scripts to document what is happening.

7.8.8. input

```
input text
```

This will send `text` to the standard input of the current node. If the current node has not requested input (e.g. It hasn't yet executed a scanf() or something similar) the text will be queued.

7.9. RAPMC Miscellaneous Commands

7.9.1. help, ?

Print a help summary of all RAPMC commands.

7.9.2. chdir, cd

```
chdir path
cd path
```

Change the current working directory for the current node(s) and RAPMC.

7.9.3. lcd

```
lcd path
```

Change only the local RAPMC current working directory.

7.9.4. lpwd

```
lpwd
```

Print only the local RAPMC current working directory.

7.9.5. node

```
node n
n
```

Set the default node to `n`. The current node for future commands that do not have a node number preceding them will be the default node.

7.9.6. pwd

```
pwd
```

Print the current working directory of the current rap node.

7.9.7. source, .

```
source fileName
. fileName
```

Send the contents of the file given by fileName to RAPMC's input. Return control to the keyboard at the end of the file or when the user hits *<control>-C*.

7.9.8. users

```
users
```

Display the user queue. This will show all users waiting for the RAP and show how many sessions you (and each of the waiting users) have open.

7.9.9. background

```
background
bg
```

The `background` command is similar to hitting `<control>-Z` and `bg` which normally will suspend and background the process. `background`, however, will additionally prohibit RAPMC from trying to read input from the terminal. Therefore, RAPMC will not get stopped, the message

```
[1]  + Stopped (tty input)  rapmc
```

will not appear, and RAPMC will be able to continue processing all incoming RAP data (i.e. RAP output that has been redirected to files or piped to programs will continue being redirected or piped). This is not possible using `^Z` and `bg` because RAPMC will be stopped. To foreground a `background`'ed RAPMC, use `fg` or `%<job number>` as you normally would after a `^Z bg`.

A script may contain a `background`. The effect will be to leave RAPMC running in the background once the entire script has been processed.

When you type `background`, the shell will not show the RAPMC process as a background job. It is still running, however. Don't let the shell fool you. For example:

```
0> bg
```

```
Stopped
% Continuing RAPMC in background      # RAPMC message when it continues
% jobs
[1] + Stopped     rapmc               # rapmc is actually still running here
% fg
rapmc
0>
```

`background` will not work when running RAPMC under the Bourne shell since the Bourne shell has no job control.

7.9.10. wait

```
wait
```

This will pause until the current rap node has exited from the previous run (called exit or returned from main) or hits *<ctrl>-C*. It is most useful in scripts where you don't want to continue until the previous run has completed.

8. Assembly language

To maximize performance, inner loops are often rewritten in assembly language. Instead of writing assembler code from scratch, the compiler's output can be used as a starting point with the following procedure:

1. Make the inner loop into a separate function in its own file.
2. Compile with the "-k" option to create a ".asm" file.
3. Edit the makefile to add the file name with a ".o" ending (instead of ".asm") to the "OBJ_FILES" list.
4. Optimize the assembly language file. Test after each change.

The following are some hints about where in the TI documentation to look for optimization tricks.

8.1. Register allocation

The C compiler uses registers as indicated in the following table:

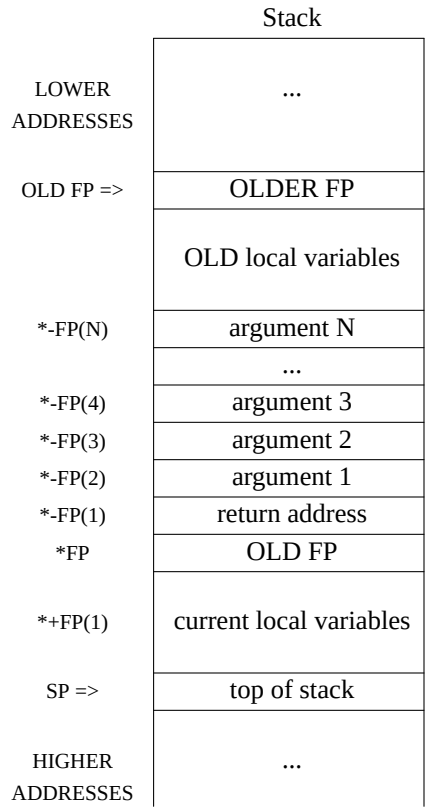
Register usage by TI C compiler				
Register	Integer	Float	Temporary	Special functions
R0	yes	yes	yes	Used to return int or float from function
R1	yes	yes	yes	
R2	yes	yes	yes	
R3	yes	yes	yes	
R4	yes	no	no	
R5	yes	no	no	
R6	no	yes	no	
R7	no	yes	no	
AR0	pointer	pointer	yes	
AR1	pointer	pointer	yes	
AR2	pointer	pointer	yes	
AR3	pointer	pointer	no	Stack Frame pointer (see text below)
AR4	pointer	pointer	no	
AR5	pointer	pointer	no	
AR6	pointer	pointer	no	
AR7	pointer	pointer	no	
IR0	yes	no	yes	Used for pointer offsets over 255
IR1	yes	no	yes	Used for pointer offsets over 255
SP	pointer	pointer	no	Stack pointer - special register
RC	yes	no	yes	Used for structure copy/assignment
RS	yes	no	yes	Used for structure copy/assignment
RE	yes	no	yes	Used for structure copy/assignment

Stack structure and compiler register usage are discussed in detail in the *TMS320C30 C Compiler Reference Manual* [8] chapter 4 on the run-time environment. The most important aspects are summarized below.

By definition, temporary registers can be changed by any function call; they need not be saved or restored by a function. Note that the compiler does sometimes hold values in "temporary" registers across C statements if there is no chance of a function call. Registers that are not temporary also can be used by the assembly language programmer so long as they are restored to the value they had at the start of the function.

When saving and restoring non-temporary registers, it is important to use the correct push and pop (PUSH/POP or PUSHF/POPF) depending on the data type (int or float) in the register. This is because registers R0-R7 are 40 bits long for floating point numbers and 32 bits long for integers. When a 40 bit float is moved to or from 32 bit memory, 8 low order bits of precision are lost; only the most significant 32 bits are saved. Thus, it is a good idea to keep floating point intermediate results that are of critical precision in registers. For integers, only the low order 32 bits of the register are used. To save and restore a register that could be either floating point or integer (as for example in an interrupt handler), you must first PUSH then PUSHF to save, and POPF and POP to restore all 40 bits.

The frame pointer AR3 is also called "FP" in the assembler. It is used to access arguments to the current function and local variables declared inside the current function. The first argument to the function is at address FP-2 (or in assembler syntax "*-FP(2)"). The second argument is at FP-3, etc. Local variables start at FP+1. This is illustrated by the diagram below:



8.2. Pipeline delays

The TMS320C30 processor is often in different phases of executing up to four instructions at once. This is not directly visible to the assembly language programmer. The phases that an instruction goes through are:

1. Fetch: get the instruction from memory
2. Decode: figure out what actions are specified by the instruction
3. Read: read any data that may be needed for the operation
4. Execute: actually does it and stores any results

There are cases where the processor inserts delay cycles because of this pipeline. If the execution phase of an instruction changes an address that is needed by the read phase of the next instruction, the next instruction is forced to wait for the previous one to complete before starting its read cycle. The C30 is not very smart about when this sort of thing might occur and is overly protective. Changing an address register will cause this sort of delay if the following instruction uses ANY other (possible unchanged) address or index register. To avoid these delays, one of the following tricks might be useful:

1. Addressing modes that change address registers as a side effect will not cause these delays. For example, use the addressing mode that adds an index register to the address register. Note that changing index registers causes the same delays as changing address registers.

2. Put instructions that are internal or use only the stack after address register loads. For example, pre-load a register with -2 so that later on it can be added to the contents pointed to by an address register rather than loading indirect from the address register and then subtracting two.

3. Once you start loading address registers, try to do all such loads in one block.

The TMS320C30 has three register groups that are independent so far as pipeline delays are concerned. If an instruction writes to a register in one group and the next instruction uses any register in the same group (for example, an indirect load or store) a pipeline delay will occur. However, registers in other groups can still be used without delay. The three register groups are:

1. Address registers (AR0 to AR7), Index registers (IR0 and IR1), Block size register (BK)

2. Data Pointer register (DP)

3. Stack Pointer

This is just one example of pipeline delays. There are many more described in Chapters 10 and 8 of the *TMS320C3x User's Guide* [11]. The following table summarizes the cases where delays occur. The last two columns indicate the user's guide page and figure numbers where more details may be found.

Instructions sequences that cause pipeline delays on the TMS320C30				
Sequence	Instruction	Pipeline delays	Description page	Example figure
PC-1	SRC in register group N	1	10-7	10-3
PC	Any use of register group N			
PC-1	DST in register group N	2	10-7	10-4
PC	Any use of register group N			
PC-2	DST in register group N	1	10-7	none
PC-1	Any instruction			
PC	Any use of register group N			
PC-2	SRC1 and SRC2 in on-chip RAM block N	1	10-9	10-5
PC-1	Any instruction			
PC	PC is in on-chip RAM block N			
PC-1	DST is not register	1	10-11	10-8
PC	SRC1 and SRC2 are not registers			
PC-1	DST1 and DST2 are not registers	1	10-11	10-9
PC	SRC is not register			
PC	SRC1 is external SRC2 is external or internal	1	10-17	none
PC	DST1 is internal or external DST2 is external	1	10-18	parallel store
PC	SRC3 is external SRC4 is internal or external	1	10-19	parallel MAC
PC-1	SRC is external	2	8-5	none
PC	DST is external			
PC-1	DST is external	1	8-5	none
PC	SRC is external			

8.3. Delayed branch

Regular branch instructions always add three delay cycles. The delayed branch instructions B<cond>D and DB<cond>D only add one cycle for the instruction itself. The three instructions following the delayed branch are executed before the branch actually happens. Some of these bonus instructions can be NOPs.

8.4. Conditional load

Conditional load instructions LDF<cond> and LDI<cond> are particularly useful for cases where the minimum or maximum needs to be determined. They do not add any extra delays.

8.5. Repeat block

The most time critical loops should use the RPTB or RPTS instructions. These allow count down loops to run without any loop overhead. Note that these always run the loop at least once. Also, the count should be the desired number of loops minus one.

Unrolling does not speed up RPTB loops since there is no loop overhead. However, when using the parallel instructions in a loop it often makes sense to unroll one loop and split it before and after the RPTB or RPTS block. See the examples at the end of the *TMS320C3x User's Manual* [11].

Since RPTB loops can not be nested, time critical loops that already contain one or more RPTB loops must use other tricks such as unrolling and/or the decrement and delayed branch on condition instruction (DB<cond>D).

8.6. Addressing modes

Check the addressing modes, some have useful side effects. There are delay cycles added due to changes to address registers caused by these side effects.

8.7. Parallel instructions

The parallel instructions only allow a very limited number of addressing mode combinations. In general check the "Operands" section of the instruction description in the *TMS320C3x User's Manual* [11] to be sure that the addressing mode you want is possible. The four basic addressing modes are register, direct, indirect and immediate. Direct is always indicated by an "@" before a name. This mode addresses up to 64k of memory directly; the upper address bits are set by the low order 8 bits of the DP register. Indirect always starts with a "*"; there are many of these. Immediate allows small constant values to be encoded in the instruction.

9. UNIX simulated RAP libraries

The simulated RAP library that runs under UNIX is useful for development and debugging of RAP programs. All functions in the RAP library are supported in the simulated library.

To use the simulated RAP libraries, the rap.h file should be taken from the "sim" subdirectory of the RAP root. The file sim/rapsim.c should be compiled and linked with your program.

Since this simulation runs in one UNIX process, it can only be used to test SIMD programs that work with one processor. In this simulation, the number of nodes (N_NODE) is set to one and the current node number (NODE_ID) is set to zero. The simulation library has dummy ring functions that just returns.

10. An application: multi-layer perception training program (mlp)

The mlp program is a simulator for feed-forward backpropagation networks. It has been used to train phoneme classifiers for continuous speech recognition.

The mlp program realizes the following goals:

1. Runs efficiently on the RAP
2. Efficiently scales up to multiple RAP boards
3. The same source code runs on UNIX workstations

4. The user does not have to do any programming to setup new network structures and training parameters.
5. All parameters needed to recreate an experiment are recorded in the log file
6. Flexibility to run a wide variety of network structures and training procedures
7. Trained network can be easily embedded into other programs (e.g. dynamic programming)

The program takes one command line argument: the parameter file name. The parameter file contains a list of parameter names and values that specify the input pattern database format, input pattern preprocessing, network structure, and training procedure.

The mlp program consists of three independent parts: a library of network simulation routines, a network builder and a training program.

The library implements two types of objects: layers and connections. A layer is an array of units. In the current mlp program, layers come in two flavors: analog (continuous valued units) and binary (unit output is 1 or 0). Connections also come in two flavors: cross-connection between all units of two layers (weight matrix) and "bus" 1-to-1 connection of a range of units in a layer to a range of units in another layer. Any set of connections can also share weights. Once a feed-forward network (directed acyclic graph) has been built by creating and connecting layers, the "forward" and "backward" library routines can be called to run the two parts of the backpropagation algorithm. The network library calls the RAP library of hand optimized assembler routines for doing matrix and vector operations. Most of the CPU cycles are spent in these routines for layers larger than twice the number of processors.

The net builder is responsible for reading the parameter file and calling the network library. It is a separate module so that other programs that utilize the final trained mlp can re-create the network from the same parameter file that trained it.

For example, to use a trained network from inside another program:

```
net_build("parameter_file_name");
net_read(net, "weight_file_name");

  ....

for(...) {
   ...
   net_input(net, input_array);
   net_forward(net);
   net_output(net, output_array);
   ...
}
```

In the parameter file, each layer is assigned a number. Layer 0 is always the input layer and layer 1 is always the output layer. The net structure is specified by a list of layer number pairs to connect. There is also support for arrays of layers. An array of layers can be connected to another array of layers by a sliding "receptive field" window of shared connections.

The training program preprocesses the data (sorting, normalizing, selecting, table lookup, linear functions, etc.) and then starts the training/test cycles. Part of the input data is never used for training, but instead is used to gauge the degree of generalization. Performance on this cross-validation set is used along with specifications in the parameter file to determine when to stop training and when to change the learning rate. The training program also collects statistics during training and after training is complete (confusion matrix, etc.). There is also support for training a net and then changing the network structure (new layers, make or break connections, etc.) and/or parameters and re-training (possibly with different output targets). This allows a net to be built out of several pre-trained nets.

When the training data becomes too large to store in DRAM on the RAP, a pre-digested version of the data file is created and repeatly read on each training cycle. As more RAP boards are added, the capacity for input data in DRAM goes up proportionally (the ring is used to distribute input patterns).

11. Direct communication between C++ programs running under UNIX and the RAP

RAPMC is one example of a UNIX program that uses the rapClient interface. The C++ object class called rapClient can be subclassed to allow users to embed direct control and communications with the RAP in their own C++ UNIX programs. To do this, one must include the following files:

```
#include "rapClient.h"
#include "rapProto.h"
#include "rapFacts.h"
```

For a simple example of how to use this interface to load a program on the RAP and communicate with it, look in the example subdirectory of the RAP software distribution.

12. Work in progress

A C++ environment is being developed for the RAP using the C++ preprocessor from AT&T. In this environment vectors or matrices are object classes that inherit from a more general class of distributed data object. The distribution of data is encapsulated so that SIMD programmers can use the machine as if it where one large array processor with one large shared memory. On top of this we plan to build a new backpropagation training program tentatively called CLONES (Connectionist Layered Object-oriented NEtwork Simulator) that runs on the RAP and SUN workstation [7].

13. Appendix A: example program

13.1. main.c

```
#include <rap.h>

main(ac,av)
int ac;
char **av;
{
  int i;

  printf("\nYO!!  WHASSUP?\n\n");

  printf("This is processor %d on a RAP with %d processors\n",
    NODE_ID, N_NODE);

  printf("got %d command line arguments\n", ac);

  for(i=0; i<ac; i++)
    printf("argument %d is %s\n", i, av[i]);

  if (ac > 2) {
     /* 2 or more arguments,
      * change to directory named in second argument
      */
     printf("changing to directory: %s\n", av[2]);
     cd(av[2]);
  }

  if (ac > 1) {
       /* 1 or more arguments:
        * print out file named in first argument
        */
     file = fopen(av[1],"r");

     if (file == NULL)
        panic("can not open file %s\n", av[1]);

     while( fgets(buf, sizeof(buf), file) != NULL )
        printf("read: %s", buf);

     fclose(file);
  }

  printf("\nGOTTA GO NOW!\n");
  exit(123);
}
```

13.2. makefile

The following is a listing of the makefile from the example subdirectory of the RAP software distribution.

```
# This makefile compiles and links an example program
# for the RAP machine.
#
# Your shell PATH variable must
# contain the path for the RAP bin subdirectory.
#
# Put the names of your object files here (C or assembler):
OBJ_FILES = yo.o

# put the name of your output executable here:
OUT_FILE = yo

# root directory for rap software distribution
# SET ROOT_DIR TO THE PATH TO THE TOP OF THE RAP DISTRIBUTION DIRECTORY
ROOT_DIR = <replace this with the distribution directory>

# C compiler options
# (see table 2-1 in the TMS320C30 C Compiler Reference [8])
# -q option is for "quiet" mode (no banners, etc.)
# -o2 option turns on the optimizer
# -mf option allows pointers outside of the 64k base page
C_FLAGS = -q -o2

# Assembler options
ASM_FLAGS =

# Linker options
# (see table 9-1 in the TMS320C30 Assembly Language Tools [9])
# the "-q" option is for "quiet" mode (no banners, etc.)
LNK_FLAGS = -q


# subdirectories of rap software root directory
BIN_DIR = $(ROOT_DIR)/bin
LIB_DIR = $(ROOT_DIR)/lib
INCLUDE_DIR = $(ROOT_DIR)/include
TI_INCLUDE_DIR = $(ROOT_DIR)/include/ti


# executables for compiler and linker
CC = $(BIN_DIR)/cl30
LNK = $(BIN_DIR)/lnk30

# generic link step
$(OUT_FILE): $(OBJ_FILES) $(LIB_DIR)/rap.lib
     ${LNK} link.cmd $(OBJ_FILES) \
            $(LNK_FLAGS) -l $(LIB_DIR)/rap.lib -o $(OUT_FILE)

# generic c compile step
.c.o:
     $(CC) $(C_FLAGS) -I$(INCLUDE_DIR) -I$(TI_INCLUDE_DIR) -c $<
     mv $*.obj $*.o

# generic assembly steps
```

```
.s.o:
    $(CC) $(ASM_FLAGS) -c $<
    mv $*.obj $*.o

.asm.o:
    $(CC) $(ASM_FLAGS) -c $<
    mv $*.obj $*.o
```

13.3. link.cmd: linker command file

The following is a listing of the link.cmd file from the example subdirectory of the RAP software distribution.

```
/* This linker command file is used by the TMS320C30 linker
** to control placement of program sections in memory on the RAP.
**
** This file consists of 3 parts:
** (1) arguments to the linker (see TMS linker manual for details)
** (2) memory block map
** (3) one memory group declaration for each memory block
*/

/* Linker arguments:
**   "-m rap.map" instructs the linker to write a
**                memory map into file rap.map
*/
-c
-m rap.map

/* MEMORY BLOCK MAP: gives names to ranges of memory addresses.
** Each line inside the MEMORY clause will create a named region
** of memory for use in this link.cmd file.
*/
MEMORY
{
    RAM0: org = 0x809800, len = 0x400      /* on-chip ram 1K word */
    RAM1: org = 0x809C00, len = 0x400      /* on-chip ram 1K word */
    SRAM: org = 0x000000, len = 0x10000    /* static ram 64K word */
    DRAM: org = 0x400000, len = 0x100000   /* dynamic ram 1M word */
}

/* The following SECTIONS determine what parts of the RAP
** program will be loaded into each of DRAM, SRAM, RAM0, and RAM1.
**
*** IMPORTANT WARNING:
*** Because of the limitations on the TI320C3x direct addressing mode,
*** ALL .data and .bss sections MUST be contained in one
*** 64k range of addresses.
*** DO NOT attempt to move selected variables to internal RAM or DRAM.
*** (use rap_malloc to allocate variables in any memory block)
*** This warning does not apply to code sections (.text) or the stack.
**
*/
SECTIONS
{
     GROUP:
     {
     /* Dynamic RAM: large and slow.
     **
     ** DRAM can be used for slow instructions (.text)
     ** and from malloc (large arrays)
     ** Putting .data, .bss, or stack in DRAM greatly increases
     ** run time.
     */
          slow_text: {
               /* put object files here for slow execution.
                * for example:
```

```
            *    your_file_name.o(.text)
            */
     }

     /* MUST BE LAST IN DRAM GROUP (marks start of malloc area) */
     DRAMend: {
          _DRAMfree = .;
          _DRAMend = 0x4fffff;
     }
} > DRAM

/* Static RAM: medium size and speed.
** SRAM can be used for data and faster text.
** The stack is in SRAM since it would not fit in
** the internal memory.
*/
GROUP:
{
     .sysdata: {}   /* system globals, such as N_NODE */

     .text: {}   /* all instructions not placed elsewhere */

     stack: {
          /* NOTE: stack size must be at least 3k
           * The stack is used to download internal ram, and
           * command line arguments.
           * WARNING: stack overflows are not detected!!
           *
           * To change the stack size, change the
           * _STACK_SIZE assignment below:
           */
          _STACK_SIZE = 01000h;

          _STACK_START = .;
          . += _STACK_SIZE;
          _STACK_MAGIC = 0x12345678;  /* detect stack overflow */
     }

     /* NOTE: Because TI320C30 direct addressing can only access
      * 64k, ALL data and bss sections must be contained
      * in one region that is less than or equal to 64k.
      *
      * The following two lines can be moved together into
      * the DRAM or internal RAM sections, but should
      * not be separated. No extra .data or .bss lines
      * should be added.
      */
     .data: {}  /* all initialized variables */
     .bss: {}  /* all uninitialized variables */


     /* MUST BE LAST IN SRAM GROUP (marks start of malloc area) */
     SRAMend: {
          _SRAMfree = .;
          _SRAMend = 0xffff;
     }
} > SRAM

/* Internal on-chip RAM bank 0
```

```
     ** Fastest memory, limited size
     */
     GROUP:
     {
          .library: {}     /* matrix and ring library routines */

          /*
           *  To put the instructions of an object file
           *  into fast on-chip RAM, add a line to the
           *  following block as shown in the following comment.
           */
          fastest_text: {
          /*  object_file_name.o(.text) */
          }

          /* MUST BE LAST IN RAM0 GROUP (marks start of malloc area) */
          RAM0end: {
               _RAM0free = .;
               _RAM0end = 0x809bff;
          }
     } > RAM0

     /* Internal on-chip RAM bank 1
     ** Fastest memory, limited size
     */
     GROUP:
     {

          /*
           *  To put the instructions of an object file
           *  into fast on-chip RAM, add a line to the
           *  following block of the form shown in the comment.
           */
          more_fastest_text: {
          /*  object_file_name.o(.text) */
          }

          .onchip: {}  /* routines in library that must be internal */

          /* MUST BE LAST IN RAM1 GROUP (marks start of malloc area) */
          RAM1end: {
               _RAM1free = .;
               _RAM1end = 0x809fff;
          }
     } > RAM1

     /* hardware vector table and software HOST_ROOT pointers */
     vectors 0x0: {}

     /* hardware register locations */
     hardware:
     {
          LDR_REG_ADDR = 0x800000;
     }
}
```

14. Appendix B: standard C functions supported by the RAP

The following is a list of the standard C routines supported in the RAP environment. For more information, use the "man" command or check any standard C library manual.

14.1. alphanumeric

```
int isalnum(char)
int isalpha(char)
int isascii(char)
int iscntrl(char)
int isdigit(char)
int isgraph(char)
int islower(char)
int isprint(char)
int ispunct(char)
int isspace(char)
int isupper(char)
int isxdigit(char)
char toascii(char)
char tolower(char)
char toupper(char)
```

14.2. math

```
double acos(double)
double asin(double)
double atan(double)
double atan2(double y, double x)
double ceil(double)
double cos(double)
double cosh(double)
double exp(double)
double fabs(double)
double floor(double)
double fmod(double x, double y)
double frexp(double value, int *exp)
double ldexp(double x, int exp)
double log(double)
double log10(double)
double modf(double value, int *iptr)
double pow(double x, double y)
double sin(double)
double sinh(double)
double sqrt(double)
double tan(double)
double tanh(double)

int abs(int)
div_t div(int numer, int denom)
long labs(long)
ldiv_t ldiv(int numer, int denom)
int rand()
void srand(int seed)
```

14.3. memory allocation

```
void *calloc(int nmemb, int size)
void free(void *ptr)
void *malloc(int size)
void *realloc(void *ptr, int size)
```

14.4. files

```
FILE *fopen(char *name, char *modes)
FILE *freopen(char *name, char *modes, FILE *stream)
int fflush(FILE *stream)
int fclose(FILE *stream)
FILE *tmpfile()
char *tmpnam(char *s)
void setbuf(FILE *stream, char *buf)

fprintf(FILE *stream, char *format, ...)
printf(char *format, ...)

fscanf(FILE *stream, char *format, ...)
scanf(char *format, ...)

int fgetc(FILE *stream)
char *fgets(char *s, int n, FILE *stream)
int fputc(int c, FILE *stream)
int fputs(char *s, FILE *stream)
int getc(FILE *stream)
int getchar()
char *gets(char *s)
int putc(int c, FILE *stream)
int putchar(int c)
int puts(char *s)
int ungetc(int c, FILE *stream)

int fread(void *ptr, int size, int nobj, FILE *stream)
int fwrite(void *ptr, int size, int nobj, FILE *stream)

int fseek(FILE *stream, long offset, int origin)
long ftell(FILE *stream)
void rewind(FILE *stream)

void clearerr(FILE *stream)
int feof(FILE *stream)
int ferror(FILE *stream)

int open(char *name, int mode)
int read(int file_num, void *buffer, int size)
int write(int file_num, void *buffer, int size)
int close(int file_num)
int lseek(int file_num, int offset, int flag)
int cd(char *path)
```

14.5. string conversions

```
sprintf(char *s, char *format, ...)
sscanf(char *s, char *format, ...)

int atoi(char *string)
double atof(char *string)
int atol(char *string)
int ltoa(long n, char *buffer)
double strtod(char *nptr, char **endptr)
int strtol(char *nptr, char **endptr, int base)
int strtoul(char *nptr, char **endptr, int base)
```

14.6. memory

```
char *movmem(char *src, char *dest, int count)
void *memchr(void *s, int c, int n)
int memcmp(void *s1, void *s2, int n)
void *memcpy(void *s1, void *s2, int n)
void *memmove(void *s1, void *s2, int n)
void *memset(void *s, int c, int n)
```

14.7. strings

```
char *strcat(char *s1, char *s2)
char *strchr(char *s, int c)
int strcmp(char *s1, char *s2)
int *strcoll(char *s1, char *s2)
char *strcpy(char *s1, char *s2)
int strcspn(char *s1, char *s2)
int strlen(char *s)
char *strncat(char *s1, char *s2, int n)
int *strncmp(char *s1, char *s2, int n)
char *strncpy(char *s1, char *s2, int n)
char *strpbrk(char *s1, char *s2)
char *strrchr(char *s, char c)
int strspn(char *s1, char *s2)
char *strstr(char *s1, char *s2)
char *strtok(char *s1, char *s2)
```

14.8. variable number of function arguments

```
type va_arg(va_list ap, type)
void va_end(va_list ap)
void va_start(ap, parmN)
```

14.9. time

```
char *asctime(struct tm *timeptr)
char *ctime(struct tm *timeptr)
double difftime(int time1, int time0)
struct tm *gmtime(int time)
struct tm *localtime(int time)
```

```
int mktime(struct tm *timeptr)
int strftime(char *s, int maxsize, char *format,
   struct tm *timeptr)
```

14.10. misc

```
void abort()
void atexit(void (*func)())
void *bsearch(void *key, void *base, int nmemb,
   int size, int (*compar)())
void exit(int status)
int getopt(int argc, char **argv, char *opts)
void longjmp(jmp_buf env, val)
void qsort(void *base, int nmemb, int size, int (*compar)())
int setjmp(jmp_buf env)
```

15. Acknowledgments

Hardware was contributed to this project by Cypress Semiconductor, Xilinx Inc., and Toshiba America. Texas Instruments lent in-circuit emulators for the initial debugging. The International Computer Science Institute and its sponsors are gratefully acknowledged for supporting this work. UNIX is a trademark of AT&T.

16. References

[1] J. Beck, ‘‘The Ring Array Processor (RAP): Hardware,’’ International Computer Science Institute TR-90-048, 1990.

[2] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, & J. Beer, ‘‘The RAP: a Ring Array Processor for Layered Network Calculations,’’ *Proc. of Intl. Conf. on Application Specific Array Processors*, pp. 296-308. IEEE Computer Society Press, Princeton, N.J., 1990.

[3] J. Bilmes & P. Kohn, ‘‘The Ring Array Processor (RAP): Software Architecture,’’ International Computer Science Institute TR-90-050, 1990.

[4] N. Morgan, ‘‘The Ring Array Processor (RAP): Algorithms and Architecture’’ International Computer Science Institute TR-90-047, 1990.

[5] N. Morgan, C. Wooters, H. Bourlard, & M. Cohen, ‘‘Continuous Speech Recognition on the Resource Management Database using Connectionist Probability Estimation,’’ ICSI Technical report TR-090-044, also to be published in proceedings of ICSLP-90, Kobe, Japan.

[6] N. Morgan, H. Hermansky, C. Wooters, P. Kohn, & H. Bourlard, ‘‘Phonetically-based Speaker-Independent Continuous Speech Recognition Using PLP Analysis with Multilayer Perceptrons,’’ submitted to *IEEE Intl. Conf. on Acoustics, Speech, & Signal Processing*, Toronto, Canada, 1991.

[7] P. Kohn, ‘‘CLONES: A Connectionist Layered Object-oriented NEtwork Simulator,’’ In preparation.

[8] Texas Instruments, *TMS320C30 C Compiler Reference Guide*, SPRU034A, 1989.

[9] Texas Instruments, *TMS320C30 Assembly Language Tools*, SPRU035, 1988.

[10] Texas Instruments, *TMS320C30 Simulator*, SPRU017, 1989.

[11] Texas Instruments, *Third-Generation TMS320 User’s Guide*, SPRU031, 1988.

[12] D.E. Rumelhart, G.E. Hinton & R.J. Williams, ‘‘Learning Internal Representations by Error Propagation,’’ in *Parallel Distributed Processing. Exploration of the Microstructure of Cognition.* vol. 1: Foundations, ed. D. E. Rumelhart & J. L. McClelland, MIT Press, 1986.

[13] W. Press, B. Flannery, S. Teukolsky & W. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1986.

Table of Contents

1. Introduction 1
2. The RAP: overview and current status 1
3. Differences between UNIX and RAP C environments 1
   3.1. Include rap.h 2
   3.2. Stack is fixed size: no large local arrays 2
   3.3. Small memory model: no large global arrays 2
   3.4. Memory spaces 3
   3.5. Word addressable instead of byte addressable 3
   3.6. Opening files for binary or character I/O 3
   3.7. Floating point precision 3
   3.8. Floating point 4
   3.9. Compiler bugs 4
   3.10. DSP does not stop for bad instructions or invalid pointers 4
   3.11. Global variables 4
4. How to use the RAP for parallel processing 4
   4.1. Processor farm 5
   4.2. SIMD 5
   4.3. Pipeline 6
   4.4. MIMD 6
5. Quick start: Running a simple program on the RAP 6
   5.1. Set up PATH 6
   5.2. Set up .rapmcrc 6
   5.3. Copy files from example directory 6
   5.4. Make and Run 7
6. RAP programming environment 7
   6.1. The main function 7
   6.2. RAP libraries 7
      6.2.1. Standard C functions 7
      6.2.2. Differences in memory allocation 7
         6.2.2.1. rap_malloc 8
         6.2.2.2. malloc_usage 8
         6.2.2.3. ckmalloc 8
         6.2.2.4. MALLOC macro 8
      6.2.3. Differences in stdio 8
         6.2.3.1. BINARY vs. ASCII files 8
         6.2.3.2. SIMD mode file reading 9
         6.2.3.3. Changing size and memory type of file buffers 9
         6.2.3.4. Multi-processor file writing 9
      6.2.4. RAP specific routines 9
         6.2.4.1. Matrix and vector functions 9
            6.2.4.1.1. mul_mv_v 9

6.2.4.1.2. muladd_mv_v 10
6.2.4.1.3. mul_vm_v 10
6.2.4.1.4. muladd_vm_v 10
6.2.4.1.5. add_vv_v, sub_vv_v, mul_vv_v 10
6.2.4.1.6. mul_svv_m 10
6.2.4.1.7. muladd_svv_m 10
6.2.4.1.8. mul_vv_s 10
6.2.4.1.9. muladd_sv_v 11
6.2.4.1.10. set_s_v 11
6.2.4.1.11. max_v 11
6.2.4.1.12. copy_v_v 11
6.2.4.1.13. muladd_mb_v 11
6.2.4.1.14. muladd_svb_m 11
6.2.4.1.15. dsigmoid_vv_v 11
6.2.4.2. Table lookup 12
6.2.4.2.1. make_table 12
6.2.4.2.2. make_sigmoid 12
6.2.4.2.3. table_v_v 12
6.2.4.2.4. table_s_s 12
6.2.4.2.5. sigmoid_v_v 12
6.2.4.2.6. sigmoid 12
6.2.4.3. Ring functions 13
6.2.4.3.1. ring_distribute 13
6.2.4.3.2. ring_write and ring_read 13
6.2.4.3.3. ring_sync 13
6.2.4.3.4. ring_sum 13
6.2.4.3.5. Lowest level ring macros 13
6.2.4.3.6. ring_get, ring_put 14
6.2.4.4. Random functions: Numerical Algorithms 14
6.2.4.4.1. rand_func 14
6.2.4.4.2. seed 14
6.2.4.4.3. frandom 15
6.2.4.4.4. random_v 15
6.2.4.5. Panic: error exit 15
6.2.5. Linker command file 15
6.2.5.1. Description 15
6.2.5.2. Moving code into internal RAM or DRAM 15
6.2.5.3. Changing the stack size 16
6.2.5.4. Map file 16
7. RAP debugger: RAPMC 16
7.1. Introduction 16
7.2. Starting up RAPMC 16
7.3. RAPMC command line arguments 17
7.4. RAPMC Scripts 17

7.5. RAPMC Simulated Batch Job Queue 17
7.6. RAPMC commands 17
7.7. RAPMC RAP Manipulation Commands 18
7.7.1. load 18
7.7.2. run 18
7.7.3. Load & Run command 18
7.7.4. examine 18
7.7.5. quit 19
7.7.6. kill 19
7.7.7. reset 19
7.7.8. shelve 19
7.8. RAPMC I/O Commands 19
7.8.1. redirect, > 19
7.8.2. append, >> 20
7.8.3. pipe, | 20
7.8.4. getback, < 20
7.8.5. miss 20
7.8.6. catch 20
7.8.7. echo 20
7.8.8. input 20
7.9. RAPMC Miscellaneous Commands 20
7.9.1. help, ? 20
7.9.2. chdir, cd 20
7.9.3. lcd 21
7.9.4. lpwd 21
7.9.5. node 21
7.9.6. pwd 21
7.9.7. source, ... 21
7.9.8. users 21
7.9.9. background 21
7.9.10. wait 22
8. Assembly language 22
8.1. Register allocation 22
8.2. Pipeline delays 24
8.3. Delayed branch 26
8.4. Conditional load 27
8.5. Repeat block 27
8.6. Addressing modes 27
8.7. Parallel instructions 27
9. UNIX simulated RAP libraries 27
10. An application: multi-layer perception training program (mlp) 27
11. Direct communication between C++ programs running under UNIX and the RAP 29
12. Work in progress 29
13. Appendix A: example program 30

13.1. main.c 30
13.2. makefile 31
13.3. link.cmd: linker command file 33
14. Appendix B: standard C functions supported by the RAP 36
14.1. alphanumeric 36
14.2. math 36
14.3. memory allocation 37
14.4. files 37
14.5. string conversions 38
14.6. memory 38
14.7. strings 38
14.8. variable number of function arguments 38
14.9. time 38
14.10. misc 39
15. Acknowledgments 39
16. References 39