sun®
microsystems

TAAC-1 Application Accelerator: User Guide

Part No: 800-2177-11
Revision A of 15 September 1988

Credits and Trademarks

Sun Workstation® is a registered trademark of Sun Microsystems, Inc.

SunOS™, Sun Microsystems™, SunView™, SunWindows™, and the combination of Sun with a numeric suffix are trademarks of Sun Microsystems, Inc.

TAAC-1™ Application Accelerator is a trademark of Sun Microsystems, Inc.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Contents

Figures

Tables

Preface

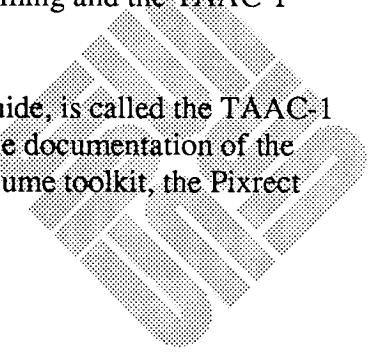
The TAAC-1 User Guide describes the operation and functionality of the TAAC-1 Application Accelerator, a powerful, user-programmable accelerator used for compute-intensive applications such as image processing, high-quality rendering, simulation, and scientific visualization.

Using this guide and the TAAC-1 requires a working knowledge of the C programming language, and the UNIX/SunOs operating system.

This latest version of the User Guide covers TAAC-1 software release 2.2. The 2.2 release includes the volume rendering toolkit, a Pixrect interface, an expanded image processing library, an expanded graphics library, and demos that show the new software in operation. See the Release Notes for 2.2 for more information on the exact nature of the new software.

As part of the 2.2 support effort, the original User Guide has been split into two documents. The User Guide, which formerly contained all TAAC-1 documentation, now contains an introductory chapter, a hardware overview, and chapters on TAAC-1 programming, the C compiler, the assembler, the linker, and the object librarian. Two other chapters describe assembly language programming and the TAAC-1 utilities.

The new document, originally part of User Guide, is called the TAAC-1 Software Reference Manual. It contains all the documentation of the host and TAAC-1 subroutine libraries, the volume toolkit, the Pixrect interface, and the TAAC-1 demos.



1

Introduction

1

Introduction

The TAAC-1 is designed to focus on the computationally intense and display portions of applications involving spatial and geometric data. High performance is achieved by dedication to the application task, leaving the operating system, user interface, data base management, multiprocessing, and networking tasks to the Sun 3 or 4 workstation.

Architectural Goals

The TAAC-1 Application Accelerator is designed to provide maximum performance for the class of problems which involve geometric or spatial data. Design goals have been performance, flexibility, and ease of use.

The performance goal is to provide the processing speed of dedicated, special purpose hardware. The flexibility goal is to provide the user with a highly programmable tool to meet changing needs while continuing to provide a very high level of performance.

Processors offering the performance and flexibility goals stated above have traditionally been very hard to use, with the vendor often providing only an assembler or very limited function library. The ease-of-use goal has been met by a high level language compiler, to make the power of the accelerator available in a familiar programming environment.

Exploit Parallelism

The TAAC-1 contains multiple arithmetic and functional units which efficiently implement the fine-grain parallelism of many algorithms. For instance, operand address calculation, memory access, multiple arithmetic operations, and control flow can be accomplished in a single processor instruction. As such, one TAAC-1 assembly instruction can correspond to multiple microprocessor assembly instructions.

Minimize Latency

The TAAC-1 is a low-latency (as opposed to heavily pipelined) design. Arithmetic operations complete in one instruction cycle. This feature makes the processor efficient for both scalar and vector processing and increases the simplicity of programs compiled for the TAAC-1.

Multiple pipeline stages are difficult to keep full in general purpose software. Program branches require additional instruction cycles to fill or flush the pipeline. These difficulties are eliminated with a low latency design. Conditional and adaptive algorithms which include a lot of testing and branching benefit significantly.

Support Varied Data Structures

The class of applications having spatial and geometric data is not in any way restricted to a single data structure. The TAAC-1 is designed to support several of the most common data organizations: multi-dimensional arrays, linked lists, and structures.

Special hardware in the TAAC-1 processor provides support for stepping through one, two, or three dimensional arrays of 32-bit data without incurring overhead for address calculation. For example, the array can as easily hold the results of a three dimensional seismic survey as a two dimensional matrix of floating point values for an analysis problem. Multiple address registers and multiple ALUs allow the concurrent computation of address and data.

Allow Transparent Memory Access

The TAAC-1 is designed to allow the processor direct access to a large local memory with minimum overhead and software transparency. Local bus access and cycle times are transparent to the processor.

After initiating a memory access in one instruction, the TAAC-1 goes on to process the next instruction unless:

- the operation was a READ and (1) the instruction specifies that the data read is to be used in the current instruction cycle or (2) the read will not complete in the current cycle, and the data read will be used in the next cycle.

- the operation was a WRITE which will not complete in the current cycle, and the current instruction changes the data, address, or address mode registers.

- the operation will not finish in the current instruction cycle, and the next instruction contains another memory access.

This feature allows optimization of the inner loops of programs written in TAAC-1 assembler by interleaving memory access and computational cycles.

Avoid Data Bandwidth Bottlenecks

One of the TAAC-1 design goals is to provide data paths to keep its high performance arithmetic elements supplied with data at or near their rated throughput. Several features of the TAAC-1 architecture reflect this goal:

- Six internal processor buses, providing a variety of paths between the processing units.

- 128 general purpose registers available at all times, rather than a few registers for each procedure level.

- A large local memory so that application data can reside locally and host bus I/O bottlenecks can be avoided.

- Vector data paths from the memory to the processor for fast access to sequentially stored data.

Approach Capability of Dedicated Hardware

At the level of technology of an add-on board, the TAAC-1 strives to come close to the performance of dedicated hardware: to do convolutions as fast as an image processor, geometric transforms as fast as a transformation pipeline, line and polygon drawing as fast as custom integrated circuits, and arithmetic as fast as an array processor. Software for the TAAC-1 can dynamically reconfigure the multiple processors and data paths in the processor, to match the needs of each algorithm. In effect, the processors and data paths can be configured into a "soft" pipeline for the most efficient execution of any particular code. The pipeline can be changed on an instruction-by-instruction basis to provide the best overall throughput.

Efficiently Execute C Programs

Another design goal has been efficient execution of C language programs. This language was chosen because of the ease of developing compilers and because of its increasing pervasiveness. A C compiler allows the user to maintain compatibility with existing and common algorithms without having to learn new ways of thinking and programming.

Provide High Level Language Compiler

There is a cost to meeting these performance goals. The TAAC-1 is a highly complex device. However, the user is shielded from the complexity of the TAAC-1 by the high level language compiler. The compiler is augmented with functions which give the programmer specific command over many of the hardware resources of the TAAC-1 (see C Compiler, built-in functions). In addition, a growing set of TAAC-1 subroutines, including control functions, optimized math functions, graphic routines, and image processing routines will aid system utilization.

2

Hardware Overview

2

Hardware Overview

This chapter provides an overview of TAAC-1 hardware and system capabilities, beginning with discussions of the main architectural features, as shown in the system architecture diagram on the next page. Additional information sections provide explanations of data/image memory organization and Brooktree RAMDAC configuration.

2.1. TAAC-1 Description

The TAAC-1 is a very high performance computational engine with an embedded display system which generates a high quality video signal. The two board set plugs into the Sun workstation and provides additional computing power and a high resolution full-color (32 bits/pixel) frame buffer and display system.

The TAAC-1 has a large local memory which can be used either as storage for large data sets or as a frame buffer to store images for display from the TAAC-1 video output circuitry.

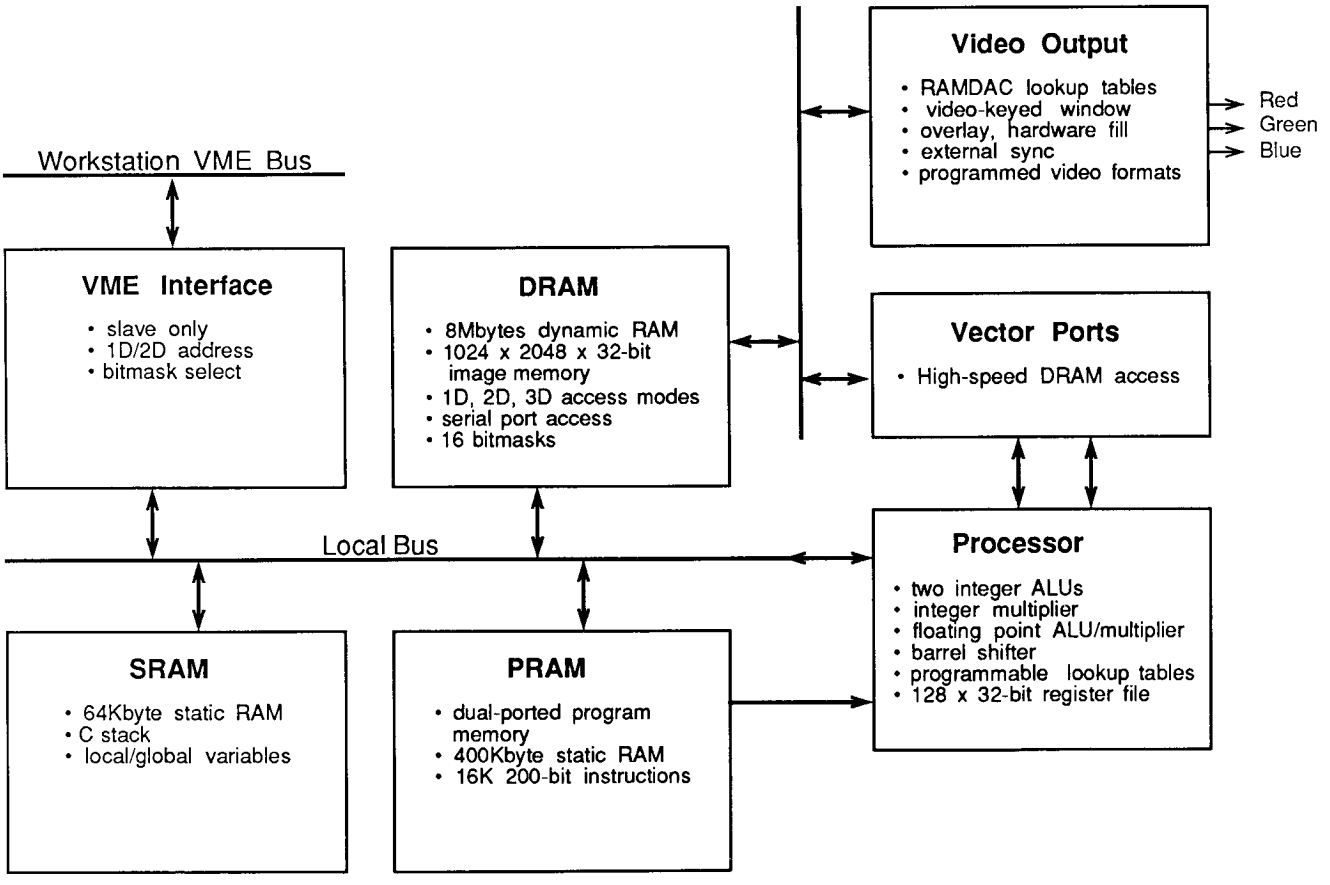
Video signals produced on the TAAC-1 can be displayed in a window on the Sun workstation screen or on a separate monitor. The TAAC-1's video controller can be programmed for a wide range of video formats, ranging from low resolution 512 x 512 interlaced to 1024 x 1024 non-interlaced. Standard video formats currently supported include Sun, Hires (1024 x 1024), RS-343 and RS-170 , with or without genlocked external sync.

2.2. Host VME Interface

The TAAC-1 is connected to the Sun workstation VME bus through a bi-directional host bus interface. All communications to and from the host occur through this interface.

The VME bus interface consists of two parts - a control register in VME address space and the interface to the local bus (LB) on the TAAC-1. The control register, called the slave mode register (SMR), consists of TAAC-1 processor control bits, TAAC-1 data/image memory access mode bits and TAAC-1 memory control bits. The memory control bits select which type of TAAC-1 memory is being mapped into Sun virtual memory space. This is necessary since the TAAC-1 local bus address space (16Gbyte) is larger than the VME address space (4Gbyte), and it is of course not feasible to occupy the entire VME space. Therefore, the TAAC-1 host library of memory access routines allocate approximately 8Mbytes of VME virtual address space. This is sufficient to map to each TAAC-1 memory type, including the entire 1024 x 2048 x 32 bits or 8Mbytes of data/image space.

Figure 2-1 *TAAC-1 System Architecture*



2.3. Local Bus

The major internal bus on the TAAC-1 is called the local bus. It provides the connection between the TAAC-1 processor, various memories on the TAAC-1, and the host interface. The local bus has 32 bits of data and 32 bits of address as well as miscellaneous control lines.

The TAAC-1 memory space is addressed as 2^{32} full 32-bit words and is divided into four memory system types as designated by local bus address bits 30-31. Each memory system type therefore maps to 2^{30} words or 4Gbytes. The four types are:

Bit Field 00	Local Bus
Bit Field 01	Host Workstation
Bit Field 10	reserved
Bit Field 11	reserved

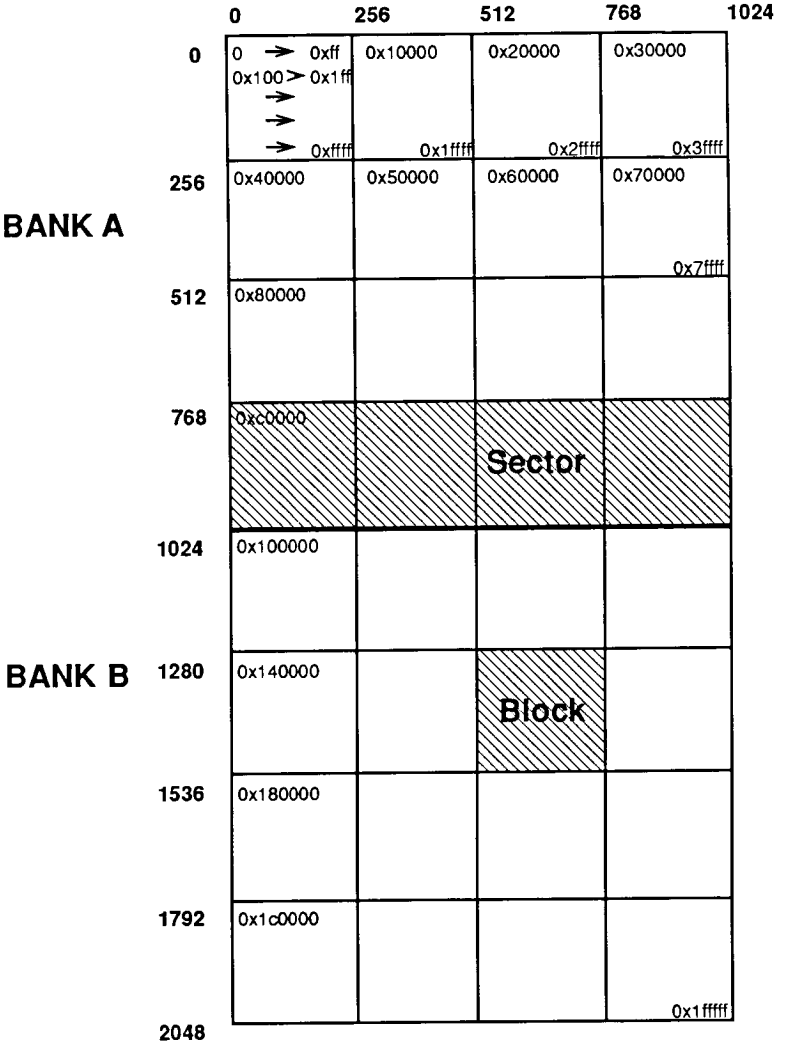
Currently, all data transfers are controlled by the host. Note that the TAAC-1 is 32-bit word addressable and the Sun is byte addressable; the Sun must transfer four bytes when writing or reading a single TAAC-1 word.

2.4. Data/Image Memory

The data/image memory is both dual-purpose and dual-ported. It consists of 8Mbytes of memory, organized as two million 32-bit words. It is fabricated with 256Kbit video dynamic random access memory chips (VRAM). This memory is dynamic in nature and in this manual is referred to as DRAM.

Each memory chip has a 64K x 4 internal organization. In the TAAC-1, the basic memory structure is called a *sector* and consists of 1 Mbyte of memory organized as 256 x 256 x 4 x 32 bits. Each sector is built from 32 memory chips (256 x 256 x 4 x 32 = 64K x 4 x 32). In 2D terms, a sector corresponds to 1024 x 256 x 32-bit pixels, as shown in the next diagram. There are eight 1Mbyte sectors in DRAM memory. Each sector consists of four 256Kbyte *blocks*.

Figure 2-2 *1D/2D Addressing Relationships in Data/Image Memory*



2D addresses are shown in boldface on the outside of the box. The corresponding 1D addresses (in hexadecimal) are shown inside the blocks.

As its name implies, the data/image memory can hold either data or images. When used to hold images, the memory serves as the frame buffer (bit map memory), supplying data to the TAAC-1 video output circuit. For display, the memory is organized as 1024 x 2048 x 32-bit pixels.

The data/image memory may be used more traditionally to hold data. The user can partition the memory into image and data segments as needed by the application being performed. A typical usage is 4Mbytes

for display (1024 x 1024 x 32) and 4Mbytes (1Mword) for data store. Using the Address Count Register (also called the AC register), this memory is addressable in 1D, 2D, or 3D modes.

The data/image memory has two ports:

- a bi-directional *random access port* on the local bus. The TAAC-1 processor and the host interface can read and write the data/image memory through this port via the local bus.

- a bi-directional *serial port* which is connected to the video display bus and the processor vector buses.

The serial port of the data/image memory provides very fast access to data stored at sequential addresses. The serial port supplies data to the display bus at the rate necessary to support high quality video display.

The serial port is also connected to the processor vector buses. Each of the two bi-directional vector buses can transfer 32-bit data words between the processor and sequential addresses in the data/image memory at processor clock rates. The serial buses provide data paths much like those occurring in dedicated vector processor and image processing systems.

2.5. Program Memory

The TAAC-1 processor is controlled by programs stored in the program memory. This 400Kbyte memory is built of fast static RAM and is dual-ported. The 32-bit local bus port is used to read and write program data from the host interface or the TAAC-1 processor. The second port is directly connected to the TAAC-1 processor and is used by the sequencer to read the next program instruction.

The program memory stores up to 16K instructions. Each processor instruction actually uses 32 bytes (256 bits) of local bus address space; only the least significant 200 bits are currently defined and used.

In this manual, instruction memory is referred to as program memory or PRAM.

2.6. Scratchpad/ Stack Memory

The TAAC-1 has a 16Kword scratchpad/stack memory connected to the local bus, built of very fast static RAMs. Data in the scratchpad/ stack is available to the processor in one clock cycle. The scratchpad memory is designed to store frequently used variables and small data sets. Large data sets or display lists are stored in the data/image memory.

The TAAC-1 C compiler uses scratchpad/stack memory for the C stack and for global and static program variables. The C stack grows from high to low SRAM memory, while the global and static variables are allocated space at the beginning of SRAM.

In this manual, this memory is referred to as scratchpad or stack memory, or SRAM.

2.7. Control Register Memory

All TAAC-1 control registers are addressable from the local bus, which translates into a simple and uniform access to all control functions from either the TAAC-1 processor or the host.

Without itemizing each control register, it is worth noting that of the 128Mwords designated as register space, 2Mwords are presently assigned to registers on the processor and video boards. Local bus address bit 19 effectively differentiates between register addresses on the two boards (bit 19 = 0 for processor board registers and 1 for video board registers).

In this manual, register space will be referred to as registers or REG.

2.8. Memory Address Space

All memories can be read and written from the local bus (LB) and are differentiated by address bits 27 - 29. The field assignments for these memory types are:

Bit Field 0xx	Data/Image Memory (DRAM)
Bit Field 10x	Program Memory (PRAM)
Bit Field 110	Scratchpad Memory (SRAM)
Bit Field 111	Control Register Memory (REG)

When communicating from the host to the TAAC-1, only one of these memory types is addressable at any one time, as determined by the page bits in the slave mode register.

TAAC-1 address space usage is summarized in the next table.

Table 2-1 *TAAC-1 Local Bus Address Map Summary*

Memory	*Address Range*	*Capacity*
LOCAL BUS	0x0000 0000 - 0x3fff ffff	1G word
WORKSTAT.	0x4000 0000 - 0x7fff ffff	1G word
RESERVED	0x8000 0000 - 0xbfff ffff	1G word
RESERVED	0xc000 0000 - 0xffff ffff	1G word
DRAM/VRAM	0x0000 0000 - 0x1fff ffff	512M word
Sector 0	0x0000 0000 - 0x0003 ffff	256K word
Sector 1	0x0004 0000 - 0x0007 ffff	256K word
Sector 2	0x0008 0000 - 0x000b ffff	256K word
Sector 3	0x000c 0000 - 0x000f ffff	256K word
Sector 4	0x0010 0000 - 0x0013 ffff	256K word
Sector 5	0x0014 0000 - 0x0017 ffff	256K word
Sector 6	0x0018 0000 - 0x001b ffff	256K word
Sector 7	0x001c 0000 - 0x001f ffff	256K word
reserved	0x0020 0000 - 0x1fff ffff	504M word
PRAM	0x2000 0000 - 0x2001 ffff	256M word
Instructions	0x2000 0000 - 0x2001 ffff	128K word
reserved	0x2002 0000 - 0x2fff ffff	
SRAM	0x3000 0000 - 0x37ff ffff	128M word
Scratchpad	0x3000 0000 - 0x3000 3fff	16K word
reserved	0x3000 4000 - 0x37ff ffff	
REGISTERS	0x3800 0000 - 0x3fff ffff	128M word
Processor Reg	0x3800 0000 - 0x381f ffff	2M word
Video Reg	0x3820 0000 - 0x383f ffff	2M word

NOTE: Each DRAM sector corresponds to a 1024 x 256 image space.

2.9. Processor

The heart of the TAAC-1 Application Accelerator is the processor. It is composed of multiple arithmetic/computational units connected by multiple buses. The processor has ports to the local bus and to the serial port of the data/image memory through the vector port buses. A private bus connects the program memory to the processor. This bus increases efficiency by keeping instructions off the local bus.

The TAAC-1 is a very long instruction word (VLIW) computer, sometimes known as a wide instruction word computer (WIWC), meaning that it executes several operations in the same instruction. Another description frequently given for VLIW devices is that they are machines with horizontal code. In the TAAC-1, each wide instruction word is equivalent to several microprocessor assembly instructions, controlling the operation of one or more computational elements and the movement of one or more pieces of data. Each instruction also handles program flow control.

The next diagram shows that the processor contains two registered integer ALUs, a 32 x 32-bit multiplier/accumulator, a single/double precision floating point multiplier/ALU/accumulator, a 32-bit barrel shifter, an 8K x 32-bit PROM and 8K x 32-bit RAM lookup table and a 16-bit microsequencer.

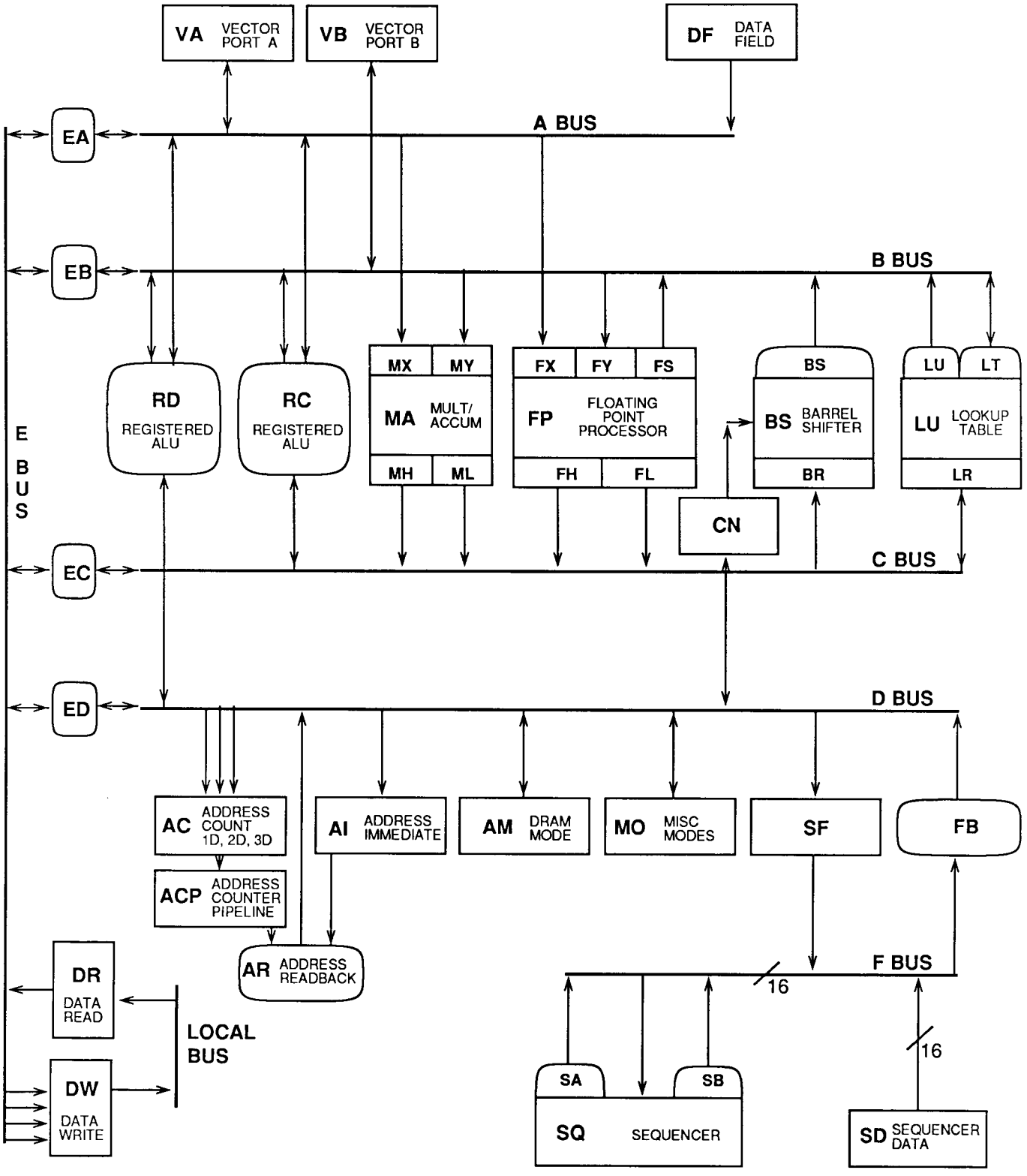
ALUs RC and RD

RC and RD are registered integer ALUs, each containing a 64-register file. The two ALUs operate independently and simultaneously. Because of data paths to and from RC and RD, RC is generally most appropriate for data operations, while RD is better suited for address calculation.

Multiplier/Accumulator (MA)

The Multiplier/Accumulator performs signed and unsigned multiplication on two 32-bit numbers and either stores the result or adds it to (or subtracts it from) a 64-bit accumulation register. The MA rounding modes are controlled by the Miscellaneous Mode Register (MO).

Figure 2-3 *TAAC-1 Processor Architecture*



Floating Point Processor (FP)

The Floating Point Processor performs floating point arithmetic operations: addition, subtraction, multiplication, comparison, and conversion between floats and integers. The 32-bit FX and FY inputs can be loaded simultaneously. The result is stored in a 64-bit output register (FS). The floating processor contains an independent multiplier and ALU capable of simultaneous operatation.

Barrel Shifter (BS)

The barrel shifter performs left or right shifts or rotates. Barrel shifter inputs are written to the BR register. The number of bits to shift can be a constant in the instruction word or a variable loaded to the count register (CN). When a variable is used, there must be at least one cycle between loading the CN register and performing the shift. Because the output of the barrel shifter is not registered, it must be read in the same cycle the shift is performed.

Miscellaneous Lookup RAM (LT)

The Miscellaneous Lookup RAM is an 8K by 32-bit lookup table, which can be loaded by the processor and indexed with a 13-bit unsigned integer from bits 0-12 of the LR register. LT has no initial values. The output is available in the LT register.

To load the lookup table:

- write the lookup table address to the LR register
- on a subsequent cycle, write the data to the LT register

To read the lookup table:

- write the lookup table address to the LR register
- on a subsequent cycle, read the data from the LT register

Lookup PROM (LU)

The Lookup PROM has two parts: an 8K by 8-bit Exponent Lookup PROM and an 8K by 23-bit Mantissa Lookup PROM. Both PROMs are addressed with values in the LR register. The output is available in the LU register. These lookup tables provide both floating-point and integer lookups, depending on the setting of MO register bits 21-23. The TAAC-1 library contains functions to access the Lookup PROM.

For additional information on the internal structures of the processing elements, consult the assembly language chapter.

2.10. Buses and Data Paths

Central to the TAAC-1 performance is the interconnection of processing elements as illustrated in the previous diagram. The ability to independently select the sources and destinations for the six data buses in each instruction gives the TAAC-1 much of its flexibility and power. While no bus has a particular hard-wired function, each data path generally has a common set of uses and can be best understood by looking at how data moves through the processor.

The E Bus

The E Bus serves primarily as a data path for memory I/O and access to the Local Bus (LB). With the Data Read (DR) and Data Write (DW) registers serving as an interface between the Local Bus and the processor, the E Bus provides a data path to all memory types for all buses and the processing elements on them. The transceiver buffers connecting the E Bus with the A, B, C and D buses permit data flow through the processor in the same instruction without delay.

The A and B Bus

A and B buses primarily carry operand sources, providing inputs to both registered integer ALUs, the multiplier/accumulator, the floating point multiplier and floating point ALU. The E bus often serves as an A or B bus source for memory data reads.

The register file for each integer ALU has one read port connected to the A and B buses, permitting two register operands to be specified to any of the arithmetic units in the same instruction. There are also corresponding register file data paths for declaring any register the destination for A and B bus sources.

The A and B buses can also be the source or destination of the two vector ports (VA, VB), for direct access to sequential data/image memory on virtually every processor clock cycle.

In addition, the barrel shifter and lookup tables can source the B bus to form effective pipeline loops with integer data. For example, a single instruction loop can be constructed to move integer data from an ALU into the lookup table input, while moving the current lookup table output into the ALU's input. Similarly, the barrel shifter can be piped with an integer ALU or multiplier/accumulator to perform ALU mask → shift → merge, or ALU mask → shift → multiply loop operations in a single instruction.

The C Bus

The C Bus serves as direct output bus for all the arithmetic units (except ALU RD) and as a path to the E Bus and the TAAC-1 memories. The C Bus can also be used to source the register file in ALU RC, so the multiplier/accumulator and floating point processor can write results to a register file in one instruction. Therefore, RC tends to be used for more data calculations than RD. The C Bus logically supplies the barrel shifter and lookup table inputs.

The D Bus

The D Bus is used primarily for memory address and control, since this bus sources the two memory address registers (AC, AI) and the control registers (AM, MO). The output on integer ALU RD is connected directly to the D Bus, so RD is more often used for address and control calculations than RC.

The ALU pair (RC, RD) and dual output bus (C, D) arrangement allows for the efficient execution of high-level language operations such as stack-based variable access. In this situation, RD and the D Bus handle getting the data while RC, the C bus and the arithmetic units operate on the data.

The shift count for the barrel shifter can come from the count register on the D Bus or from the instruction data field. This allows calculated shift counts to be computed in RD concurrently with other arithmetic operations. Also interfaced to the D Bus via the SF register for writes and the FB transceiver for reads is the sequencer data bus - the F Bus.

The F Bus

The F Bus connects the sequencer to the D Bus for calculated input values and to the instruction data field for compiler- or linker-generated input values. The inputs from the compiler or linker are usually branch or procedure instruction addresses, or counts for the sequencer's two internal loop counters. The sequencer internal register and stack values can be written into memory by moving them onto the F Bus, D Bus and E Bus.

2.11. Address Registers

Two memory address registers exist as destinations from the D Bus - the Address Immediate Register (AI) and the Address Count Register (AC).

Address Immediate Register (AI)

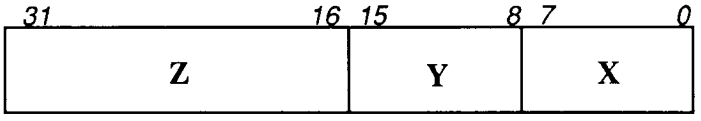
The 32-bit AI register is used as a normal address register (i.e., load the desired 32-bit linear address) and as such is used for addressing all memory types - DRAM, SRAM, PRAM, and registers on the local bus.

Address Count Register (AC)

The AC register provides special 2D and 3D addressing mode support in the TAAC-1, which further reduces the requirements for an ALU to calculate an equivalent linear address from 2D or 3D indices. In 2D and 3D modes, only DRAM memory access is possible.

For loading, the AC register is considered to have three fields - the least significant byte or **X** field (bits 0-7); byte 1 (bits 8-15), the **Y** field; and bytes 2-3 (bits 16-31), the **Z** field. *These field names are given in boldfaced capital letters to distinguish them from x,y or x, y, z address indices in 2D or 3D arrays.* The fields, **X Y Z**, can be independently loaded.

Figure 2-4 *Format of AC Register Fields*



1D Addressing

When used in the 1D mode, the 32-bit AC register performs exactly as the AI register – an address is loaded into all three fields simultaneously.

2D Addressing

When using the AC register to handle a 2D image or array address, the x index is loaded in the least significant short word (fields **X** and **Y**) and the y index value is loaded in the **Z** field, the most significant short word. Only 11 bits of these 16-bit, 2D indices are relevant. Once the AC register has been loaded, the fields can be independently incremented or decremented without involving the ALUs.

3D Addressing

In 3D mode operation, the three array indices (x, y, z) are loaded into the AC fields **X**, **Y**, and **Z**, respectively. The three fields can be

independently incremented or decremented without an arithmetic carry across fields.

There are actually two 3D modes:

- The normal 3D "dice" mode reorganizes the AC address for accessing memory by alternating **X**, **Y**, and **Z** bits, providing a cubical or voxel memory organization.
- In 3D "slice" mode, the **X**, **Y**, **Z** fields of the AC register are loaded in the same manner; but the address is passed through for memory access without bit reordering. This mode is most useful in indexing 256 x 256 images such as CT scan data.

DRAM Mode Register (AM)

The DRAM Mode Register (AM) controls the modes of data/image memory access through the address registers (AI and AC) and the vector ports. The AM register controls these fields:

Word/Channel Mask Mode Enable

This field affects the operation of DRAM writes only. Its function is different for random writes and vector port (serial) writes.

For random writes, word mask mode allows the processor, in a single instruction, to write the same 32-bit value to as many as four sequential DRAM addresses, starting on a four-word boundary. The actual words written are determined by the four-bit *word mode* mask. The least significant bit of the mask enables writes to the least significant word of the four word DRAM cluster (or the DRAM word addressed with the two LSBs = 0). Correspondingly, the most significant bit of the mask enables random writes to the DRAM word addressed with the two LSBs = 1.

For vector port writes, the four-bit mask does not act as a write mask spatially, but in depth as a *channel* or byte mask for each 32-bit write operation. The least significant bit of the mask enables serial writes to the least significant byte or channel 0 (red) and the most significant mask bit correspondingly controls writes to channel 3 (alpha).

Thus for random writes, this control bit enables the word mask mode and for serial/vector port writes it enables a channel mask mode. The vector ports are described in detail in a later section.

Word/Channel Mode Mask

This four-bit AM value selects the DRAM words to be written in a single operation when doing random writes and the DRAM channels to be written when doing serial, vector port writes. This mask operation is enabled by the word/channel mask mode enable bit.

Bitplane Mask ID

Selects a bitplane mask (0-15), to be set and/or used in subsequent writes to DRAM memory only. 16 32-bit masks are available. Note that selecting the bitplane mask ID via a host subroutine does not affect the AM register and therefore would have no effect on processor writes to DRAM. You must set the mask ID in the AM register.

Bitplane Mask Write Enable

Enables use of the current bitplane mask in random Local Bus writes to DRAM. When bitplane mask writes are disabled, the current mask is ignored. The bitplane mask is not used for vector port writes (see Word/Channel Mode Mask).

1D/2D/3D Select

Selects the mode in which the AC register **X**, **Y**, and **Z** counters are loaded and incremented/decremented, as well as how the effective address is calculated from these counters. See the Address Count Register (AC) section.

Bounds Checking (Z-Buffer) Enable

Enables conditional writes to DRAM memory. If bounds checking is enabled, the processor permits writes to DRAM only if the value in the upper 16 bits of the Data Write Register (DW) is less than or equal to the upper 16 bits of the Data Read Register (DR). The comparison between DR and DW is an unsigned 16-bit compare. To use this feature for z-bounds checking:

- Initialize the DRAM area to be used to some z value (0xffff0000 puts all pixels in the background).
- Store z values in the upper 16 bits of each pixel; pixel colors in the lower 16 bits.
- For each pixel, read the pixel at that address, then write the pixel. If the z value of the new pixel (DW) is less than or equal to the z

value of the current pixel (DR), then the write will take place. Otherwise the processor will inhibit the write.

DRAM Access Mode

Selects random access, shift register load (DRAM to shift register), shift register store (shift register to DRAM), or serial write enable (shift register to DRAM). The normal mode is random access; the other modes are for use of the vector ports. See the Vector Ports section for more information.

Read Disable

This bit is used for diagnostics and by the register dump subroutine. It disables reads into the DR and loads the AI address into the address readback (AR) register. It should normally be zero.

Miscellaneous Mode Register (MO)

The Miscellaneous Mode (MO) register controls modes for the vector ports, integer ALUs RC and RD, the floating point processor, the multiplier/accumulator, and the lookup table PROM. MO contains these fields:

VA and VB Strides

The stride is the amount used to increment the address when reading or writing the DRAM shift register with the vector ports. Each read or write automatically advances the address pointer, by one word (the default), two, three, or four words. A stride of three, for example, reads or writes every third word.

RC and RD Configurations

These fields set the mode of each ALU. Normal mode is word mode. In byte mode, carry is inhibited between bytes, and the user selects which byte the status will come from. In halfword mode, carry is inhibited between halfwords, and the user selects which halfword the status will come from.

Floating Point Processor Modes

The FP clock mode, fast mode, and configuration fields should not be changed. The default values currently select FP fast mode and single precision. FP round mode selects the direction of rounding for floating-point numbers. The default is round-to-zero.

Multiplier/ Accumulator Round Modes

The multiplier round mode selects the rounding option for integer multiplies. No rounding is the default.

Lookup Table Function

Selects the function returned by the lookup PROM.

For further information regarding AM and MO register fields, consult the assembly language chapter. Many of these functions are controlled by library subroutines.

2.12. Vector Ports

The vector ports are high-speed buses connected between the processor and the serial port of data/image memory. In most graphics hardware, the serial port is accessed only by the display controller, for image refresh. In the TAAC-1, the processor directly accesses the serial port for reads and writes. This is achieved by dividing the data/image memory into two distinct banks, A and B, and dedicating a high-speed serial port to each bank, so display controller access and processor access can occur concurrently on different buses. The processor can access either bank using a vector port or the random access port connected to the local bus. The processor accesses the serial ports of Banks A and B through Vector Port A (VA) and Vector Port B (VB), respectively. Vector Port A is a source or destination for processor bus A. Vector Port B is similarly tied to processor bus B.

The vector ports give the TAAC-1 processor access to a new 32-bit word from data/image memory on every processor cycle within the limits of the 1024-word serial shift register internal to the DRAM array. Therefore, in reading or writing to data/image memory through the vector ports, most applications will be forced to break the single cycle access on every 1024th access, to load or store the serial shift register to or from the actual DRAM array. This process requires a full memory cycle or three processor cycles to set up another fully random address. The shift register can only be loaded on 1024-word boundaries (sequential addresses 0-1023, 1024-2047, 2048-3091, etc). This 1024-word memory page is sequential in linear address space and corresponds to a 256 pixel by 4-line spatial area in 2D image space.

The division between memory banks A and B is between sectors 3 and 4, and in 2D image space corresponds to the vertical boundary at

y=1024, as shown in Figure 2-2. Bank A is spatially 1024 x 1024 x 32 bits starting at linear address 0x0 or 2D address (0,0) and Bank B is the same size beginning with sector 4 at linear address 0x100000 or 2D address (0,1024). This division allows effective 1024 x 1024 double buffer operation with the processor alternately creating the new image in banks A and B (using vector ports A or B) while the display controller concurrently refreshes the image from the opposite bank.

2.13. Detailed Information: Data/Image Memory

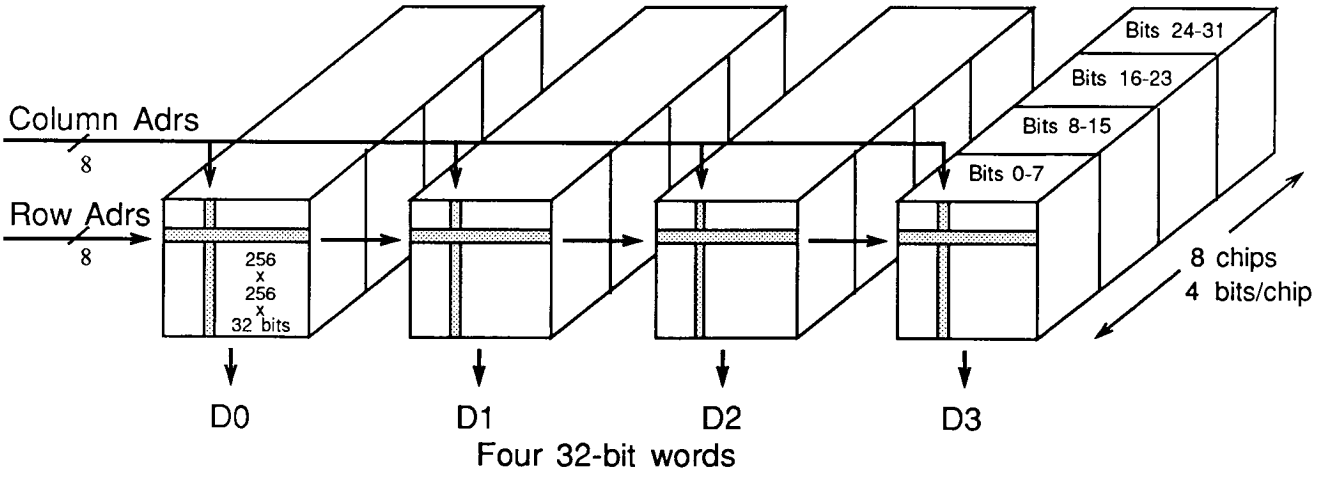
This section describes how data/image memory is addressed in 1D, 2D, and 3D modes.

The TAAC-1 is configured with 8Mbytes or 2Mwords (32-bit) of data/image memory. To most efficiently utilize this memory, you need to know how the memory is organized and how it is addressed in 1D, 2D, and 3D modes.

Linear (1D) Addressing

The data/image memory is built with 256K video random access memory chips. The memory is organized as four parallel 256 x 256 memory matrices, so that 32 memory chips addressed in parallel can simultaneously access four 32-bit words from an 8-bit row and 8-bit column address. In the TAAC-1, this basic memory structure is known as a *sector* and consists of 4 x 256 x 256 x 32 bits of memory. A 1Mbyte (256K word) sector can be addressed with only 18 bits.

Figure 2-5 *Sector Organization*



For more memory, sectors are simply replicated, requiring additional address bits for sector selection. The TAAC-1 has eight sectors for a total of 8Mbytes (2Mwords, or 2^{21}) of contiguous data/image memory. The next illustration shows the related row, column, and sector fields in the 21-bit DRAM address.

Figure 2-6 *Sector Address Map*

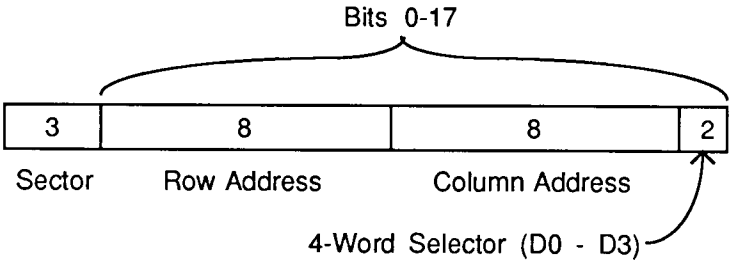
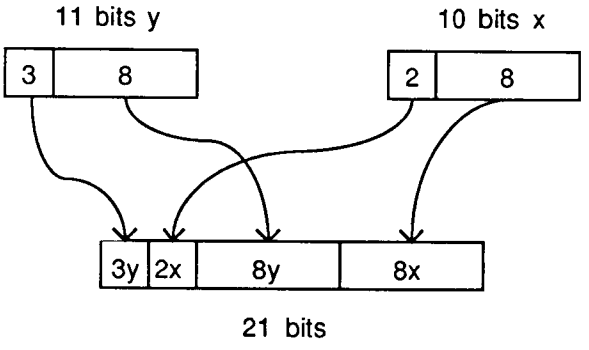


Image (2D) Addressing

The 8Mbytes of data/image memory, which is uniquely addressed by a 21-bit address, can also be addressed as two-dimensional x/y image space. In 2D space, the 8Mbytes of memory is organized as a 1024 x 2048 x 32-bit image array. The next diagram shows how the bit fields of the two spatial coordinates (x,y) relate to the 21-bit linear address, with 10 bits of spatial x addressing and 11 bits of spatial y addressing. This bit shuffling is handled in hardware for AC register addressing.

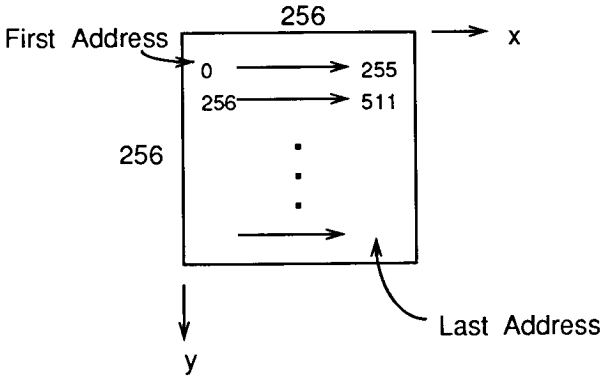
Figure 2-7 *2D X/Y Address to Linear Address Mapping*



Analyzing this bit re-structuring, you can see that the 16 least significant bits (8 LSBs of x and 8 LSBs of y) or 64K words of linear address space form a 256 x 256 replicated pixel structure called a *block*. Sequential linear addresses in this 16-bit address space map left to right (8 bits of x), top to bottom (8 bits of y) in 2D raster image

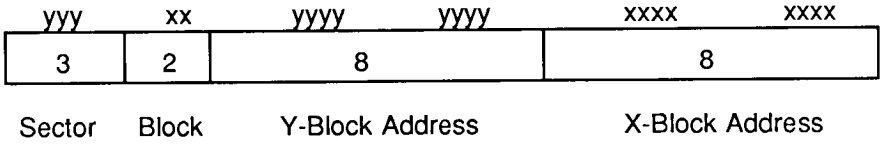
space. The next diagram shows the relationship between the first 16 bits (or block) of 1D data space and 2D image space.

Figure 2-8 *Linear Addresses in a 2D Image Block*



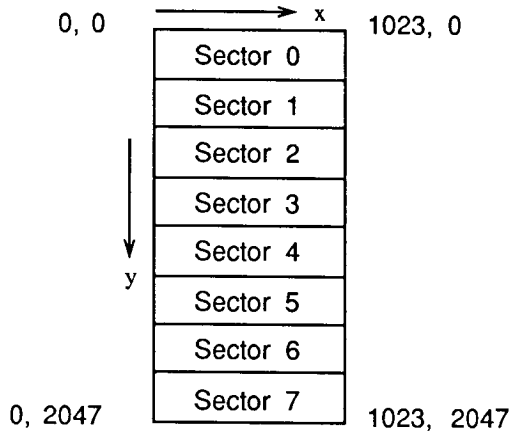
Comparing the address map of Figures 2-4 and 2-5, you see that the three most significant bits of the 11-bit spatial y address are mapped directly to the three most significant bits of the sector field and the two most significant bits of the spatial x field can be considered the block number within a sector. The next figure also illustrates these relationships.

Figure 2-9 *1D/2D Addressing Relationships*



With two bits of image x address forming the block field, four blocks of 256 x 256 x 32-bit pixels are needed to define one sector. Four sequential blocks in linear 1D address space are organized as four left-to-right blocks in one 1024 x 256 sector in 2D image space.

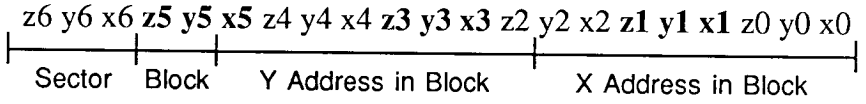
Figure 2-10 *Sectors in 2D Image Space*



Volumetric (3D) Addressing

The TAAC-1 efficiently addresses data/image memory in two different 3D modes - "dice" and "slice." In "dice" mode, the 21-bit DRAM address is derived from three seven-bit fields corresponding to the spatial indices, x, y and z. The three index fields are rearranged by the AC addressing hardware , as shown in the next diagram, into seven three-bit fields of the same significance, each with alternating bits of x, y and z.

Figure 2-11 *3D Index to Linear Address Mapping - "Dice" Mode*



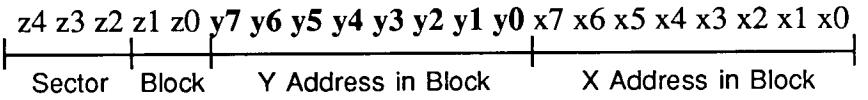
This format change produces a mapping of 3D data triplets in sequential addresses in data memory. This mapping has the advantage of compacting 3D arrays of data to contiguous linear space. Because of the corresponding relationships of 2D space to linear address space, 3D arrays are not scattered over image memory, but instead are compacted into groups of image blocks (256 x 256 pixels). The 3D image space table summarizes the relationships between 3D data space and 1D/2D memory space.

Table 2-2 *3D Image Space Relationships*

3D Cube Size	Total Words	2D Interpretation			Cubes Available
		Blocks(256x256)	X	Y	
8	512	1/128	256	2	4K
16	4K	1/16	256	16	512
32	32K	1/2	256	128	64
64	256K	4 (1 Sector)	1024	256	8
128	2Mb	32 (8 Sectors)	1024	2048	1

In 3D "slice" mode addressing, the 8Mbytes of data/image memory is mapped as 32 slices (addressed by a 5-bit z index) of 256 x 256 image data (addressed by 8-bit x and y indices). This mode is particularly useful in storing multiple 256 x 256 x 32-bit images such as CT or NMR data. The 3D indices are directly mapped into linear address space without any bit shuffling, as shown in the next diagram.

Figure 2-12 *3D Index to Linear Address Mapping - "Slice" Mode.*



2.14. Video Output

The video output portion of the TAAC-1 receives digital pixel data from the data/image memory via the serial port and the display bus. The video data is logically separated into four 8-bit channels:

- channel 0 (also known as the red channel) is for least significant image data bits 0-7;
- channel 1 (the green channel), for bits 8-15;
- channel 2 (blue), bits 16-23; and
- channel 3 (alpha), bits 24-31.

Each 8-bit channel of video passes through its own video lookup table and digital to analog converter (DAC) to provide a palette of 2^{24} or 16.7 million possible colors. The video output circuitry handles all video timing and can be locked (genlocked) to an externally applied composite sync signal such as NTSC video or Sun video. Most of the timing parameters are programmable, offering the flexibility needed for special purpose applications such as driving flat-panel displays at various line rates. The programming of the sync hardware is complex and the customer should consult with the factory for unusual video display formats. Video pan control is available in four-pixel increments and video scroll control in single-line resolution.

Read Masks

Another video pipeline feature provided by the TAAC-1 is full bit depth video channel read masks. The entire 32-bit pixel depth is logically ANDed with the read mask; the resultant value is used to index the colormap.

Overlay/Blink

The alpha channel (most significant image bits 24-31) can be used as a full 8-bit overlay channel providing 256 possible overlay colors from a palette of 16.7 million. In addition to a master overlay enable control bit, individual alpha channel bitplanes can be enabled as active overlay planes with an eight-bit overlay mask register. This provides a simple and effective means for creating and activating multiple independent overlay images from the single eight-bit alpha channel (e.g., three independent overlays using bits 24-27, bits 28-29 and bits 30-31 or double-buffered 4-bit overlays).

A hardware blink is also available on the alpha channel only.

Channel Select

The flexibility of the TAAC-1 video section permits multiple display modes. By enabling and properly loading the full 256 x 24-bit colormaps available to each 8-bit image channel, it is possible to utilize image memory to display four independent 8-bit channels in pseudo color, an 8-bit display with 8-bit overlay, or a 24-bit true-color image with 8-bit overlay.

Scan Line Fill

Use of the TAAC-1's scan line fill mode to create images offers performance advantages by substantially reducing the number of memory writes required. In normal image creation, it is necessary to define every pixel even though many of the pixel runs on a given scan line may be the same value. With scan line fill it is only necessary to

define the pixels at the edges of constant-shaded areas. The TAAC-1 effectively does a bit-by-bit parity fill between pixels on the same scan line.

Because of the memory organization in the TAAC-1, this parity fill is oriented to a four-pixel structure rather than to a single-pixel parity fill. Therefore, to turn on a bitplane from pixel 100 to pixel 400, pixels 100-103 and 397-400 must be set. If only pixels 100 and 400 are set, the parity fill will only turn on every fourth pixel in that bitplane between pixels 100 and 400.

To further improve the usefulness of this feature, control registers provide independent fill enable for the RGB channel set and the alpha channel.

Video Keying into Sun Windows

Unlike dual-monitor configurations where separate monitors are dedicated to Sun and TAAC-1 video outputs, the single-monitor system lets you see video from the workstation and TAAC-1 video from the same display device.

The TAAC-1 video section is designed to provide TAAC-1 video within the extensive domain of Sun windowing. The TAAC-1 features a video keying circuit which allows it to insert TAAC-1 video into Sun video. This video mix occurs on the TAAC-1 and is keyed to a user-selectable color in Sun video. TAAC-1 video is inserted or keyed into Sun video based on a color in Sun video, to form a composite Sun/TAAC-1 frame. Video keying provides a "porthole" into TAAC-1 image memory in the context of the Sun windowing system by creating a Sun window with a unique, user-selectable color assigned to a canvas area inside the window.

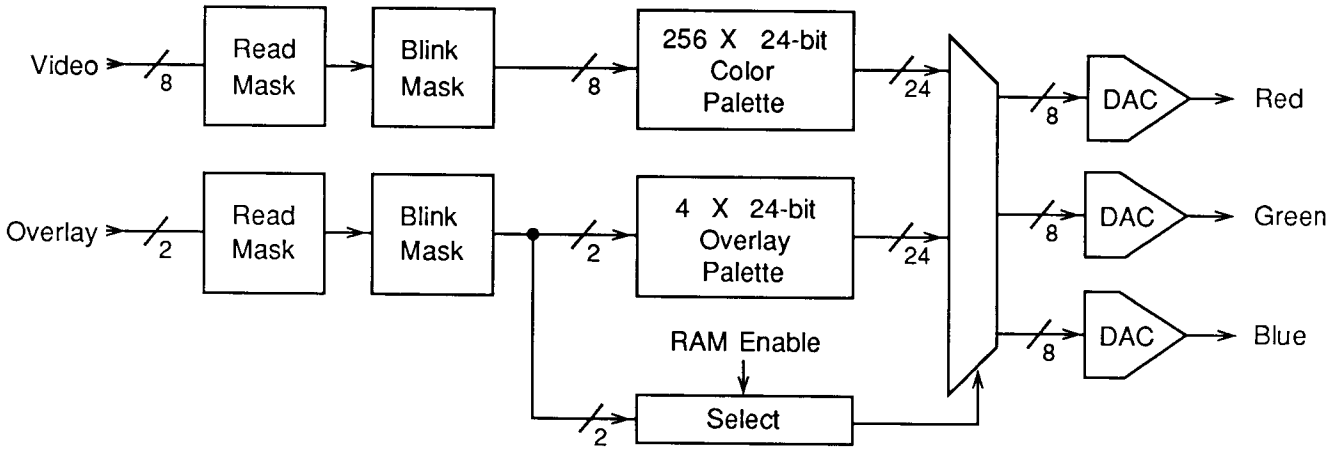
See the TAAC-1 utility `tatool` to learn how to up the TAAC-1 and Sun keying parameters. For more information on the TAAC-1 video, see the next section.

2.15. Detailed Information: Brooktree RAMDACs

This section explains the function, use, and internal configuration of the Brooktree RAMDACs.

The Brooktree RAMDAC has a 256 x 24 color lookup table with triple 8-bit digital to analog converters (DACs). RAMDAC features include bitplane masking and blinking, color overlay capability (four 24-bit colors) and a color palette or lookup table RAM.

Figure 2-13 *Brooktree Functional Block Diagram*



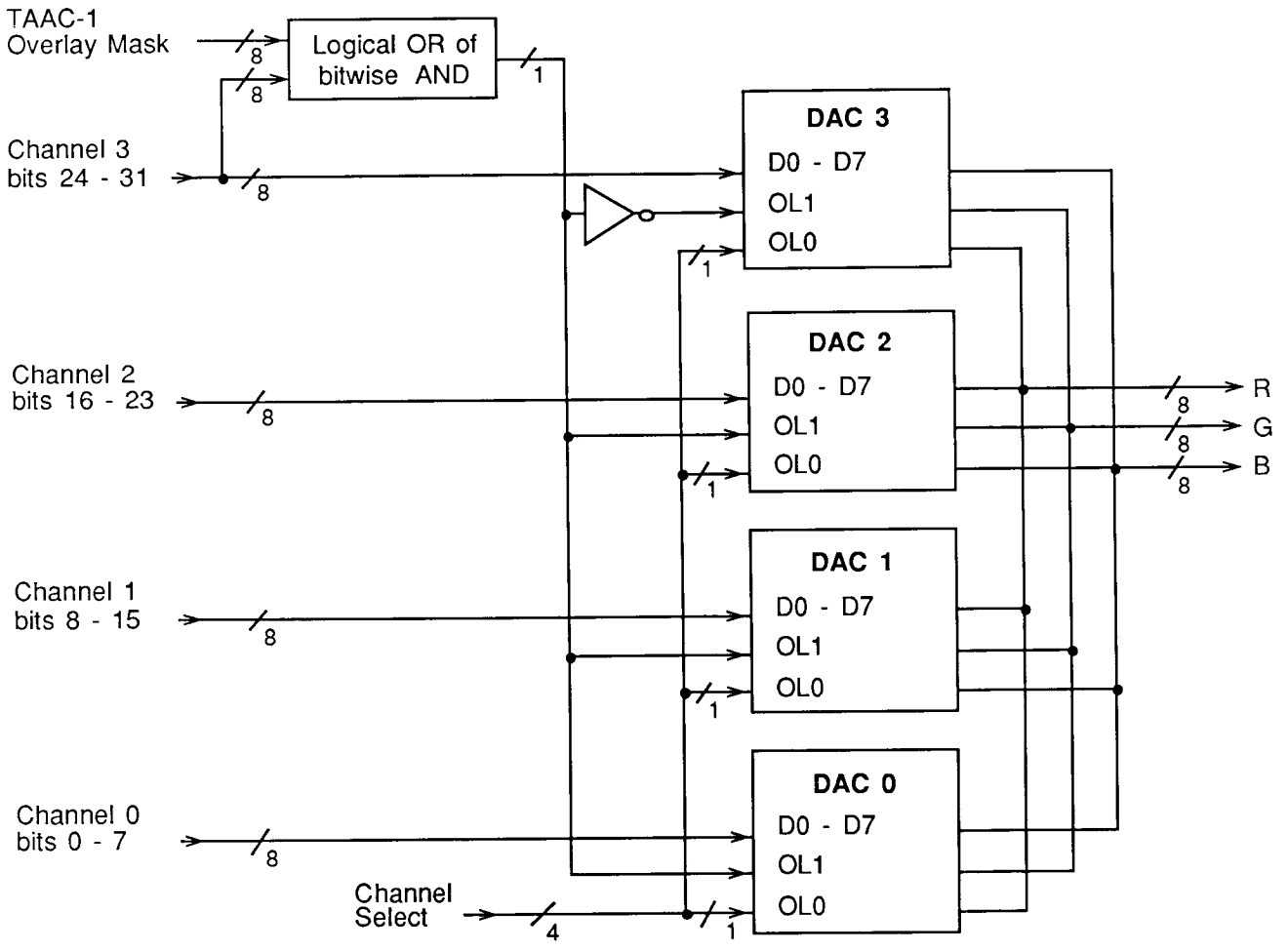
The 24-bit output to the video DACs is either the color palette RAM output (indexed by 8-bit incoming video) or one of four overlay colors as controlled by the two RAMDAC overlay bits and the RAM enable bit of the internal RAMDAC command register.

NOTE: The RAMDAC overlay bits and four overlay colors should *not* be confused with the implementation of the TAAC-1 overlay function. The RAMDAC overlay capabilities are used to effectively enable or disable TAAC-1 video channels. More on this subject later.

The TAAC-1 implementation uses four Brooktree RAMDACs, as shown in the next diagram. One RAMDAC is associated with each of the four 8-bit video channels. The RGB outputs of all four RAMDACs are physically tied together to form the RGB monitor signals. This physical arrangement and the two overlay bits on each RAMDAC are

used to implement independent channel select and a full 8-bit overlay channel.

Figure 2-14 *RAMDAC Configuration*



With the four overlay colors in all RAMDACs (0-3) defined to be black, the individual overlay control bits (OL0) as defined by the four channel select or enable bits can be used to switch each RAMDAC's output from the color palette RAM to black. Since the DAC adds no current drive to the hard-wired RGB output network when outputting black, the RAMDAC or video channel is effectively disabled.

The full eight-bit TAAC-1 overlay function is implemented with the channel 3 RAMDAC (alpha channel) and the use of the other overlay channel control bit (OL1). When the bitwise AND of the channel 3 video bits (24-31) with the overlay mask register (both are 8 bits wide) is true, RAMDACs 0, 1, and 2 are switched to a black overlay color or disabled. At the same time, the inverted signal enables the

channel 3 RAMDAC, thus outputting the overlay channel through its own color palette.

For the channel select and alpha channel overlay to work, the RAMDAC overlay colors must be loaded to black and the command register must be in the state:

```
TA_USE_LOOKUP | TA_MASK0_OFF | TA_MASK1_OFF
```

The overlay select truth tables appears in the next table. For more information, refer to the reference pages for the functions `ta_set_channel()` and `ta_set_channel_select()`, in the host library chapter.

Table 2-3 *RAMDAC Palette Multiplexing*

Overlay OL0, Bit 0	*Overlay OL1, Bit 1*	*RAM Enable* `TA_USE_LOOKUP`	*RAM Enable* `TA_USE_OVERLAY`
0	0	Color Palette	Overlay Color 0
0	1	Overlay Color 1	Overlay Color 1
1	0	Overlay Color 2	Overlay Color 2
1	1	Overlay Color 3	Overlay Color 3

CAUTION: Since the outputs of the RAMDACs are tied together, it is important that the RGB colormaps of each RAMDAC be properly loaded before the channel is enabled, to avoid over-driving and possibly damaging the video monitor.

For most users, it is not necessary to thoroughly understand the RAMDAC operation. Utilization of the provided library functions for controlling these video functions is suggested. Examining a few simple operating cases will help clarify this operation. For an example program, refer to the TAAC-1 programming chapter.

Example 1: Single 8-Bit Image (Pseudo or Grey) in Channel 0

1. Load red, green, and blue color palettes in RAMDAC 0 as desired using `ta_set_colormap()`.

2. Load black (0) in RGB palettes of RAMDACs 1, 2, and 3 using `ta_set_colormap()`, or do not enable them when you call `ta_set_channel_select()`.

3. Enable RAMDAC 0 with `ta_set_channel_select()`.

Example 2: Full 24-bit True-Color Image

1. Load red color palette (ramp) in RAMDAC 0 and black (0) in green and blue palettes of RAMDAC 0 using `ta_set_colormap()`.

2. Load green color palette (ramp) in RAMDAC 1 and black (0) in red and blue palettes of RAMDAC 1.

3. Load blue color palette (ramp) in RAMDAC 2 and black (0) in red and green palettes of RAMDAC 2.

4. Load black (0) in RAMDAC 3 or do not enable via `ta_set_channel_select()`.

5. Enable RAMDACs 0, 1, and 2 using `ta_set_channel_select()`.

Example 3: True-Color with 8-bit Overlay

1. Load red color palette (ramp) in RAMDAC 0 and black (0) in green and blue palettes of RAMDAC 0 using `ta_set_colormap()`.

2. Load green color palette (ramp) in RAMDAC 1 and black (0) in red and blue palettes of RAMDAC 1.

3. Load blue color palette (ramp) in RAMDAC 2 and black (0) in red and green palettes of RAMDAC 2.

4. Load RGB color palettes of RAMDAC 3 with 8-bit overlay colormap.

5. Enable selected overlay mask bits (probably all) and the overlay mode using `ta_set_overlay_mask()` and `ta_set_overlay_mode()`.

6. Enable all four RAMDACs with `ta_set_channel_select()`.

The eight-bit overlay channel can easily be configured as two four-bit double-buffered overlays and switched by simply reloading the overlay mask.

3

TAAC-1 Programming

3

TAAC-1 Programming

This chapter describes the general approach for developing applications on the TAAC-1 application accelerator, focusing on the type of program that has components running both on the host Sun workstation and on the TAAC-1. This type of application involves communication between, and synchronization of, separate host and TAAC-1 processes.

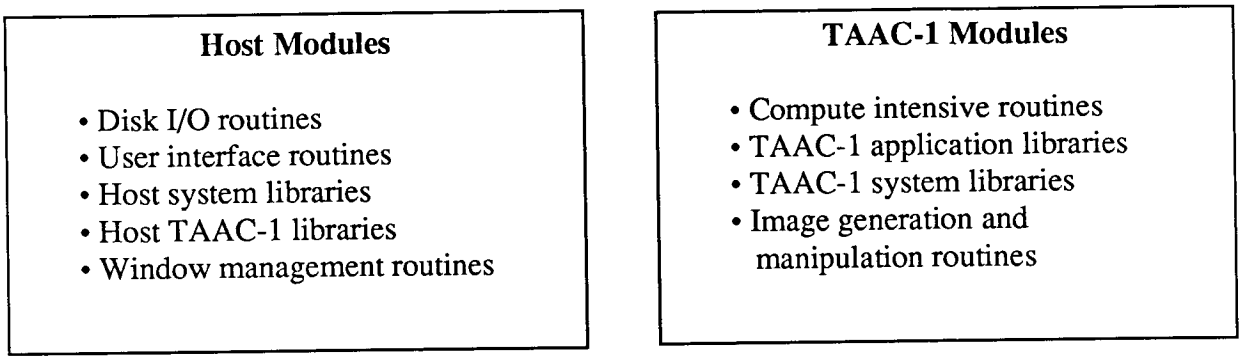
This chapter consists of two overall parts, the development overview and the tutorial. The development overview discusses the division of tasks between the two processors, as well as communication, synchronization, TAAC-1 memory allocation, window management, and building a TAAC-1 program. The tutorial part of the chapter contains an instructional programming sequence and TAAC-1 example programs.

3.1. Dividing Tasks Between the Sun and TAAC-1

The first step in developing a TAAC-1 application is to assign portions of the program to run on the host and portions to run on the TAAC-1. Assignments are based on the types of tasks performed on the host and the TAAC-1, as shown in the next diagram.

TAAC-1 software does not currently support standard I/O routines. Furthermore, the TAAC-1 is a slave processor and cannot write to the host. Therefore, all reading and writing of data to and from the TAAC-1 is done by host processor routines.

Figure 3-1 *Division of Tasks Between Host and TAAC-1*



Synchronizing Programs

Host and TAAC-1 programs are asynchronous. Synchronizing the two programs is typically done with a simple handshaking protocol like the one in the next illustration, which shows one possible organization for the handshaking, based on clearing and setting a flag.

Notice in the illustration that the handshake flag (`ioflag` in this example) is in TAAC-1 memory. The host program loads the TAAC-1 program and writes data to TAAC-1 memory. Then it sets `ioflag`, telling the TAAC-1 to proceed. (`TC_ioflag` is the address of `ioflag` in TAAC-1 memory.) The TAAC-1 program waits for the host program to set `ioflag`, and clears the flag when it has finished its tasks.

To further illustrate this synchronization technique, the program organization described above is duplicated in Figure 3-3, using program fragments with the appropriate TAAC-1 library routines.

Figure 3-2 *Handshaking Protocol Between Host and TAAC-1*

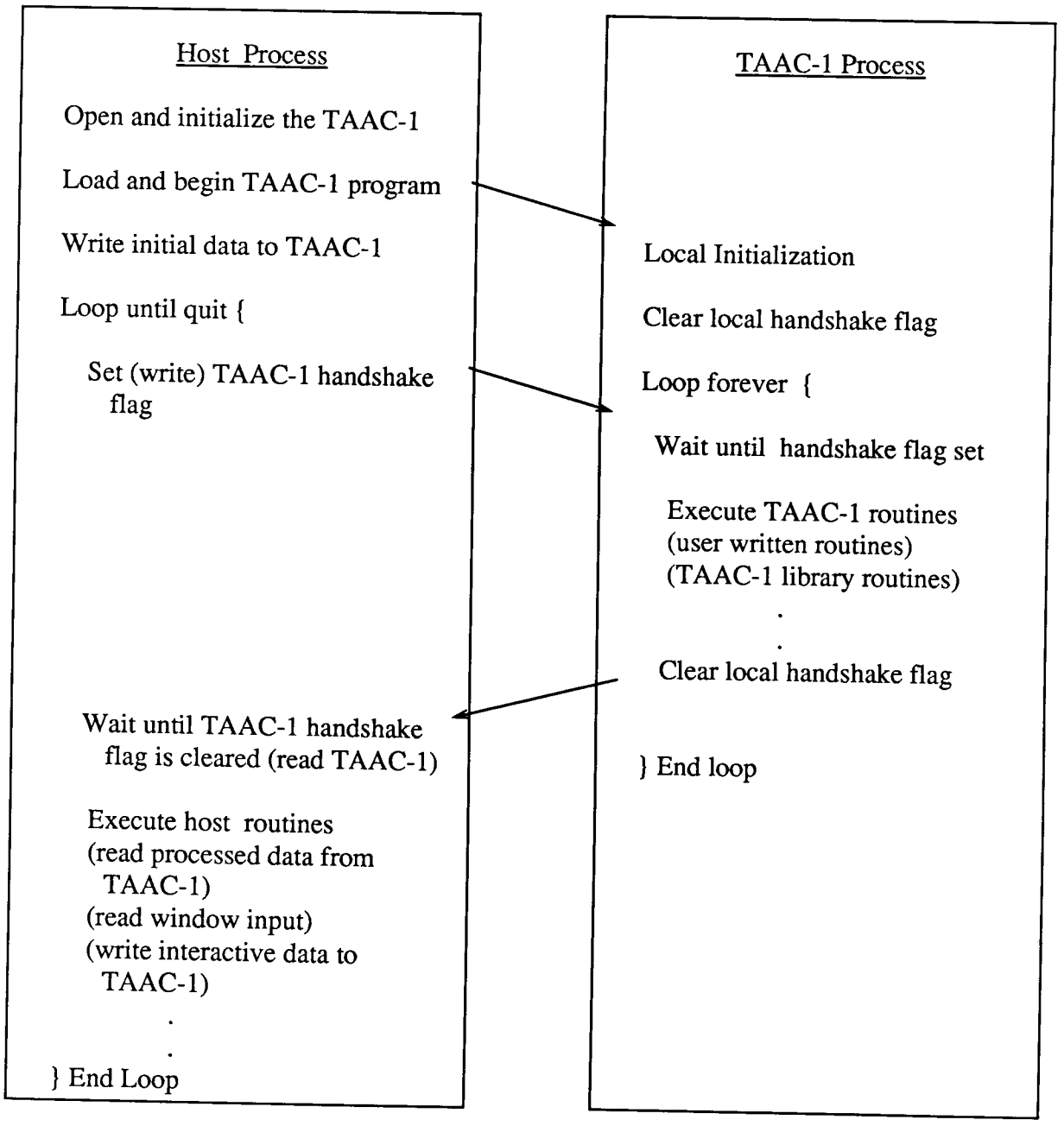


Figure 3-3 *Program Fragment Showing Host/TAAC-1 Handshaking*

Host Process

```
#include <taac1/taio.h>
#include "taacfile_map.h"

main() {
TA_HANDLE *tah;
int ioflag_val;
    .
    .
if ((tah= ta_open(0)) == NULL){
  printf("error opening TAAC");
  exit(-1);
}
if (ta_init(tah) == TA_FAILURE){
  printf("error on TAAC init");
  exit(-1);
}
if (ta_run(tah, "taacfile.abs")
  == TA_FAILURE){
  printf("error on ta_run");
  exit(-1);
}
/* write initial data to TAAC-1 */
ta_write(tah, &hostvar,
  sizeof(hostvar), TC_taacvar);
   .
   .
while (!done){
  ioflag_val = 1;
  ta_write(tah, &ioflag_val, sizeof
    (ioflag_val), TC_ioflag);
  while (ioflag_val==1)
    ta_read(tah,&ioflag_val,sizeof
      (ioflag_val),TC_ioflag);
      .
      .
  } /* end while not done */
} /* end main */
```

TAAC-1 Process

```
#include <taac1/builtin.h>

int ioflag = 0;

main() {
    .
    .
    .
while (1) {
  while (ioflag==0);  /*wait
                      until set */
    .
    .
    .
  ioflag = 0;
  } /* end while 1 */
} /* end main */
```

3.2. TAAC-1 Memory Usage

Variables used in TAAC-1 programs may be assigned storage in:

- registers (ALU RC or RD) - 64 registers in each ALU
- scratchpad memory (SRAM) - 16Kwords
- data/image memory (DRAM) - 2Mwords

ALU Registers

Frequently-used variables should be placed in registers, to significantly reduce the time required for variable access. Because of the large number of registers available, entire transformation matrices or convolution kernels can be stored in register variables. Designation of registers is done using standard C notation:

```
register datatype variablename;
```

You can also specify a particular ALU or register number. For example:

```
register RD int x;     /* specifies a register in ALU RD */
register float y @2;   /* specifies register #2 in ALU RC */
register float z @66;  /* specifies register #2 in ALU RD */
```

SRAM

SRAM holds the C stack; in addition, this is the default location for all global and static variables. Care must be taken not to overwrite the C stack with global or static variables that are too large. No runtime detection of stack overflow conditions exists.

DRAM

DRAM should be used for images and very large data structures. An example of a DRAM variable declaration is:

```
DRAM float z[1000];
```

This declaration does not specify the variable's location in DRAM. The TAAC-1 linker will perform the memory allocation; however, you must tell the linker which portions of the DRAM are available. (This prevents the linker from allocating memory you are using for images or other data not declared in your program.) The linker `-d` command line option allows you to specify the starting and ending addresses of each available segment of DRAM. As an example, to designate two

1Mbyte blocks available for DRAM variables, one at 0x80000 and one at 0x180000, use this linker command line option:

```
% talink -d 0x80000 0xbffff -d 0x180000 0x1bffff foo.obj...
```

See the linker chapter for more details.

You can also specify a particular DRAM address from within a TAAC-1 program by setting up a pointer to an absolute memory address. For example, to point to the start of an image stored at location 0x100000, a TAAC-1 program would contain these lines:

```
#define TADATA 0x100000
int *p;

p = (int *)TADATA;
```

Be aware that, if you store variables in DRAM, the current DRAM Mode Register setting will affect writes to those variables. For example, if bitmask mode is enabled, the current bitmask will be applied to all writes that use the AC or AI register. For more information, refer to the description of the DRAM Mode Register in the Hardware Overview chapter.

The built-in functions of the TAAC-1 compiler can also be used to access DRAM memory in 1D, 2D, or 3D modes. See the compiler chapter for details.

3.3. Reading and Writing Variables in TAAC-1 Programs

In order to read or write a variable in a TAAC-1 program, the host program must have the address of the variable in TAAC-1 memory space. To get this information, the variable must be made global, as shown in the preceding program fragment for the variable `ioflag`. When TAAC-1 programs are linked, the names and addresses of all global variables are written to a `.map` file. The entry for `ioflag` in the `.map` file might look like:

```
SY   _ioflag 0x30000000
```

This the absolute address in TAAC-1 memory space of the variable `ioflag`. Its address indicates that it is in SRAM (Scratchpad RAM) memory. (For more information about TAAC-1 memory space, see the hardware chapter.) To simplify the usage of this information, a

utility program, `tamakedef`, reads a `.map` file and produces a header (`.h`) file whose entry for `ioflag` might look like:

```
#define TC_ioflag 0x30000000
```

If you include this header file in your host program, you can write to any global variable in your TAAC-1 program, giving as the address the variable name prefixed by `TC_` and using any of the host library memory access routines, such as `ta_read()` and `ta_write()`. This symbolic referencing eliminates any need to know absolute TAAC-1 addresses.

Using Host Library Read/Write Routines

TAAC-1 address space is mapped to the host's virtual memory space. As a result, transfer of data between the host and TAAC-1 is straightforward. Because the TAAC-1 local bus address space is larger than the VME address space, it is necessary to select which type of TAAC-1 memory you want to access: scratchpad memory, data/image memory, program memory, or control registers. The host library memory access routines set the VME interface slave mode register (SMR) to address the correct memory type, and then perform the read or write.

NOTE: The TAAC-1 cannot act as VME bus master and read or write host memory.

More Direct Access to TAAC-1 Memory

It is sometimes desirable to avoid the relatively small overhead of the host library read/write routines, especially when you are randomly accessing TAAC-1 variables from the host, rather than reading or writing entire arrays. The host library has two routines for this purpose, `ta_map()` and `ta_use_map()`.

`ta_use_map()` sets up the SMR to address a particular TAAC-1 memory type. It is used in conjunction with `ta_map()`, which returns the host virtual memory address for the specified TAAC-1 memory address. The following code fragment illustrates the process of writing to variables in TAAC-1 data/image memory.

```
(assume TAAC-1 is open and initialized; tah points to the
TA_HANDLE interface structure)
```

```
#define TAACDEST 0x100000
{
int *p;
int a[100];
char *filename;
.
.
if( (p = ta_map(tah, TAACDEST))==NULL )
        exit(-1);
if(ta_use_map(tah,  TAACDEST)==TA_FAILURE)
        exit(-1);
p[0] = a[0];
p[4] = a[3];
.
.
}
```

`ta_use_map()` must be called each time the slave mode register is changed (that is, after any other host library routine has been called). These routines can also be used with the address of a variable listed in the `.map` file, as discussed in the previous section. For most cases, use the read/write routines provided with the library instead of these lower-level routines, since they add little processing overhead.

Note: direct data transfers between a file and TAAC-1 memory are not supported. While `ta_map()` and `ta_use_map()` can still be called, the data must be moved through an intermediate memory array.

3.4. Data Types

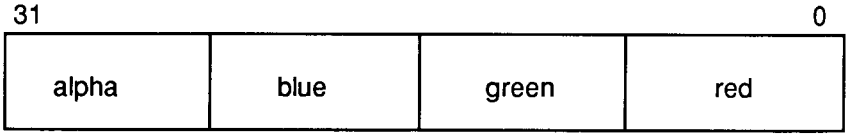
Since the TAAC-1 is a word-addressable machine, the smallest data type is 32 bits; this applies to types `char`, `short`, `int`, and `long`. The 32-bit data word requirement may affect your program in one or more of the ways discussed in this section.

Data Transferred from a Host Program

If you transfer an array of type `char` to TAAC-1 memory, it will be transferred using 32-bit writes. An array containing a sequence of alpha, blue, green, and red bytes:

byte 0	alpha
byte 1	blue
byte 2	green
byte 3	red
byte 4	alpha
byte 5	blue
.	.
.	.

would be transferred to TAAC-1 memory as a series of 32-bit words, in this order:



Structures Within TAAC-1 Programs

Structures inside programs compiled by the TAAC-1 C compiler may have a different size than they would have in the same program compiled by the host C compiler. This structure, for example, cannot be redefined as a single `int`, since on the TAAC-1 it is considered to contain four 32-bit values:

```
struct {
   char red;
   char green;
   char blue;
   char alpha;
}
```

3.5. Window Management

There are two ways to see TAAC-1 video in a Sun window:

- In your host application program, create a window using SunView or any window system, and fill its canvas with a solid TAAC-1 keying color.

- Use the TAAC-1 `tatool` utility, which provides a SunView window for displaying TAAC-1 video. `tatool` is useful for standalone TAAC-1 programs and for host programs without a window interface. See the utilities chapter for a guide to the use of `tatool`.

The TAAC-1 host library contains routines to display a TAAC-1 canvas in a Sun window and to control which portion of TAAC-1 image memory is displayed. Here are descriptions of the key routines:

ta_taac_canvas()

Displays a TAAC-1 canvas that has been created by the SunView `window_create()` function, and fills it with the Sun keying color from the configuration file `$TAAC1/hardware taconfig.<hostname>`. `ta_taac_canvas()` also calls `ta_set_display()` and `ta_set_view()`; see their descriptions below and in the host library chapter of the TAAC-1 Libraries manual..

The `tatool` utility calls `window_create()` and `ta_taac_canvas()` to create its window. For information on `window_create()`, see the *SunView Programmer's Guide.*

As an alternative to `ta_taac_canvas()`, the window functions can be provided by any window system without affecting communication with TAAC-1 routines. The TAAC-1 demo library contains *unsupported* window routines that can be used by persons unfamiliar with SunView. They are described in the demos chapter of the TAAC-1 Libraries manual.

ta_realign()

Reads the current location of the TAAC-1 canvas and moves the TAAC-1 video to match it, by calling the host routine `ta_set_window()`, described below. This function can be used after you have called `ta_taac_canvas()`. `ta_taac_canvas()` takes care of moving the TAAC-1 video in response to window events created by using the mouse; `ta_realign()` must be called only when the window moves without an event occurring (e.g., an explicit program-directed move).

ta_set_view()

Sets the 2D memory location (in pixel coordinates) that is to be mapped to the first visible pixel in the upper left corner of the displayed video. Location (0,0) is the upper left corner of TAAC-1 image memory and is the default set by `ta_init()` and by `ta_taac_canvas()`.

ta_set_display()

Sets the output video and sync sources, according to the hardware configuration file. It can set Sun-only, TAAC-1-only, or mixed TAAC-1 and Sun video. In a windowing situation, `ta_init()` sets Sun video

as the default. Therefore, your program may need to call `ta_set_display()` to display TAAC-1 or mixed video.

`ta_taac_canvas()` calls `ta_set_display()` with the argument `TA_ON`, which displays mixed video in single-configuration systems or TAAC-1 video in dual-configuration systems. Programs that call `ta_taac_canvas()` do not need to call `ta_set_display()`.

`ta_set_window()`

Sets the location and dimensions of the TAAC-1 window on the screen. If the view of the TAAC-1 frame buffer is to remain constant as the Sun window is moved, this function can be called to reset the window parameters. This function is only for programs that set up windows but do *not* call `ta_taac_canvas()` or the unsupported window routines.

Window Management Example

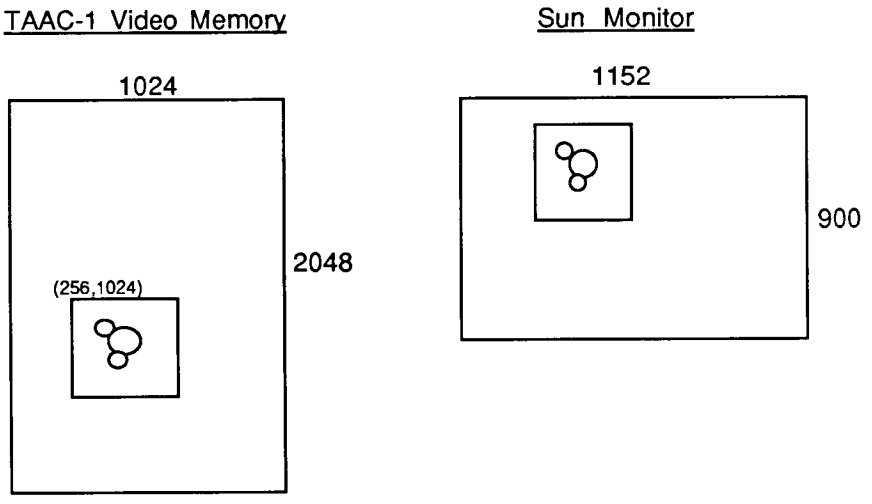
An example will further clarify the window functions. Assume the part of the TAAC-1 frame buffer to be displayed begins at the two-dimensional address x=256, y=1024 in TAAC-1 data/image memory and is 512 x 512. The Sun window (to be created) is also 512 x 512. The program shown below creates a SunView window containing a TAAC-1 canvas and maps (256, 1024) to be the first displayed pixel, as shown in the figure that follows. `window_create()` and `window_main_loop()` are SunView functions.

```
#include <suntool/sunview/h>
#include <suntool/canvas.h>
#include <taac1/taio.h>
Frame tatoolframe;
Canvas tatoolcanvas;
TA_HANDLE *tahandle;

main (argc, argv)
int argc;
char *argv[];
{
   if (!tahandle = ta_open (0))) {
   printf ("Error opening TAAC-1\n");
   exit (1);
   }
   if (ta_init (tahandle)) {
   printf("Error initializing TAAC-1\n");
   exit(1);
   }
```

```
    tatoolframe = window_create (NULL,FRAME);
    tatoolcanvas = window_create (tatoolframe, CANVAS, 0);

    /* display TAAC-1 canvas */
    ta_taac_canvas (tahandle, tatoolframe, tatoolcanvas);

    /* set first displayed pixel = 256, 1024 */
    ta_set_view (tahandle, 256, 1024);


    window_main_loop (tatoolframe);
    ta_close (tahandle);
}
```

For more information about the supported window routines, see the windows section of the host library chapter of the TAAC-1 Libraries manual.

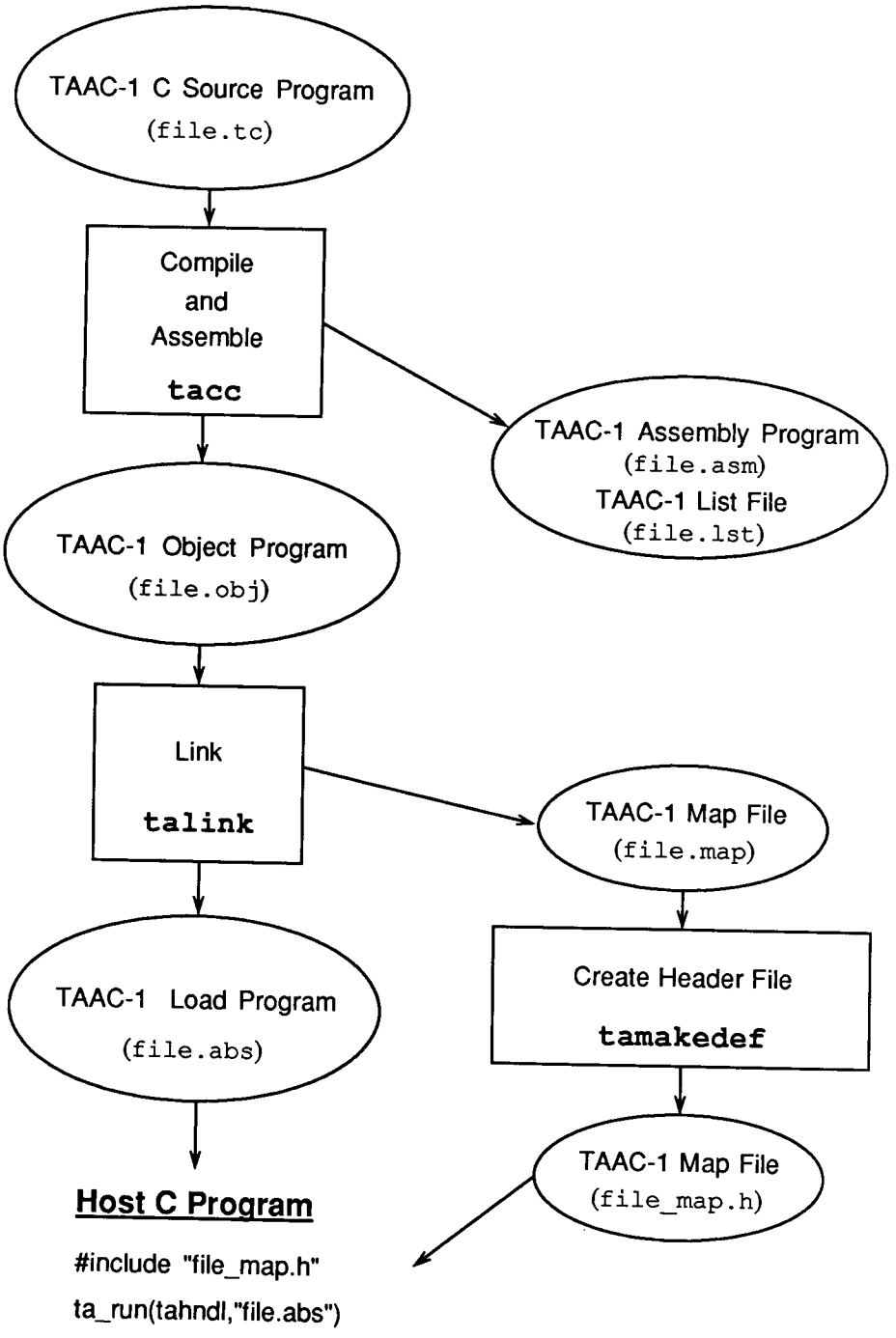
Figure 3-4 *Display of TAAC-1 Video in Sun Monitor*



3.6. Building a TAAC-1 Program

The chart on the next page illustrates the steps in building software for the TAAC-1 from a C source. The initial step is to *compile* the code. The TAAC-1 C compiler generates an assembly-code source file and automatically invokes the *assembler*. The resulting object files are then *linked* to create a load program (`.abs` file). The load program is *loaded* at runtime by the host program using the library routine `ta_run()`, or as a stand-alone routine using the utility program `tarun`.

Figure 3-5 *Building a TAAC-1 Program*



The linker also produces a `.map` file with global variable and subroutine address locations. The utility program `tamakedef` produces a header file from the `.map` file for inclusion by the host program. The tutorial contains example `.map` and header files.

The `taabs2o` Utility

You can link a TAAC-1 absolute file (`.abs`) with a host program, eliminating the need to load a separate `.abs` file. The utility `taabs2o` creates a `.o` file from a `.abs` file. This file can then be linked into the host program.

To load an absolute file that has been linked with a host program, call:

```
ta_runm (tahndl);
```

instead of:

```
ta_run (tahndl,"file.abs");
```

All other steps in the process remain the same.

The Makefile in the upcoming tutorial uses the `taabs2o` utility to link the TAAC-1 `.abs` file into the host executable.

TAAC-1 Development Notes

There are three ways to enhance program efficiency on the TAAC-1:

- Using the *built-in functions* of the TAAC-1 C compiler for fast access to specialized TAAC-1 components. These functions produce in-line code for direct access to the lookup tables, the vector ports, and AC register reads and writes, without the overhead of subroutine calls.

- Programming with the *TAAC-1 library routines*, which perform mathematical operations, graphics and image processing functions, and control functions. These routines have been optimized for the TAAC-1.

- Inserting *in-line assembly code* within C programs. In-line code is often substituted for program inner loops.

3.7. Tutorial

The tutorial describes two interdependent programs, one running on the host and the other running on the TAAC-1. The host program prompts the user for information, sends input data to the TAAC-1, and loads and runs the TAAC-1 program. The TAAC-1 program draws an image in TAAC-1 data/image memory which can be displayed in a Sun window using the `tatool` utility. You can select color bars, greyscale, or a solid-color image. If you select color bars or greyscale images, you can select the density of the individual bars. If you choose the single-color image, you can select the color.

To use this tutorial, you need:

- A Sun 3 or Sun 4 workstation with a TAAC-1 and its associated software. The tutorial assumes the use of a single-color monitor and Suntools software.
- A working knowledge of the C language.
- Basic UNIX and Sun experience, including familiarity with a text editor. Familiarity with `make` is also helpful.

The source files described in this tutorial are contained in the directory `$TAAC1/tutorial` and are printed at the end of this section. The tutorial assumes that you have the environment variable `TAAC1` assigned correctly, and that you have the directory `$TAAC1/bin` added to your path. See the *TAAC-1 Software Installation Guide*, part number 800-2441-xx, for details.

Host Program

For the host program, see the file `$TAAC1/tutorial/colors.c`, which contains these steps:

- open a link to the TAAC-1
- initialize the TAAC-1
- load and run the TAAC-1 program
- prompt the user for input and send data to TAAC-1 memory
- set the handshaking flag to tell the TAAC-1 to proceed

Items to Note in the Host Program

`#include <taac1/taio.h>`

This file contains the definitions and declarations used by the TAAC-1 host library.

`#include "colorbar_map.h"`

This file contains the addresses of all global variables declared in the TAAC-1 program. The utility `tamakedef` reads the `.map` file generated by the TAAC-1 linker and produces a header file consisting of the name of each TAAC-1 global variable, prefixed by `TC_`, and its absolute TAAC-1 memory address. The library read/write routines use these addresses to access TAAC-1 memory.

`TA_HANDLE *tahndl;`

`ta_open` returns a pointer to this structure, which is defined in `taio.h`. This pointer is the first argument in other host library routines.

`ta_open(), ta_close()`

These routines open and close the link to the TAAC-1. `ta_open` must be the first host library routine called.

`ta_init()`

Sets the default values in the TAAC-1. This is usually the second host library routine called.

`ta_set_display()`

Sets the display to Sun video, TAAC-1 video, or mixed (TAAC-1 video in a Sun window). `ta_init` sets the default to Sun video only, which must be changed to view TAAC-1 video.

`ta_runm()`

Loads and runs the TAAC-1 program which has been linked into the host program.

`ta_read(), ta_write()`

Reads and writes TAAC-1 memory. These routines write to 1D addresses. To load an image to TAAC-1 memory, see the host library chapter listing (in the TAAC-1 Libraries manual) for `ta_write2d()`.

`TC_ioflag`

`TC_ioflag` is defined to be the address of the handshaking flag on the TAAC-1. When the host has written the input data to TAAC-1 memory, it sets the handshaking flag in TAAC-1 memory to tell the TAAC-1 to proceed. The TAAC-1 clears this flag to notify the host that it has finished.

`usleep()`

The host program calls usleep(100) inside the loop that checks the handshaking flag, to avoid tying up all the host processor's resources.

TAAC-1 Program

The TAAC-1 program, contained in `$TAAC1/tutorial/colorbar.tc` consists principally of standard C code. It performs these steps in a continuous loop:

- Clear the handshaking flag (`ioflag`) and loop reading this flag, waiting for the host to send input data. The host will set this flag when it has finished writing to TAAC-1 memory.
- When the handshaking flag has been set by the host, call the appropriate image routine (color bars, greyscale wedges, or solid color).

Items to Note in `colorbar.tc`

`#include <taac1/builtin.h>`

This file contains defines used by TAAC-1 programs, including macro definitions for built-in functions. The built-in functions are a set of macros that directly address specific elements of the TAAC-1 processor. These macros can be used to write to TAAC-1 registers such as Address Count (AC), DRAM Mode (AM), and the Barrel Shift Input (BR), and also to write to data/image memory using the AC register.

`main()`

Notice that both `colors.c` and `colorbar.tc` have a `main()`. Each program is compiled and linked separately; in addition, the TAAC-1 absolute (executable) file can be linked into the host program so that it can be easily downloaded from the host, avoiding file I/O and elminating the need to maintain two executable files for the application task.

`ioflag, command[3]`

These global variables, along with their addresses, are written to the `.map` file by the TAAC-1 linker `talink`. `ioflag` is the handshaking flag used to synchronize with the host program.

`output(WR_AM, TA_AM2D)`

Sets the DRAM Mode Register (AM) in the TAAC-1 for 2D addressing. `output()` is a built-in function used to write to specified TAAC-1 registers. `output()` is a built-in compiler function; `TA_AM2D` is defined in the `builtin.h` include file.

`set_ac(), upd_ac(), write_ac()`

The first two functions set and update, respectively, the Address Count Register (AC). `write_ac()` writes to the address contained in AC. The three functions together write each pixel of the image. All three are built-in compiler functions.

Building and Running the Program

The steps used to build and run this tutorial program are:

1. Compile and link the TAAC-1 program.
2. Use the `tamakedef` utility to generate a header file containing TAAC-1 global addresses.
3. Use the `taabs2o` utility to convert the TAAC-1 absolute image file to an object file that can be linked into a host program.
4. Compile and link the host program.

5. Execute the host program.

Compile and Link the TAAC-1 Program

The TAAC-1 program must be compiled and linked first, since the TAAC-1 linker generates global variable addresses needed by the host program. To compile `colorbar.tc`, enter:

```
% tacc -c -fsingle colorbar.tc
```

The TAAC-1 C compiler (`tacc`) generates the file `colorbar.asm`, containing the assembly language source code, and then goes on to assemble the `.asm` file, generating `colorbar.obj`, the relocatable object module. (`.obj` files are analogous to the `.o` files produced by the standard C compiler.)

The `-c` option includes C source code as comments in the assembly source and listing files. The `-fsingle` option ensures that floating-point operations are single-precision and passes float arguments without converting them to doubles. This program does not declare any floating-point variables; however, it is a good idea to specify `-fsingle` as a matter of course for all programs, except when you want double precision, which is not yet supported. For the compiler to produce an assembly code listing, use the `-l` option. The C compiler chapter describes other compiler options.

After compilation, the object files (in this case, only one file) need to be linked. Enter:

```
% talink colorbar
```

The TAAC-1 linker, `talink`, produces an absolute image file with a `.abs` suffix. The image file is formatted for the TAAC-1. The linker also produces a `.map` load map file containing names and TAAC-1 memory addresses of program segments and global symbols in the program. See the linker chapter for more on map file formats. You can specify the absolute file name using the `-o` option, and the load map file name by using the `-m` option. If you omit these options, the linker by default will use the name of the first file in the link list. The command:

```
% talink -o colorbar.abs -m colorbar.map colorbar.obj
```

would produce the same result as the simpler `talink` command above.

If this tutorial program used any of the mathematical, graphics, image processing, or control functions from the TAAC-1 library, the link list would also need to contain `-ltaac1`.

Use `tamakedef` to Generate an Include File for the Host Program

The `tamakedef` utility reformats the `.map` file into a set of defines that can be included in a host C program. Run `tamakedef` by entering:

```
% tamakedef colorbar.map colorbar_map.h
```

The header (`.h`) file contains defines for the TAAC-1 memory address of each routine and global variable defined in `colorbar.tc`. The load map generated by the linker, and the corresponding header file produced by `tamakedef`, are shown below:

Figure 3-6 *Examples of Load Map and Header Files*

colorbar.map:

```
SG regdump_code 0 0x5 0x376
SY regdump 0x5
SG colorbar_code 0 0x377 0x411
SY _main 0x399
SY _S.main 0x377
SY _makcolorbar 0x3bd
SY _S.makcolorbar 0x39a
SY _makgreybar 0x3ed
SY _S.makgreybar 0x3c6
SY _maksolidcolor 0x40c
SY _S.maksolidcolor 0x3f6
SG c_startup 0 0x0 0x4
SY _cstart_ 0x0
SG regdump_sram 1 0x30000000 0x30000095
SY DPCONTROL 0x30000000
SY DUMPLOC 0x30000001
SG colorbar_data 1 0x30000096 0x300000a1
SY _ioflag 0x3000009e
SY _command 0x3000009f
```

colorbar_map.h:

```
#define TCregdump 0x5
#define TC_main 0x399
#define TC_makcolorbar 0x3bd
#define TC_makgreybar 0x3ed
#define TC_maksolidcolor 0x40c
#define TC_cstart_ 0x0
#define TCDPCONTROL 0x30000000
#define TCDUMPLOC 0x30000001
#define TC_ioflag 0x3000009e
#define TC_command 0x3000009f
```

Use `taabs2o` to Convert the Absolute Image File to an Object File

The `taabs2o` utility converts `.abs` files to `.o` files suitable for linking with a host program. The conversion produces a single executable containing both host and TAAC-1 code. When the conversion is complete, the function `ta_runm()` loads the TAAC-1 program into TAAC-1 memory and begins its execution. To use `taabs2o`, enter:

```
% taabs2o colorbar.abs colorbarabs.o
```

It is also possible to skip this step and keep separate executables, one for the TAAC-1 and one for the host. See the utilities chapter for details.

Compile and Link the Host Program

The TAAC-1 portion of the tutorial is complete. Now we can compile the host program, which "includes" the header file produced by the `tamakedef` utility. To compile the host portion, enter:

```
% cc -o colors colors.c colorbarabs.o -ltaac1
```

`-ltaac1` in a host link command refers to the TAAC-1 host library of initialization, processor control, memory access, and video control routines.

The Makefile in the `$TAAC1/tutorial` directory compiles and links all files in this tutorial.

Execute the Host Program

This program must be executed from within a Suntools environment, if the TAAC-1 video is to be displayed in a Sun window. Before executing the program, open a TAAC-1 display window by entering:

```
% tatool -t&
```

Move this window, if necessary, to the upper left corner of your screen, because the initialization routine `ta_init()` in the host program maps (0,0) of the TAAC-1 video to the upper left corner of the Sun monitor.

Execute the program by entering:

```
% colors
```

The program prompts you to choose the display type: colorbars, greyscale, or solid color; for colorbars or greyscale, it also asks for bar density; for solid color images, it asks for a color.

The source code and Makefile for this program appear on the next pages.

```
#*****************************************************************************
#*                                                                           *
#*      Copyright (c) 1988, Sun Microsystems Inc., Raleigh, North Carolina   *
#*                                                                           *
#*                                                                           *
#*****************************************************************************
# @(#)Makefile1.5 88/05/15
# makefile for colorbars
# include the predefined rules for building TAAC-1 programs
include /usr/include/taac1/makerules

# define the constants
CFLAGS = -O                                    # flags for cc
LIBS = -ltaac1                                 # libraries for host program
OBJS = colors.o colorbarabs.o# object files for host program

TCFLAGS = -c -fsingle# flags for tacc
TLFLAGS =                                      # flags for talink
TLIBS = -ltaac1 # libraries for TAAC-1 program
TOBJS = colorbar.obj # object files for TAAC-1 program

# compile (implicit) and link for host program
colors: $(OBJS)
      $(CC)  $(CFLAGS) -o colors $(OBJS) $(LIBS)

# dependencies for host program
colors.o: colors.h colorbar_map.h

#create a .o file from a TAAC-1 .abs file (TAABS2O is defined in makerules)
colorbarabs.o: colorbar.abs
      $(TAABS2O) colorbar.abs colorbarabs.o

#create a .h file from the TAAC-1 .map file (TAMAKEDEF is defined in makerules)
colorbar_map.h: colorbar.map
      $(TAMAKEDEF) -d -c colorbar.map colorbar_map.h

# compile (implicit) and link TAAC-1 program (TALINK is defined in makerules)
# makerules also describes how to make a .obj file from a .tc file
colorbar.abs colorbar.map: $(TOBJS)
       $(TALINK) $(TLFLAGS) $(TOBJS) -o colorbar.abs $(TLIBS)

# dependencies for TAAC-1 program
colorbar.obj: colors.h

clean:
      rm -f $(TOBJS) $(OBJS) colorbar.asm colorbar.abs colorbar_map.h \
            colorbar.map colorbar.lst core *%

install: colors

spotless: clean
      sccs clean
```

```
/*********************************************************************/
/*                                                                   */
/*    Copyright (C) 1988, Sun Microsystems                           */
/*                                                                   */
/*********************************************************************/
/*                                                                   */

/*
 * Tutorial Program - colors.c (Host Code)
 *
 *        Loads TAAC-1 program, asks for user input and
 *      sends data to TAAC-1, which generates the images.
 *
*/


#include <stdio.h>
#include <taac1/taio.h>                    /* TAAC-1 host include file */

#include "colors.h"                        /* colorbar defines */
#include "colorbar_map.h"                  /* global defines from tamakedef */
#define GO 1                               /* host tells TAAC-1 to proceed */


main ()
{

    TA_HANDLE *tahndl;                     /* Handle returned from ta_open */
    TA_HANDLE *taacstartup ();             /* routine to open, init TAAC-1 */
    void taacgo ();
    void taacdone ();
    void write_error ();
    void read_error ();
    int command[3];                        /* command words for TAAC-1:
                                              word 0: bartype - COLORBAR,
                                                      GREYSCALE, or SOLIDTYPE
                                              word 1: colorbar density
                                              word 2: color index for solid color
                                                                          */

                                           /* local copies of command words */
    int bartype;                        /* COLORBAR, GREYSCALE, or SOLIDTYPE */
    int barden;                            /* color bar density */
    int colorind;                          /* color index for solid color */
    int ioflag;                            /* flag value */
    int tastatus;                          /* returned status */


    /* open, initialize, load, run TAAC-1 */
    tahndl = taacstartup ();

    /* get user input and send commands to TAAC-1 */
```

```
while (1) {

  bartype = -2;
    while ((bartype < QUIT) || (bartype > MAXTYPE)) {
        /* get test type selected */
        printf(" Enter test type.[0-2 - Imagetype, 3 - Help, -1 - Quit]> ");
        scanf("%d",&bartype);
    }
  command[0] = bartype;

    switch(bartype) {
        case COLORBAR:                          /* make a color bar */
         barden = -1;
            while ((barden < 0) || (barden > 7)) {
                printf("        Enter colorbar density.[0-7]> ");
                scanf("%d",&barden);
            }
          command[1] = barden;
            break;

        case GREYSCALE:                 /* make grey scale */
         barden = -1;
            while ((barden < 0) || (barden > 2)) {
                printf("        Enter grey bar density.[0-2]> ");
                scanf("%d",&barden);
         }
         command[1] = barden;
            break;

        case SOLIDTYPE:
         colorind = -1;
         while ((colorind < 0) || (colorind > 7)) {
                printf("        Enter number: \n");
                printf("           0   black\n");
                printf("           1   blue\n");
                printf("           2   green\n");
                printf("           3   yellow\n");
                printf("           4   red\n");
                printf("           5   magenta\n");
                printf("           6   cyan\n");
                printf("           7   white\n");
                scanf("%d",&colorind);
            }
         command[2] = colorind;
         break;

       case QUIT:                 /* do nothing   */
           break;

     case HELP:
       default:                        /* if input wrong character, same as help */
           printf("       Options include:\n");
```

```
            printf("           -1   Quit\n");
            printf("            0   Colorbars\n");
            printf("            1   Grey Scale\n");
            printf("            2   Solid Color\n");
            printf("            3   This Help Menu\n");
            break;

    } /* end switch */


  if (bartype == QUIT) {
      break;
  }

  else {
     /* wait for TAAC-1 to finish processing previous command
        Note: TAAC-1 program, loaded in taacstartup routine,
          will set value at address TC_ioflag to 0 when ready.
          Variables with name TC_... are defined in file
          colorbar_map.h created by tamakedef utility. */
      ioflag = 1;
      while (ioflag) {
         if ((tastatus = ta_read (tahndl, &ioflag, sizeof(ioflag),
            TC_ioflag)) != sizeof(ioflag))
               read_error (tastatus);
          usleep(100);
        }

        /* send command words to TAAC-1 */
        if ((tastatus = ta_write (tahndl, command, sizeof(command),
                TC_command)) != sizeof(command)) {
            write_error (tastatus);
        }

        /* set handshaking flag to tell TAAC-1 to proceed */
      ioflag = GO;
        if ((tastatus = ta_write (tahndl, &ioflag, sizeof(ioflag),
                TC_ioflag)) != sizeof(ioflag)) {
            write_error (tastatus);
        }
  }

} /* end while (1) */

/* wait for TAAC-1 to finish */
while (ioflag) {
  if ((tastatus = ta_read (tahndl, &ioflag, sizeof(ioflag),
          TC_ioflag)) != sizeof(ioflag)) {
       read_error (tastatus);
    }
  usleep(100);
}
```

```
    /* cause TAAC-1 processor to loop at PC = 0 */
    ta_zero(tahndl);

    /* turn off TAAC-1 video */
    ta_set_display (tahndl, TA_DEFAULT);

    /* close TAAC-1 link */
    ta_close(tahndl);

} /* end main*/



/********taacstartup********/

/* open TAAC-1 and begin program execution */
TA_HANDLE *taacstartup ()
{
    TA_HANDLE *tahndl;                      /* handle returned from ta_open() */
    int tastatus;                           /* returned status */

    /* open TAAC-1 link */
    if ((tahndl = ta_open (0)) == NULL) {
        fprintf (stderr, "Error opening TAAC-1\n");
        exit (-1);
    }

    if ((tastatus = ta_init (tahndl)) != TA_SUCCESS) {
        fprintf (stderr, "ta_init failed, returned %d\n", tastatus);
        exit (-1);
    }

    if ((tastatus = ta_set_display (tahndl, TA_ON)) != TA_SUCCESS) {
        fprintf (stderr, "ta_set_display failed, returned %d\n", tastatus);
        exit (-1);
    }

    if ((tastatus = ta_runm (tahndl)) != TA_SUCCESS) {
        fprintf (stderr, "ta_runm failed, returned %d\n", tastatus);
        exit (-1);
    }
    return (tahndl);
}



/********write_error********/

void write_error(status)
int status;
{
    fprintf(stderr,"ta_write failed, returned  %d\n",status);
    exit(-1);
}
```

```
/********read_error********/

void read_error(status)
int status;
{
    fprintf(stderr,"ta_read failed, returned  %d\n",status);
    exit(-1);
}
```

```
/*********************************************************************/
/*                                                                   */
/*      Copyright (C) 1988, Sun Microsystems        */
/*                                                                   */
/*********************************************************************/
/*                                                                   */

/*
 * Tutorial Program - colorbar.tc (TAAC-1 code)
 *
 *    Receives user input from the host program and creates
 *    simple images.
 *
 * Use 'tatool -t' to view the results in a TAAC-1 window.
 *
 */

#include <taac1/builtin.h>
#include "colors.h"

#define ADD_ONE_RGB 0x00010101

static int rgba[8] = { BLACK,BLUE,GREEN,YELLOW,RED,MAGENTA,CYAN,WHITE};

void makcolorbar();
void makgreybar();
void maksolidcolor();

/* Host-TAAC globals */
int ioflag;                                  /* Host-TAAC handshaking flag */
int command[3];                              /* command words for TAAC-1:
                                                word 0: bartype - COLORBAR,
                                                        GREYSCALE, or SOLIDTYPE
                                                word 1: colorbar density
                                                word 2: color index for
                                                        solid color */

main()
{
    register int bartype;                    /* bartype - COLORBAR,
                                                GREYSCALE, or SOLIDTYPE */


   /* Change DRAM addressing MODE to 2-D */
    output (WR_AM, TA_AM2D);

    /* Clear ioflag to tell host the TAAC-1 is ready; the host
       will set this flag when data is ready */
    ioflag = 0;

    while(1) {
```

```
        while(ioflag == 0);                    /* wait to be reset from host */

      bartype = command[0];
        switch (bartype) {
            case COLORBAR :
                makcolorbar (command[1]);
                break;

            case GREYSCALE :
                makgreybar (command[1]);
                break;

            case SOLIDTYPE :
                maksolidcolor (command[2]);
                break;

        } /* end of switch bartype */

        ioflag = 0;                            /* Signal host that TAAC-1 is done */

    } /* end of loop forever */

} /* end main*/

/********COLOR BARS********/

void makcolorbar(barden)
register int barden;
{
    register int y;
    register int width;
    register int num_bars;
    register int bar;
    register int barx;
    register int color = 0;

    width = (1 << (7 - barden));              /* width of each bar in pixels */
    num_bars = (1 << (2 + barden));          /* number of bars across 512 pixels */


    for (y=0; y < IMAGESIZE; y++) {           /* start y loop */

      set_ac(AC_LDX|AC_LDY|AC_LDZ, y<<16);
        for (bar=0; bar < num_bars; bar++) {          /* start bar loop */

            color = rgba[bar & 7];                    /* color this bar */

            for (barx = 0; barx < width; barx++) {

                write_ac(color);                      /* write pixel */
                upd_ac(AC_INCX | AC_INCY);            /* increment x adrs */
```

```
            }  /* end for each pixel this bar */
        }  /* end for each bar this line */

    }  /* end for each line */
}  /* end makcolorbar */

/********GREY SCALE********/

void makgreybar(barden)
register int barden;
{
    register int y;
    register int width;
    register int num_bars;
    register int bar;
    register int barx;
    register int color = 0;

     width = (1 << (2 - barden));          /* width of each bar in pixels */
    num_bars = (1 << (7 + barden));      /* number of bars across 512 pixels */

    for (y=0; y < IMAGESIZE; y++) {        /* start y loop */

      /* load address of (0, y) to AC register. Note that y address
         must be shifted left 16 bits */
      set_ac(AC_LDX|AC_LDY|AC_LDZ, y<<16);

        for (bar=0; bar < num_bars; bar++) {/* start bar loop  */
            for (barx = 0; barx < width; barx++) {

                write_ac(color);                    /* write pixel     */
                upd_ac(AC_INCX | AC_INCY);          /* increment x adrs*/

            }  /* end for each pixel this bar */

          /* set new color   */
            if ((color & 0xffL) == 0xffL)
              color = 0;
            else
              color += ADD_ONE_RGB;

        }  /* end for each bar this line*/

      color = 0;

    } /* end for each line */
}  /* end makgreybar*/

/********SOLIDCOLOR********/

void maksolidcolor(colorind)
register int colorind;
```

```
{
    register int x,y;
    register int color;

    color = rgba[colorind];

   /* Load starting address  of (0, 0) to AC register */
   set_ac(AC_LDX | AC_LDY | AC_LDZ, 0);

   for (y=0; y < IMAGESIZE; y++) {                    /* start y loop */

      /* load address of (0, y) to AC register. Note that y address
         must be shifted left 16 bits */
      set_ac(AC_LDX|AC_LDY|AC_LDZ, y<<16);
        for (x = 0; x < IMAGESIZE; x++) {

            write_ac(color);                          /* write pixel     */
            upd_ac(AC_INCX | AC_INCY);                /* increment x adrs*/

      }  /* end for each pixel this scanline */

      upd_ac(AC_INCZ);                                /* increment y adrs */

    } /* end for each line */
}     /* end maksolidcolor */
```

3.8. Example Programs: Introduction

The remaining sections of this chapter present examples of TAAC-1 programs used to:

- Draw double-buffered and channel-buffered images
- Draw an image in the overlay channel
- Demonstrate the blink mask
- Demonstrate the graphics library

The example programs use the TAAC-1 library and must be linked with the option `-ltaac1`. The source files, and a makefile called `Makexample`, are available in the directory `$TAAC1/tutorial`. To compile and link these programs, use the command:

```
% make -f Makexample all
```

`db.abs`, `dbchan.abs`, `overlay.abs`, and `blink.abs` are standalone TAAC-1 programs, without a host process. To run these programs, use the `tainit` utility to initialize the TAAC-1 and `tatool` to call up a TAAC-1 window . Then use the `tarun` utility to run the program:

```
% tainit -m
% tatool -t&
% tarun <absolute file name>
```

For more information about these utilities, see the utilities chapter.

The last example describes the use of the graphics library, with a host process (`poly`) and a TAAC-1 process (`drawpoly.abs`). With a single-monitor configuration, use `tatool` to view the displayed results.

3.9. Example Programs: Double-Buffering, Channel-Buffering

Double-buffering is often used to facilitate the perception of smooth motion or transition from one image to the next. The viewer typically sees one image while the next image is being generated. On the TAAC-1, you can double-buffer spatially or in depth:

- *Spatial double-buffering*. Define two buffers in different areas of data/image memory (DRAM). Use `t_set_view()` to display the

contents of one buffer while your program draws in the other buffer; then swap buffers and repeat. This method can be used for full-color or pseudocolor images.

- *Channel-buffering (double-buffering in depth).* Display a pseudocolor image from a single channel (red, green, blue or alpha) of image memory, while you are generating the next image in another channel of the *same area* of memory. With this method, the x,y addressing stays the same, no matter which buffer you are using.

Both double-buffering examples generate the same image, a rectangle that moves in a circular motion, using the `t_rect()` TAAC-1 library routine. The first example uses two separate DRAM buffers, one in Bank A and one in Bank B, and changes the y-offset variable to write to one or the other of these buffers. The second example alternates writing and displaying between the red and green channels. In the code for both examples, important lines appear in boldface text.

Double-Buffering in Banks A and B

The first example program, `db.abs`, calls the library routine `t_rect()` to draw a full-color rectangle. While `t_rect()` is drawing in the work buffer of Bank A, the program is displaying out of the buffer in Bank B. Bank A of memory contains y addresses 0-1023; Bank B contains y addresses 1024-2047. When `t_rect()` is finished, the work buffer and display buffer are swapped.

The basic algorithm for double buffering between Bank A and Bank B is:

call `t_set_view()` to set the display buffer
    while (...)
        wait for vertical interval, to be sure `t_set_view()` is done
        erase the working buffer
        draw into the working buffer
        call `t_set_view()` to display the working buffer

Items to Note in `db.tc`

```
#include <taac1/t_math.h>
```

`t_math.h` contains the function prototypes for the TAAC-1 mathematical library functions.

```
yoff = 0;
```

Initializes `yoff`, the y-offset to the working buffer.

```
t_set_view ()
```

Sets the address of the first displayed pixel. Takes effect at the end of the next vertical interval.

```
yoff ^= 0x400;
```

Toggles `yoff` between 0 and 1024, to swap working buffers.

```
t_erase (0, yoff, 512, 512, BLACK);
```

Erases the working buffer using serial writes. Data/image memory is divided into two banks, Bank A (y addresses 0-1023) and Bank B (y addresses 1024-2047). There is a high-speed port (vector port) dedicated to each bank. The display controller has access to these ports, for image refresh; and the processor has access to the same ports, for fast serial reads and writes. However, the processor should access the same bank of memory that is being displayed *only* during the vertical blanking interval.

In this example, therefore, `t_erase` must take place *during* the vertical blanking interval or *after* the vertical interval, when `t_set_view` will have swapped the display and working buffers.

```
t_rect (x, y, 100, 100, CYAN);
```

Library routine `t_rect()` draws a rectangle. Adding `yoff` to the y coordinate allows `t_rect` to write to the current working buffer. For more information, refer to the graphics section of the TAAC-1 library chapter.

```
/*****************************************************************/
/*                                                               */
/*    Copyright (C) 1988, Sun Microsystems                       */
/*                                                               */
/*****************************************************************/
/*                                                               */

/* db.tc: draws a double-buffered rectangle at the end of an (imaginary)
100-pixel arm that sweeps around in a circle with a center at (256, 256) */

#include <taac1/builtin.h>
#include <taac1/t_math.h>

/* define colors */
#define BLACK 0x0
#define CYAN 0xffff00

#define DEG2RAD 3.14159/180./* degrees-to-radians conversion */
#define RADIUS 100.                        /* length from center to upper left
                                              corner of rectangle */
#define XYCENTER 256                       /* center coordinates */
#define INC 2.                             /* increment in degrees */
main()
{
      register int x,y;                    /* x and y coordinates */
      register float deg, radians;         /* angle in degrees, radians */
      register int yoff;                   /* y-offset to working buffer */

      /* initialize working buffer offset and set display buffer */
      yoff = 0;
      t_set_view (0, yoff);

      /* initialize angle */
      deg = 0;

      while(1) {

          /* toggle buffer offset between 0 and 1024 */
            yoff ^= 0x400;

          /* wait for vertical interval, so that t_set_view will
             take effect, and erase new working buffer */
            while (cc(CC_VERT));
            while (!cc(CC_VERT));
            t_erase (0, yoff, 512, 512, BLACK);

          /* increment angle and convert to radians */
          deg = deg + INC;
          radians = deg * DEG2RAD;
          /* calculate x and y coordinates (add y offset to y coordinate) */
          x = (int)(RADIUS * t_cos(radians)) + XYCENTER;
          y = (int) (RADIUS * t_sin(radians)) + XYCENTER + yoff;
```

```
            /* draw 100x100 rectangle */
            t_rect (x, y, 100, 100, CYAN);

            /* display buffer we have just written to */
            t_set_view (0, yoff);
        }

    }
```

Channel-Buffering

The next example, `dbchan.abs`, draws pseudocolor rectangles in channels 0 and 1 (the "red" and "green" channels). While it is displaying out of the red channel, it draws in the green channel, and vice versa. The basic algorithm for channel buffering is:

call `t_channel_select()` to select the channel to be displayed
while (...)
    set the bitmask to write to the work channel
    set the wordmask to erase only the work channel
    wait for beginning of vertical interval
    erase the work channel
    draw into the work channel
    call `t_channel_select()` to display the work channel
    toggle the channel-select variable

Items to Note in `dbchan.tc`

```
#define CYAN 0x101
```

Defines the rectangle shade for both the red and green channels. The bitmask determines which bitplanes are written to each time.

```
t_set_bitmask_mode (TA_ON);
```

Turns on bitmask mode. If bitmask mode is OFF (the default), the bitmask is ignored.

```
t_set_channel_select (chan_select);
```

Enables one or more video channels. In this example, `chan_select()` is toggled to enable either the red or green video channel.

```
amsav = input(RD_AM) & ~TA_AMWM_MASK | TA_AMWM;
output(WR_AM, amsav | chan_select);
```

Sets the DRAM Mode Register (AM) for wordmask mode and sets the word mask for the proper channel, for use by `t_erase()`.

```
t_erase(0, 0, 512, 512, BLACK);
```

`t_erase()` is a library function used to erase TAAC-1 image memory using the fast vector ports. Vector port writes ignore the bit mask. However, when word mask mode is enabled, the wordmask acts as a channel mask for writes to data/image memory.

Because we are viewing Bank A (y-addresses 0-1023) while we are erasing it, we must call `t_erase` during the vertical interval. To ensure that we are at the start of the vertical interval, we must first wait for active video and then wait for the start of the vertical interval.

```
t_set_bitmask(chanmask[chan_select]);
```

Sets the bitmask to protect the channel being displayed and to enable writes to the "working" channel. The bitmask applies to random writes to data/image memory, not to vector port writes.

```
chan_select ^= 0x3;
```

Toggles the display and working channels.

```
/*****************************************************************************/
/*                                                                           */
/*     Copyright (C) 1988, Sun Microsystems                  */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*
* Example program: dbchan.tc
*
*      Draws a double-buffered rectangle at the end of an (imaginary)
*      100-pixel arm that sweeps around in a circle with a center at
*      (256, 256) . Double-buffered in the red and green channels.
*
*/

#include <taac1/builtin.h>
#include <taac1/t_math.h>

/* define pseudocolors for red and green channels */
#define BLACK 0x0
#define CYAN 0x101

#define RED_CHAN 0x1                                    /* red channel */
#define GREEN_CHAN 0x2                                  /* green channel */

#define DEG2RAD 3.14159/180.                /* degrees-to-radians conversion */
#define RADIUS 100.                         /* length from center to upper left
                                               corner of rectangle */
#define XYCENTER 256                        /* center coordinates */
#define INC 2.                              /* increment in degrees */
main()
{
      static int chanmask[] = {0, 0xff, 0xff00};   /* bitmasks for red
                                                        and green channels */
      char red[2], green[2], blue[2];               /* colormap entries */
      register int x,y;                            /* x and y coordinates */
      register float deg, radians;           /* angle in degrees, radians */
      register int amsav;                           /* saved AM register */
      register int chan_select;                    /* channel-select toggle */

      /* set channel select to display red channel */
      chan_select = RED_CHAN;
      t_set_channel_select (chan_select);

      /* load red and green colormaps with black and cyan colors */
      red[0] = 0;
      green[0] = 0;
      blue[0] = 0;
      red[1] = 0;
      green[1] = 0xff;
      blue[1] = 0xff;
      t_set_colormap (TA_RED, 0, 2, red, green, blue);
```

```
    t_set_colormap (TA_GREEN, 0, 2, red, green, blue);

    /* turn on bitmask mode */
    t_set_bitmask_mode (TA_ON);

    /* set AM register for wordmask mode, mask out wordmask */
    amsav = input(RD_AM) & ~TA_AMWM_MASK | TA_AMWM;

    /* set AM register to write to green channel */
    chan_select = GREEN_CHAN;
    output(WR_AM, amsav | chan_select);

    /* initialize angle */
    deg = 0;

    while(1) {
        /* set bitmask to write to work buffer */
        t_set_bitmask(chanmask[chan_select]);

        /* erase work buffer (non-displayed channel) during
           vertical interval. */
        while (cc(CC_VERT));
        while (!cc(CC_VERT));
        t_erase (0, 0, 512, 512, BLACK);

        /* increment angle and calculate x and y coordinates */
        deg = deg + INC;
        radians = deg * DEG2RAD;
        x = (int)(RADIUS * t_cos(radians)) + XYCENTER;
        y = (int)(RADIUS * t_sin(radians)) + XYCENTER;

        /* draw 100x100 rectangle */
        t_rect (x, y, 100, 100, CYAN);

        /* display channel we have just written to */
        t_set_channel_select (chan_select);

        /* toggle channel-select and set AM register to write to
           new work buffer */
        chan_select ^= 0x3;
        output (WR_AM, amsav | chan_select);
    }

}
```

3.10. Example Program: Overlay Mode

`overlay.abs` draws a square in the overlay channel of image memory. One or more bitplanes in the alpha channel can be used for overlay on the TAAC-1. If a bit is set in the overlay channel, after being logically ANDed with the overlay mask, the pixel at that location is displayed using the channel 3 colormaps instead of the video from channels 0-2.

This program does *not* change the Brooktree RAMDAC overlay colors, which must be loaded with black. This is the default state after calling the host initialization routine `ta_init()` or the `tainit` utility.

Items to Note in `overlay.tc`

```
t_set_channel_select (RGBA);
```

Displays all four channels. This selection allows a full-color image in the red, green, and blue channels and a pseudocolor image in the alpha channel. Other combinations are possible, including channel-buffering between pseudocolor images in the red and green channel while using the alpha channel for overlay.

```
t_set_bitmask_mode (TA_ON);
t_set_bitmask (ALPHAMASK);
```

Turns on bitmask mode and set the bitmask for the alpha channel.

```
t_set_overlay_mode (TA_ON);
```

Turns on overlay mode.

```
t_set_overlay_mask (0xff);
```

Sets the overlay mask. This mask is logically ANDed with the alpha channel value. If the result of the AND is non-zero, the channel 3 (alpha) video is displayed instead of the video from channels 0-2. The host initialization routine `ta_init()` turns on all overlay mask bits as a default, but `t_set_overlay_mask()` can be used to disable bitplanes from the overlay operation.

```
t_set_read_mask (TA_ALPHA, 0xff);
```

Sets the alpha channel readmask. The alpha channel value is logically ANDed with the readmask before being applied as an index to the colormap.

```
/*************************************************************************/
/*                                                                       */
/*     Copyright (C) 1988, Sun Microsystems                  */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
* Example program: overlay.tc
*
*      Draw a square in the alpha (overlay) channel
*
*/
#include <taac1/builtin.h>
#define RGBA 0xf                        /* all-channels select*/
#define ALPHA 0x8                       /* alpha channel select */
#define ALPHAMASK 0xff000000            /* bitmask for alpha channel */
#define RED 0x1000000                   /* pseudocolor shade for alpha chan */
main()
{
      static char red[] = {0, 0xff};
      static char green[] = {0, 0};
      static char blue[] = {0, 0};

      /* load pseudocolors into colormap for alpha channel */
      t_set_colormap(TA_ALPHA, 0, 2, red, green, blue);

      /* clear entire screen */
      while (cc(CC_VERT));
      while (!cc(CC_VERT));
      t_erase (0,0,512,512,0);

      /* display all four channels */
      t_set_channel_select (RGBA);

      /* turn on bitmask mode and set bitmask for alpha channel */
      t_set_bitmask_mode (TA_ON);
      t_set_bitmask (ALPHAMASK);

      /* turn on overlay mode */
      t_set_overlay_mode (TA_ON);

      /* set the overlay mask */
      t_set_overlay_mask (0xff);

      /* set the alpha channel readmask */
      t_set_read_mask (TA_ALPHA, 0xff);

      /* draw square */
      t_line (100,100,200,100,RED);
      t_line (200,100,200,200,RED);
      t_line (200,200,100,200,RED);
      t_line (100,200,100,100,RED);
}
```

3.11. Example Program: Blink Mode

blink.abs draws a blinking cross in the alpha (overlay) channel. Only the alpha channel can be blinked.

The blink mask controls which bits of the alpha channel are blinked. If a bit in the blinkmask is a "1" and the corresponding bit in the alpha channel is a "1," then that bit is blinked between 0 and 1. For example, if the blinkmask contained 0xf, then an alpha channel value of 0x18 would alternate between 0x10 and 0x18 as it indexed the alpha channel colormap.

If the alpha channel is being used as an overlay, the overlay will blink between two alpha channel colors (*not* between the overlay color and the underlying channel 0-2 colors).

The Brooktree RAMDAC command register controls the blink rate.

Items to Note in `blink.tc`

```
val = t_get_btcommand(TA_ALPHA);
t_set_btcommand(TA_ALPHA, (val & ~TA_BLINK_MSK) |
    TA_BLINK_FAST);
```

t_get_btcommand returns the current RAMDAC command register state. t_set_btcommand is used to set the blink rate to `fast`. The host initialization routine `ta_init` disables blinking but sets the default blink rate to `TA_BLINK_OCCULT` (75% on, 25% off).

```
t_set_blink_mask(TA_ALPHA, 0x2);
```

Enables blinking in the alpha channel by setting the blink mask. Only the bitplanes with a corresponding "1" in the mask will be blinked.

```
/*********************************************************************/
/*                                                                   */
/*     Copyright (C) 1988, Sun Microsystems                  */
/*                                                                   */
/*********************************************************************/
/*                                                                   */
/*
* Example program: blink.tc
*
*      Draw a blinking cross in the alpha (overlay) channel.
*
*/
#include <taac1/builtin.h>
#define RGBA 0xf                        /* all-channels select*/
#define ALPHA 0x8                       /* alpha channel select */
#define ALPHAMASK 0xff000000            /* bitmask for alpha channel */
int shade, val;
main()
{
      static char red[] = {0, 0xff, 0, 0};
      static char green[] = {0, 0, 0xff, 0};
      static char blue[] = {0, 0, 0, 0xff};

      shade = 0x3000000;                /* pseudocolor for alpha channel */

      /* load pseudocolors into colormap for alpha channel */
      t_set_colormap (TA_ALPHA, 0, 4, red, green, blue);

      /* display all four channels */
      t_set_channel_select (RGBA);

      /* turn on bitmask mode and set bitmask for alpha channel */
      t_set_bitmask_mode (TA_ON);
      t_set_bitmask (ALPHAMASK);

      /* turn on overlay mode */
      t_set_overlay_mode (TA_ON);

      /* set the blink rate */
      val = t_get_btcommand (TA_ALPHA);
      t_set_btcommand (TA_ALPHA, (val & ~TA_BLINK_MSK) | TA_BLINK_FAST);

      /* set the blink mask  to blink bit 1 of alpha (overlay) channel */
      t_set_blink_mask (TA_ALPHA, 0x2);

      /* draw element to be blinked; color will alternate between
         red and blue */
      t_line (175, 200, 225, 200, shade);
      t_line (200, 175, 200, 225, shade);
}
```

3.12. Example Program: TAAC-1 Graphics Library

This section contains an extended example of a program that calls TAAC-1 graphics library functions to draw polygons. `poly` runs on the host. Its functions are:

- initialize TAAC-1
- ask user for input (data filename, type of polygon (wireframe or shaded)and viewing distance)
- read file containing polygon data
- write data to TAAC-1 memory
- load and run the TAAC-1 program

`drawpoly.tc` runs on the TAAC-1, drawing one or more wireframe or shaded polygons, depending on user input. Its functions are:

- read input data from the host
- initialize the state table used by the graphics routines
- set up the transformation matrix and projection elements
- transform all vertices
- for shaded polygons, transform all normals and shade vertices
- for each polygon, clip and project vertices and render polygon

`drawpoly.tc` draws pseudocolor polygons which are double buffered in the red and green channels. The polygons rotate according to increments that are fixed in the program.

Items to Note in `poly.c`

`poly.c` is the main host source file.

```
printf ("Enter input filename\n");
```

Asks for the name of a text file containing polygon data in this format:

number of polygons
for each polygon:

number of vertices
for each vertex:
x, y, z, xnormal, ynormal, znormal

The vertex coordinates and normals are floating-point values; the normals must be already normalized. To make this example as straightforward as possible, the list contains no shared vertices; each polygon's vertices and normals are defined separately.
The example text file, `cube.dat`, describes a cube with coordinates between -0.25 and +0.25. The example program can handle a maximum of 25 polygons and a maximum of 100 vertices total, but this is an arbitrary restriction, not one imposed by the graphics library.

```
stat = ta_set_channel_select (tahndl, ON, OFF, OFF, OFF);
```

This routine sets the video output to pseudocolor from channel 0.

There are four RAMDACs, corresponding to the four video channels, each with its own red, green, and blue colormaps. The red outputs from all four RAMDACs are wired together, as are the green and blue outputs. If you are loading the colormaps for pseudocolor operation, it is a good idea to set the channel select first, to avoid overdriving the monitor. For example, if the colormaps for all four RAMDACs were loaded and indexed to simultaneously drive full intensity (255) out on the red channel, the monitor could be damaged.

```
pseudomap (tahndl);
```

This example subroutine initializes the colormaps for channel 0 (the "red" channel) and channel 1 (the "green" channel) with four pseudocolor ramps. Each ramp has 64 shades. It is not part of the host library.

```
stat = ta_runm (tahndl);
```

Loads the TAAC-1 program that was linked into the host program and begins TAAC-1 execution.

Items to Note in `drawpoly.tc`

`drawpoly.tc` is the main TAAC-1 source file.

```
#include <taac1/t_graphics.h>
```

Include file used by graphics library routines. This file contains function prototypes and includes the file `ta_graphics.h`, containing structure definitions.

`t_scale4x4(), t_translate4x4(), t_rotate4x4(), t_rotate3x3()`

These routines generate transformation matrices for scaling, translation, and rotation. In this example, the z coordinates are scaled by 1/2 and translated so that the z values will be within the clipping boundaries of (0, w). See `t_fastpcliph()` and `t_pcliph()`.

`t_mat4mul(), t_mat3mul()`

`t_mat4mul()` is a library routine used to concatenate two 4 x 4 matrices. `t_mat3mul()` concatenates two 3 x 3 matrices.

```
t_xformh (vmatrix, vertices, xfm_vertices, tot_vert);
t_xformnh (nmatrix, normals, xfm_normals, tot_vert);
```

`t_xformh()` transforms the vertex list of homogeneous [x y z w] coordinates by a 4 x 4 matrix. `t_xformnh()` transforms the normal list of non-homogeneous [x y z] coordinates by a 3 x 3 matrix. The final vertex transformation matrix contains the reciprocal of the view distance in matrix element [2][3], for perspective projection. The routine `t_proj()` will complete the perspective projection by dividing by the transformed w coordinate.

```
t_pseudo (state, (float *)xfm_normals, colors, tot_vert);
```

`t_pseudo()` calculates the pseudocolor value at each vertex, using as input the polygon color from the state table and the transformed normal list. In this example the entire vertex list can be shaded in a single call, since all the polygons are the same color. `t_pseudo()` produces a color list consisting of a single color per vertex. For full-color shading, call t_fastshade(), which produces a color list consisting of three colors (red, green, blue) per vertex. The color values range between 0.0 and 255.0.

t_fastpcliph () and t_pcliph ()

`t_fastpcliph()` clips the vertex list for a single-contour polygon; it does not clip colors; `t_pcliph()` handles multiple-contour polygons and clips colors. The vertices are clipped so that:

-w <= x <= w
-w <= y <= w
 0 <= z <= w

There are also non-homogeneous versions of these clippers, `t_fastpclipnh()` and t_pclipnh(), and an orthographic clipper (`t_porthoclip()`) that clips against arbitrary boundaries. See the graphics library section of the TAAC-1 library chapter for details.

```
t_proj (state, (float *)clipped_vertices],(int*)proj_out,
    num_vert_out, VTX_SIZE);
```

`t_proj()` projects the vertices for a single polygon, using the scale and translate factors set in the projection structure. In this example, `t_proj` scales the transformed x and y coordinates by 255.0 and translates them by +256.0, to place them in screen space. It scales z by 32767.0, to take full advantage of the z-buffer depth (for shaded polygons).

If the input coordinates are homogeneous [x y z w] and `view_dist` in the projection structure is non-zero, `t_proj()` divides x, y, and z by w (before translation) to complete the perspective projection.

`t_proj()` produces a list of projected (screen) coordinates in the format expected by the polygon routines: the lower 16 bits of each coordinate contains the fractional part; the upper 16 bits, a signed integer. This format ensures sub-pixel accuracy.

```
t_wfpoly (state, NUM_CONTOURS, &num_vert_out,
    (int *)proj_out, 0, 0);
t_fastpoly (state, num_vert_out, (int *)proj_out,
    colors_out);
```

`t_wfpoly()` renders a wireframe polygon; `t_fastpoly()`, a flat- or Gouraud-shaded polygon. In this program each polygon has a single contour.

```
/*****************************************************************************/
/*                                                                           */
/*    Copyright (C) 1988, Sun Microsystems                                   */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*
* Example program: poly.c
*
*     Opens TAAC-1, loads and runs TAAC-1 program, prompts for
*     user input, andsends data to TAAC-1 memory
*
*/
#include <taac1/taio.h>
#include <stdio.h>
#include <taac1/ta_graphics.h>
#include "drawpoly_map.h"
#define TAAC_READY 2                        /* TAAC-1 ready for input */
#define GO 1                                /* host tells TAAC-1 to proceed */
#define MAX_VERTS 100                       /* maximum total vertices */
#define MAX_POLYS 25                        /* maximum polygons */
TA_HANDLE *tahndl;                          /* Handle returned from ta_open */
int num_poly;                               /* number of polygons */
struct t_coord4 vertices[MAX_VERTS];        /* input vertex list */
struct t_coord3 normals[MAX_VERTS];         /* input normal list */
int vtxcount[MAX_POLYS];                    /* number of vertices per polygon */

main ()
{

    FILE *inptr;                            /* input file pointer */
    char infil[80];                         /* input filename */
    int ioflag;                             /* flag value  */
    int command[3];                         /* command words:
                                            word 0: polygon type: TA_WIREFRAME
                                                    or TA_GOURAUD
                                            word 1: total number of vertices
                                            word 2: total number of polygons */
    float viewdist;                         /* 0 => orthographic, non-zero =>
                                               perspective view */
    int tot_vert;                           /* total number of vertices */
    int type;                               /* 0 => shaded, 1=> wireframe */
    int stat;                               /* returned status */
    int i,j;
    float *pv, *pn;                         /* ptr to vertex and normal arrays */
    TA_HANDLE *taacstartup ();              /* initialization routine */


    /* open, initialize, load, run TAAC-1 */
    tahndl = taacstartup ();

    /* select channel 0 and load pseudocolor table in red and green channels */
```

```
stat = ta_set_channel_select (tahndl, ON, OFF, OFF, OFF);
pseudomap(tahndl);

/* read in vertex list */
printf("Enter input filename\n");
scanf("%s", infil);
if ((inptr = fopen(infil, "r")) == 0)
    printf("error opening input file\n");

/* read number of polygons */
fscanf(inptr, "%d", &num_poly);

/* for each polygon, read number of vertices, vertex x,y,z, normals
   x,y,z */
tot_vert = 0;                                    /* total vertices */
pv = (float *)vertices;
pn = (float *)normals;
for (i=0; i<num_poly; i++) {
    fscanf(inptr, "%d", &vtxcount[i]);
    for (j=0; j<vtxcount[i]; j++) {
      fscanf(inptr, "%f %f %f", pv, pv+1, pv+2);
      pv += 3;
      *pv++ = 1.;                          /* w = 1. */
      fscanf(inptr, "%f %f %f", pn, pn+1, pn+2);
      pn += 3;
      tot_vert++;
    }
}

/* get type of polygon, wireframe or shaded */
printf("Enter 0 for shaded polygon, 1 for wireframe >");
scanf("%d", &type);
/* get viewing distance (0 = orthographic) */
printf("Enter viewing distance between 0 and 2.(0 for orthographic) >");
scanf("%f", &viewdist);

/* set up command words */
command[0] = type;
command[1] = tot_vert;
command[2] = num_poly;

/* wait for TAAC-1 ready */
ioflag = 1;
while (ioflag != TAAC_READY) {
    if( (stat= ta_read(tahndl,&ioflag,sizeof(ioflag),TC_ioflag))!=
          sizeof(ioflag))
        read_error(stat);
}
/* send vertex list, normal list, and vertex count list
   to TAAC-1 memory */
if((stat= ta_write(tahndl,vertices,sizeof(vertices),TC_vertices))!=
          sizeof(vertices))
```

```
        write_error(stat);
    if((stat= ta_write(tahndl,normals,sizeof(normals),TC_normals))!=
            sizeof(normals))
        write_error(stat);
    if((stat= ta_write(tahndl,vtxcount,sizeof(vtxcount),TC_vtxcount))!=
            sizeof(vtxcount))
        write_error(stat);

   /* send command words to TAAC-1 and set handshaking flag */
    if((stat= ta_write(tahndl,command,sizeof(command),TC_command))!=
            sizeof(command))
        write_error(stat);
    if((stat= ta_write(tahndl,&viewdist,sizeof(viewdist),TC_viewdist))!=
            sizeof(viewdist))
        write_error(stat);

    ioflag = GO;
    if( (stat= ta_write(tahndl,&ioflag,sizeof(ioflag),TC_ioflag))!=
              sizeof(ioflag))
        write_error(stat);

    /* close TAAC-1 link (TAAC-1 program will continue to run) */
    ta_close(tahndl);

}


/********       taacstartup     ********/

/* open TAAC-1 and begin program execution */
TA_HANDLE *taacstartup ()
{
    TA_HANDLE *tahndl;                  /* handle returned from ta_open() */
    int tastatus;                       /* returned status */

    /* open TAAC-1 link */
    if ((tahndl = ta_open (0)) == NULL) {
        fprintf (stderr, "Error opening TAAC-1\n");
        exit (-1);
    }

    if ((tastatus = ta_init (tahndl)) != TA_SUCCESS) {
        fprintf (stderr, "ta_init failed, returned %d\n", tastatus);
        exit (-1);
    }

    if ((tastatus = ta_set_display (tahndl, TA_ON)) != TA_SUCCESS) {
        fprintf (stderr, "ta_set_display failed, returned %d\n", tastatus);
        exit (-1);
    }

    if ((tastatus = ta_runm (tahndl)) != TA_SUCCESS) {
```

```
        fprintf (stderr, "ta_runm failed, returned %d\n", tastatus);
        exit (-1);
    }
    return (tahndl);
}


/********       write_error     ********/


write_error(status)
int status;
{
      fprintf(stderr,"ta_write failed, returned  %d\n",status);
      exit(-1);
}


/********       read_error     ********/

read_error(status)
int status;
{
      fprintf(stderr,"ta_read failed, returned  %d\n",status);
      exit(-1);
}
```

```
/*********************************************************************/
/*                                                                   */
/*     Copyright (C) 1988, Sun Microsystems                  */
/*                                                                   */
/*********************************************************************/
/*                                                                   */
/*
* Example program: drawpoly.tc
*
*      Draws shaded, z-buffered polygons or wireframe polygons
*
*/
#include <taac1/builtin.h>
#include <taac1/t_graphics.h>
#include <taac1/t_math.h>
#define ON 1
#define READY 2                              /* TAAC-1 ready for host input */
#define DONE 0                               /* TAAC-1 has finished */
#define MAX_VERTS 100                        /* maximum total vertices */
#define MAX_POLYS 25                         /* maximum total polygons */
#define MAX_VERTS_PER_POLY 12                /* maximum 12 vertices per polygon */
#define NUM_CONTOURS 1                       /* single-contour polygons */
#define VTX_SIZE 4                           /* homogeneous vertex coords */
#define ZSCALE 0.5                           /* scale z by 1/2 */
#define ZTRANS 0.25                          /* translate z */
#define SIZXY 512                            /* x and y screen size */
#define RED_CHAN 0x1                         /* red channel select */
#define GREEN_CHAN 0x2                       /* green channel select */
#define ZBUFF_CHAN 0xc                       /* blue and alpha channel select
                                                (hardware z-buffer) */

#define BLACK 0xffff0000                     /* black (lower 16 bits), z-buffer
                                                background (upper 16 bits) */

#define RED 63                               /* pseudocolor for cube */

/* Declare global data structures. The host program reads user input
and writes to these items. */
int ioflag;                                  /* host/TAAC handshaking flag */
int command[3];                              /* command words:
                                                word 0: polygon type: TA_WIREFRAME
                                                        or TA_GOURAUD
                                                word 1: total number of vertices
                                                word 2: total number of polygons
                                              */
float viewdist;                              /* perspective viewing distance */
struct t_coord4 vertices[MAX_VERTS];         /* vertex list */
struct t_coord3 normals[MAX_VERTS];          /* normal list */
int vtxcount[MAX_POLYS];                     /* contains number of vertices
                                                in each polygon */

main()
```

```
{
    register int num_poly;                  /* number of polygons */
    register int tot_vert;                  /* total number of vertices */
    register int type;                      /* shaded or wireframe */

    struct t_state_tbl table, *state;       /* state table, ptr to table */

    float trans_matrix[16];                 /* translation matrix */
    float scale_matrix[16];                 /* scale matrix */
    float proj_matrix[16];                  /* projection matrix */
    float vmatrix[16];                      /* vertex transformation matrix */
    struct t_proj_info projection;          /* projection factors */

    struct t_light_source light;            /* light source */
    struct t_light_model model;             /* lighting model */
    void wireframe_poly();                  /* routine for wireframe polygons */
    void shaded_poly();                     /* routine for shaded polygons */


    /* wait for host to set handshaking flag */
    ioflag = READY;
    while (ioflag==READY);

    /* read command word values into registers */
    type = command[0];
    tot_vert = command[1];
    num_poly = command[2];

     /* turn on bitmask mode */
    t_set_bitmask_mode (TA_ON);

   /* initialize state table */
    state = &table;                         /* pointer to state table */
    state->color = RED;                     /* polygon color */
    state->proj = &projection;              /* pointer to project data */
    state->fbzb_mode = TA_PCZB16;           /* hardware z-buffer, pseudocolor */

    /* set scale and translation factors for projection */
    projection.sx = 255.;
    projection.sy = 255.;
    projection.sz = 32767.;
    projection.tx = 256.;
    projection.ty = 256.;
    projection.tz = 0.;

    /* generate matrix to scale and translate z */
    t_scale4x4 (1.0, 1.0, ZSCALE, scale_matrix);
    t_translate4x4 (0.0, 0.0, ZTRANS, trans_matrix);
    t_mat4mul (scale_matrix, trans_matrix, vmatrix);

    /* if this is a perspective projection (viewdist is not zero),
```

```
        generate an identity matrix, set the [2][3] element to 1/viewdist
        and concatenate it with the base matrix. */
    state->proj->view_distance = viewdist;
    if (viewdist != 0.0) {
      t_scale4x4 (1.0, 1.0, 1.0, proj_matrix);
        proj_matrix[11] = t_recp(viewdist);
        t_mat4mul (vmatrix, proj_matrix, vmatrix);
    }

    /* set polygon type */
    state->shade_type = type;                /* shaded or wireframe */

    /* call routine based on polygon type */
    switch (type) {
        case TA_WIREFRAME:                   /* wireframe polygons */
            state->line_type = TA_PLAIN;     /* jagged lines */
            wireframe_poly (state, num_poly, tot_vert, vtxcount, vertices,
              vmatrix);
          break;
      case TA_GOURAUD:
          state->clip_colors = TRUE;         /* clip polygon colors */
          state->ramp_size = 64;             /* size of colormap ramp */
          state->backface_sense = TA_DEFLT;  /* front and backfaces
                                                shaded normally */
          state->diffuse_coeff = 0.99;       /* diffuse lighting coeff. */
          state->lighting = &model;          /* ptr to lighting model */
          model.ambient_weight = 0.01;       /* ambient light weight */
          model.lights = &light;             /* ptr to light src */
          light.direction.x = 0.;            /* light src direction */
          light.direction.y = 0.;
          light.direction.z = -1.0;
          shaded_poly (state, num_poly, tot_vert, vtxcount, vertices,
              normals, vmatrix);
          break;
    }

}


/*******     wireframe_poly       *******/

void wireframe_poly (state, num_poly, tot_vert, vtxcount, vertices,
vmatrix)
struct t_state_tbl *state;                   /* ptr to state table */
register int num_poly;                       /* number of polygons */
register int tot_vert;                       /* total vertices */
int vtxcount[];                              /* array of vertex counts */
struct t_coord4 vertices[];                  /* vertex list */
float vmatrix[];                             /* vertex transformation
                                                matrix */
{
```

```
struct t_coord4 xfm_vertices[MAX_VERTS];
                                        /* transformed vertex list */
  struct t_coord4 clipped_vertices[MAX_VERTS_PER_POLY * 2];
                                        /* clipped polygon vertex list -
                                           twice size of input list */
  int num_vert_out;                     /* number of vertices output
                                           by clipper */
  int proj_out[MAX_VERTS_PER_POLY * 2][3];
                                        /* polygon output from t_proj */
  float rotmatrix[16];                  /* 4x4 rotation matrix */
  static float axis[3] = {1.0, 0.0, 1.0};
                                        /* axis for rotation */
  register int chan_select;             /* channel-select toggle*/
  register int ndx;                     /* index to vertex list */
  register float angle;                 /* rotation angle */
  static int chanmask[] = {0, 0xffff00ff, 0xffffff00};
                                        /* masks for red and green channels*/
  register int amsav;                   /* saved AM register value */
  register int i;                       /* loop counter */

  /* set rotation angle and compute rotation matrix */
  angle = 0.01;
  t_rotate4x4 (axis, angle, rotmatrix);

  /* read AM register and set amsav to enable wordmask mode and
     the z-buffer (blue and alpha channel) wordmask. This value
     will be used later. */
  amsav = input(RD_AM) & ~TA_AMWM_MASK | TA_AMWM | ZBUFF_CHAN;

  /* set channel select to display channel 0 */
  chan_select = RED_CHAN;
  t_set_channel_select (chan_select);

  while (1) {

    /* toggle channel-select and set the wordmask to write
         to the work buffer (non-displayed channel) as well
         as the z-buffer */
    chan_select ^= 0x3;
    output (WR_AM, amsav | chan_select);

    /* erase work buffer during vertical interval. t_erase uses
        the vector ports to erase the screen. If word mask mode
        is ON, the four-bit word mask in the AM register
        determines which channels are erased. A "1" in a channel
        position means that channel will be erased. */
    while (cc(CC_VERT));
    while (!cc(CC_VERT));
    t_erase (0, 0, SIZXY, SIZXY, BLACK);

    /* set bitmask to write to work buffer */
    t_set_bitmask (chanmask[chan_select]);
```

```
        /* premultiply vertex transformation matrix by rotation matrix */
        t_mat4mul (rotmatrix, vmatrix, vmatrix);

        /* transform vertices */
        t_xformh (vmatrix, vertices, xfm_vertices, tot_vert);

        /* clip, project and render each polygon */
        ndx = 0;                                /* start of vertex list */
        for (i=0; i<num_poly; i++) {

            /* clip polygon vertices */
            t_fastpcliph (vtxcount[i], xfm_vertices+ndx, &num_vert_out,
               clipped_vertices);

            /* project clipped vertices */
            t_proj (state, (float *)clipped_vertices, (int *)proj_out,
                num_vert_out, VTX_SIZE);

            /* call polygon routine */
            t_wfpoly (state, NUM_CONTOURS, &num_vert_out, (int *)proj_out,
               0, 0);

            ndx += vtxcount[i];
          }

        /* display channel we have just drawn in */
        t_set_channel_select (chan_select);

    }
}



/*******     shaded_poly       *******/

void shaded_poly (state, num_poly, tot_vert, vtxcount, vertices,
normals, vmatrix)
struct t_state_tbl *state;                  /* ptr to state table */
register int num_poly;                      /* number of polygons */
register int tot_vert;                      /* total vertices */
int vtxcount[];                             /* array of vertex counts */
struct t_coord4 vertices[];                 /* vertex list */
struct t_coord3 normals[];                  /* normals list */
float vmatrix[];                            /* vertex transform. matrix */
{
    struct t_coord4 xfm_vertices[MAX_VERTS];
                                            /* transformed vertex list */
    struct t_coord4 clipped_vertices[MAX_VERTS_PER_POLY * 2];
                                            /* clipped polygon vertex list -
                                               twice size of input list */
    int num_vert_out;                       /* number of vertices output
```

```
                                        by clipper */
int proj_out[MAX_VERTS_PER_POLY * 2][3];
                                      /* polygon output from t_proj */
float rotmatrix[16];                  /* 4x4 rotation matrix for vertices*/
float nrotmatrix[9];                  /* 3x3 rotation matrix for normals */
float nmatrix[9];                     /* normal transformation matrix */
static float axis[3] = {1.0, 0.0, 1.0};
                                      /* axis for rotation */
struct t_coord3 xfm_normals[MAX_VERTS];
                                      /* transformed normal list */
float colors[MAX_VERTS];              /* color list generated by shader
                                        for entire vertex list
                                       (one color per vertex) */
float colors_out[MAX_VERTS_PER_POLY * 2];
                                      /* clipped color list for
                                        single polygon */
register int chan_select;             /* channel-select toggle*/
register int ndx;                     /* index to vertex list */
register float angle;                 /* rotation angle */
static int chanmask[] = {0, 0xffff00ff, 0xffffff00};
                                     /* masks for red and green channels */
register int amsav;                   /* saved AM register value */
register int i;                       /* loop counter */


/* set up initial (identity) matrix for normal transformations */
t_scale3x3 (1.0, 1.0, 1.0, nmatrix);

/* set rotation angle and compute rotation matrices
   for vertices and normals */
angle = 0.03;
t_rotate4x4 (axis, angle, rotmatrix);
t_rotate3x3 (axis, angle, nrotmatrix);

/* read AM register and set amsav to enable wordmask mode and
   the z-buffer (blue and alpha channel) wordmask. This value
   will be used later. */
amsav = input(RD_AM) & ~TA_AMWM_MASK | TA_AMWM | ZBUFF_CHAN;

/* set channel select to display red channel */
chan_select = RED_CHAN;
t_set_channel_select (chan_select);

while (1) {

   /* toggle channel-select and set the wordmask to write
      to the work buffer (non-displayed channel) as well
      as the z-buffer */
  chan_select ^= 0x3;
  output (WR_AM, amsav | chan_select);

  /* erase work buffer during vertical interval. t_erase uses
```

```
            the vector ports to erase the screen. If word mask mode
            is ON, the four-bit word mask in the AM register
            determines which channels are erased. A "1" in a channel
            position means that channel will be erased. */
        while (cc(CC_VERT));
        while (!cc(CC_VERT));
        t_erase (0, 0, SIZXY, SIZXY, 0xffffbfbf);

          /* set bitmask to write to work buffer */
        t_set_bitmask (chanmask[chan_select]);

          /* premultiply vertex matrix by rotation matrix */
        t_mat4mul (rotmatrix, vmatrix, vmatrix);

        /* premultiply normal matrix by normal rotation matrix */
        t_mat3mul (nrotmatrix, nmatrix, nmatrix);

        /* transform vertices */
        t_xformh (vmatrix, vertices, xfm_vertices, tot_vert);

        /* transform normals */
        t_xformnh (nmatrix, normals, xfm_normals, tot_vert);

        /* shade all vertices, using color in state table */
        t_pseudo (state, (float *)xfm_normals, colors, tot_vert);

        /* clip, project and render each polygon */
        ndx = 0;                                /* start of vertex list */
        for (i=0; i<num_poly; i++) {

            /* clip vertices for this polygon */
            t_pcliph (state, NUM_CONTOURS, &vtxcount[i], xfm_vertices+ndx,
               xfm_normals+ndx, colors+ndx, &num_vert_out,
               clipped_vertices, 0, colors_out);

            /* project clipped vertices */
            t_proj (state, (float *)clipped_vertices, (int *)proj_out,
                num_vert_out, VTX_SIZE);

            /* call shaded polygon routine */
            t_fastpoly (state, num_vert_out, (int *)proj_out, colors_out);

            /* increment index to point to vertices for next polygon */
            ndx += vtxcount[i];
        }

        /* display channel we have just drawn in */
        t_set_channel_select (chan_select);
    }
}
```

4

The C Compiler (tacc)

4

The C Compiler (tacc)

4.1. Introduction

The TAAC-1 compiler `tacc` is an ANSI-standard C compiler designed to take advantage of the TAAC-1 architecture. Where possible, `tacc` combines address calculations, memory access, arithmetic operations, and branch instructions within the TAAC-1's wide instruction word.

TAAC-1 Extensions

The TAAC-1 C compiler supports certain extensions to standard C for direct access to TAAC-1 capabilities. These extensions include:

1. Built-in functions: A set of macros built into the compiler permit fast access to specialized TAAC-1 functions, such as lookup tables access and 1D/2D/3D data reads/writes, without the overhead of subroutine calls.
2. Global register variables.
3. Register binding: This extension binds register variables to specific registers in ALUs RC and RD.
4. In-line code: The compiler allows you to insert in-line assembly language within a C program and to access program variables symbolically from the assembler code. One use of in-line code is to replace program inner loops for speed enhancement.
5. Function prototyping: As defined by the proposed ANSI standard, function prototyping allows function declarations to contain information about function arguments. Using function prototyping can prevent floats from being converted to doubles when being passed as function arguments.
6. `fast` functions: Usually, the compiler automatically saves and restores all registers used within each subroutine. However, eight registers from each ALU (RC and RD) are allocated for `fast` functions. These registers are not saved or restored, minimizing the overhead of calling these subroutines. In addition, function

prototyping allows arguments to `fast` functions to be passed in registers, rather than on the stack.

7. `loop/switchf` statements: These non-standard statements are additions to the standard C compiler. loop uses the sequencer counter to control looping, to improve code efficiency. `switchf` is an optimized form of the standard `switch` statement.

Other Considerations

While the compiler is a useful tool for programming the TAAC-1, porting programs from the workstation to the TAAC-1 often involves more than new compile and link processes. These issues need to be taken into consideration:

Memory

The TAAC-1 is a physical address machine whose non-contiguous memory address space is composed of functionally different memory types. *Program memory* consists of 16K 200-bit words; the TAAC-1 startup and register-dump subroutines occupy approximately 1Kword of this space.

There are two kinds of *data memory* - dynamic RAM and scratchpad RAM. *Dynamic RAM* (DRAM, or data/image memory) is used for both image display (the video frame buffer) and data storage. DRAM can be addressed in 1D, 2D, or 3D modes. The vector ports provide fast serial DRAM access. There is 8Mbytes or 2M 32-bit words of contiguous DRAM memory, which equates to 1024 x 2048 x 32 bits in 2D image space.

The `talink` linker does not automatically allocate any space for DRAM variables. If you declare any DRAM variables, you must specify a DRAM address range at link time that does not conflict with your usage of DRAM memory for image display. See the `talink` chapter for further information.

Be aware that all writes to DRAM variables are affected by the current setting of the DRAM mode register. The mode register

- enables/disables bitmask mode
- selects the current bitmask id
- enables/disables word mask mode
- specifies the current word mask

- selects 1D/2D/3D mode for AC register reads and writes
- selects random mode, serial read mode, serial write mode, or shift register load (the last three modes apply to vector port operations).

Therefore, if you have enabled bitmask mode, the current bitmask will apply to all writes to DRAM variables via the AC or AI register. If you have enabled word mask mode, the current wordmask will apply. In general, bitmask mode and wordmask mode should be disabled for writes to program variables stored in DRAM.

Scratchpad RAM (SRAM) holds the C stack. In addition, it is the default location for global and static program variables. SRAM size is much smaller than DRAM - 16K 32-bit words. The C stack grows from high to low SRAM memory, while global and static variables are allocated space at the beginning of SRAM. There is no check for collision of C-stack and program variables.

Data types

All memory is word-addressable, as opposed to byte or halfword addressability. Consequently, the smallest data size is 32 bits, including int, short, and char data types.

UNIX Support

The TAAC-1 library includes graphics primitives, image processing subroutines, mathematical functions, and video control subroutines. The TAAC-1 does not run UNIX, however, and the library does not support the full range of functions contained in a standard UNIX interface. For example, the TAAC-1 does not currently support memory allocation, string-handling, or I/O functions.

Error Detection

For efficiency considerations, arithmetic overflow/underflow errors, out-of-bounds addresses, and stack overflow/underflow, errors go undetected and produce undefined results.

Deviations from Standard C

The TAAC-1 compiler contains these deviations from standard C:

- Bit fields are not supported.
- Double precision is used to evaluate expressions only if one of the operands is `double`.

- Floating point constants are assumed to be of type `double` unless the subroutine is compiled using the `-fsingle` option, or unless the value is followed by `F` or `f` or the constant is cast as a `float`, as in these examples:

```
#define ABC 100.7F
```

or:

```
#define ABC (float)100.7
```

- External variables may be referenced more than once, using the `extern` keyword. External variables may be defined only once, by specifying them *without* the `extern` keyword.

- Scalar global variables are always initialized to zero, unless the program specifies another initial value. Global arrays and structures are not initialized unless done so explicitly by the program.

NOTE: functions returning structures are now supported.

The examples in the TAAC-1 programming chapter and the distribution demos offer further help in TAAC-1 programming.

4.2. `tacc` Command Syntax

```
tacc [options] filename
```

`-c`	Use source code as comments in the assembly code output file.
`-p`	Disable the TAAC-1 C preprocessor before compilation. Default enables the preprocessor.
`-w`	Suppress warning messages.
`-a`	Do not invoke the assembler.
`-l`	Tell assembler, if invoked, to generate a listing file.
`-fsingle`	Use single precision to pass arguments or perform computations involving variables of type `float`. Does not affect `doubles`.

`-Istring`	Tell the preprocessor to use the `string` directory to search for include files.
`-Dname`	Define a constant `name` containing a value of 1.
`-Dname=stuff`	Define a string constant `name` containing the string `stuff`.
`-Uname`	Tell preprocessor to remove any initial definition of `name`.
`-id`	Print out compiler version number.
`-peep`	Disable peephole optimization. Used mainly in compiler error isolation. Not normally used.
`-pack`	Disable microcode compaction. Not normally used.
`-stdstr`	Do not pack strings.

The compiler generates an assembly code source file, `filename.asm`. Unless you have specified the `-a` option, the compiler automatically invokes the assembler, which generates an object file, `filename.obj`.

tcc driver

The TAAC-1 software also includes a driver for the compiler, linker, and taabs2o, called `tcc`. `tcc` syntax resembles standard Unix cc syntax in many ways. It has the following options:

`-DX`	Define cpp symbol X (for cpp)
`-IX`	Add directory X to cpp include path (for cpp)
`-UX`	Delete initial defintion of cpp symbol X (for cpp)
`-fsingle`	Use single precision to pass arguments or perform computations involving variables of type `float`. Does not affect `doubles`.
`-peep`	Do not do peephole optimization
`-pack`	Do not do compaction
`-stdstr`	Standard strings (do not pack strings)
`-comment`	Include source text as comments
`-list`	Generate listing file
`-g`	Generate symbolic debugging information

-d <start> <end>	select dynamic ram to use for DRAM variables
-lX	Read object library (for taload)
-LX	Add directory X to ld library path (for taload)
-m <file.map>	generate map file in file.map
-h <file.h>	generate header file in file.h
-n <symbol>	supply symbol to taabs2o
-o <file{,.o,.abs}>	Set name of output file
-c	Produce '.obj' file. Do not run linker
-v	Report which programs the driver invokes
-E	Run source thru cpp, output to stdout
-S	Produce '.asm' file. Do not run tasm
-dryrun	Show but do not execute the cmds constructed by the driver
-help	print this message

For example, given a TAAC-1 program myprog.tc:

```
% tcc -o myprog.abs myprog.tc -ltaac1
```

would invoke `cpp` (the preprocessor), `tacc`, and `talink` to produce the absolute file myprog.abs.

```
% tcc -o myprog.o myprog.tc -ltaac1
```

would invoke `cpp`, `tacc`, `talink`, and `taabs2o` to produce an object file myprog.o that could be linked with a host program.

```
% tcc -o myprog myprog.tc -ltaac1
```

would invoke `cpp`, `tacc`, and `talink`, and then convert the absolute file to a Unix executable called myprog, which you could then run by typing:

```
% myprog
```

The first line of myprog would invoke the TAAC-1 utility `tasvc`, which would act like tarun in downloading and executing the TAAC-1

program. For a further description of `tasvc`, refer to Appendix A of this manual.

4.3. Programming Preliminaries

The `tacc` C compiler uses the standard Sun C preprocessor, allowing the use of all standard preprocessor functions, including `#define`, `#include`, and macros with arguments.

Programs use sets of intermixed variable and function definitions. TAAC-1 C uses the standard C syntax for declarations, allowing you to define pointers to arbitrary objects, arrays of arbitrary objects, or functions returning arbitrary objects. A variable must be defined before its first use. Global variables must be defined before the first function definition.

Names

Name length is unspecified, but only the first 32 characters are significant. Case distinctions are respected. The first character must be alphabetic or an underscore; the rest of the name can be alphanumeric or an underscore.

Constants

Constants can be expressed using decimal, octal, hex, character, string, or floating point notation. *Decimal constants* are any string of digits. *Octal constants* begin with the number `0`. *Hex constants* begin with `0x` and may contain digits and the letters `a` - `f`, in upper or lower case. For example, `0xffff`, `0Xf000`, and `0x001` are all hex constants. *Character constants* consist of single characters enclosed in single quotes. *String constants* consist of text strings enclosed in unescaped double qoutes. The compiler always appends a zero byte to the end of any string constant. The value of a string constant is the address of the first byte of the string.

Applying the suffix `l` or `L` to a decimal, octal, or hex constant identifies the constant as long. `U` or `u` indicates an unsigned constant.

The compiler supports these standard C escape sequences:

`\a`	alert (bell)	`\f`	form feed
`\n`	newline (line feed)	`\'`	single quote
`\t`	horizontal tab	`\"`	double quotes
`\v`	vertical tab	`\nnn`	nnn = 1-3 digit octal number
`\b`	back space	`\xnnn`	nnn = 1-3 digit hex
`\r`	carriage return	`\\`	backslash

If a backslash is followed by a newline character, both characters are thrown away. This allows you to spread a string constant over more than one line. If a backslash is followed by any other character which is not in the list above, then the backslash is thrown away. String constants contain a maximum length of 200 characters.

A floating-point constant consists of an integer part, followed by a decimal point, followed by a fractional part and/or an exponent part. Both the integer and fractional part consist of a string of decimal digits. Either the integer or fractional parts may be missing, but not both. Examples:

3.14159
39.
.125

The exponent part consists of the letter E or e, followed by an optional sign, followed by a string of digits. The exponent represents the power of ten by which the floating-point constant (mantissa) preceding the letter e is multiplied. Examples:

3.45e-2
0.0345

The compiler assumes by default that all floating point constants are of type `double` unless you:

- Invoke the `-fsingle` option when compiling
- Add the letter `f` or `F` to the end of constants, as in:
  ```
  #define abc 4.13F
  ```
- Cast the constant as a `float`, as in:
  ```
  #define abc (float)4.13
  ```

If you wish floating point expressions consisting of `float` operands and floating point constants to be carried out in single precision, you must select one of the options above. Otherwise the non-constants will be promoted to `double` and the operation will be carried out in double precision.

Comments begin with `/*` and end with the first `*/` following on the input stream. This precludes the use of comments within comments.

4.4. Expressions

Unsigned operations are the same as signed operations, with the exceptions of right shifts and comparisons. Left shifts are logical shifts (zeroes are shifted in). The right shift is a logical shift if the left operand is unsigned (zeroes shifted in). Otherwise, the right shift is arithmetic (negative numbers are sign-extended).

NOTE: Shift operations are faster if the shift amount is specified with a constant rather than a variable, and in all cases left shifts are faster than right shifts.

Operator Hierarchy

The compiler uses the following hierarchy of operators:

```
,
= += -= /= *= %= <<= >>= &= |= ^=
?: (Conditional)
||
&&
|
^
&
== !=
n
```

```
< <= > >=
<< >>
+ -
/ * %
unary ++ -- - ~ ! & * sizeof (cast)(expr)
-> . function calls subscripting
(expr) name constant
```

The comparison operators generate a zero if false or a one if true.

Expressions Involving Structures

Structure assignment is supported, including the passing of structures by value to functions. You can also take the address or select a member (using the "." operator) of a structure or union identifier. *This compiler now supports functions returning structures.*

Single and Double Precision

In standard C, all floating-point operations must use double precision. In TAAC-1 C, double precision is used only if one or both of the operands are double precision.

Error Detection

In general, errors such as arithmetic overflow or out-of-bounds addresses go undetected and have undefined results.

4.5. Variables

TAAC-1 C supports these storage classes:

```
auto
extern
static
register
typedef
```

Variables declared `register` go into a hardware register. In TAAC-1 C, global registers are permitted. Global register variables are always kept in their assigned registers and are never saved or restored.

Two storage class modifiers, `RC` and `RD`, allow you to specify the ALU whose register you will use for storage. Storage class modifiers appear anywhere storage classes appear, either in addition to or as a replacement for the storage class. Since `RC` and `RD` apply only to `register`, you must always use `register RD` or `register RC`. When storage class modifiers are unspecified, the TAAC-1 compiler looks at

the type of variable assigned to the register. Pointer variables go into the RD ALU; the rest go into the RC ALU.

The number of register variables has no enforced limit, but keep in mind that eight RC and RD registers are reserved for `fast` functions (see the `fast` function section), three are reserved in RC for the debugger and scratch use , four in RD for the debugger, C stack pointer, and scratch use, and at least one or two more registers may be needed to hold intermediate results of complicated expressions.

Other storage class notes:

- The compiler never assigns structures or union types to registers.
- The default storage class in declarations within functions is `auto`.
- The default storage class in external declarations is `extern`.
- External variables may be referenced more than once, using the `extern` keyword. External variables may be defined only once, by specifying them *without* the `extern` keyword.

DRAM Variables

The scratchpad memory is currently sized at 16Kwords. For data which takes up more than 16Kwords minus the stack size, the compiler provides the DRAM storage class modifier to indicate that dynamic RAM will be used to store the data. This modifier may be used in addition to or in place of the storage class. For instance:

```
DRAM int x[100];
static DRAM y[1000];
DRAM float z[200][200];
```

The extent of memory used is determined by the DRAM range specified when invoking the linker.

Data Types

The compiler supports these data types:

```
char                float
short               double
int                 struct
long                union
unsigned char       enum
unsigned short      fast
unsigned int
unsigned long
```

The types `char`, `short`, `int`, `enum`, and `long` all refer to 32-bit signed integers. Similarly, `unsigned char`, `unsigned short`, `unsigned int`, and `unsigned long` all refer to 32-bit unsigned integers.

The compiler does not support bit fields.

Optimizing Data Structure Definitions

The way you define your data structures has a bearing on the efficiency of your compiled programs. In the case of array subscripting, C syntax requires that the subscript be multiplied by the size of the object before the subscript is added to the pointer of the object array. If the size of the object is not one, the compiler generates a multiplication. If the size of the object is a power of two, a shift can replace the multiplication for a faster result. This means that it is computationally expensive to use an array of an array, structure, or union.

For instance, assume that your program declared:

```
struct{
   int x;
   int y;
   int z;
} points[10];
```

The compiler would have to convert the array of structure reference:

```
points[i].y = 2;
```

into:

```
*(&points + (i*3) + 1) = 2;
```

which requires a multiplication. On the other hand, suppose your program declared:

```
int xpts[10];
int ypts[10];
int zpts[10];
```

You could code:

```
xpts[i] = 2;
```

which the compiler would convert to:

```
*(&xpts + i*1) = 2;
```

Since the compiler recognizes the multiplication by one , no multiplication is required and access is faster.

Machine-Dependent Extensions

To provide extra efficiency, TAAC-1 C supports some non-standard extensions to the C language syntax.

WARNING: Using these extensions makes your code incompatible with the standard C compiler.

When declaring register variables, TAAC-1 C supports an optional, non-standard method for binding variables to registers. To bind a variable to a register, use an `@` followed by a constant expression of a register number after the variable. The constant must be in the range from 0 to 127. Register numbers 0 - 63 are in ALU RC. Register numbers 64 - 127 are in ALU RD.

As an example of variable binding, if you declared:

```
register cnt @8;
```

the variable `cnt` would be assigned to RC8. To avoid having the compiler allocate registers which you intend to specify explicitly, *always* declare registers with specific bindings *before* declaring unbound registers.

To prevent the compiler from allocating certain registers, use the non-standard `reserve` declaration, consisting of the non-standard keyword `reserve`, followed by a list of one or more constant expressions separated by commas and terminated by a semi-colon. Each constant expression must resolve to an integer in the inclusive range from 0 - 127. For example:

```
reserve 1, 2, 3;      /* RC register 1-3 */
reserve 65;           /* RD register 1 */
```

prevents RC registers 1, 2, 3, and RD register 1 from being allocated by the compiler over the scope of the declaration.

4.6. Initialization

Variables may be initialized when they are declared, according to standard C rules. Initialization of automatic or register variables is done as the program is running. Initialization of external or static

variables is done during program load. Scalar global variables are always initialized to zero, unless the program specifies another initial value. Global arrays and structures are not initialized unless done so explicitly by the program.

For reasons of expediency, string constants are handled using a semantically incorrect syntax. Since `char` data types are sized as 32 bits, you might expect string constants to use one string byte per 32-bit word. Actually, string constants are always packed four bytes per word.

This non-standard way of handling strings was chosen on the premise that strings will *not* normally be processed as arrays of characters by the TAAC-1, but will instead be passed to the host.

Other compiler changes provide a consistent method of handling strings. If you initialize an array of `char` with a string, the bytes will again be packed four bytes per word, unless the `-stdstr` option is invoked. For instance:

```
char s[] = "abcdef";
```

reserves two words of storage, the number of 32-bit words required to hold the 6-byte string, and trailing null. If you plan on actually processsing the string using array subscripting, you must initialize the string using a series of character constants:

```
char s[7] = "abcdef";
```

This example reserves seven words of storage, but the first two words contain the packed string and the remaining five words contain zeroes. To reserve ten words of storage and put one byte of the string in each word, use:

```
char s[10] = { 'a', 'b', 'c', 'd', 'e', 'f', ' '};
```

If you assign a string constant to a pointer, as in:

```
p = "abcdef";
```

then the pointer points to the first of two 32-bit words containing the string constant.

4.7. Function Definitions

Functions can be defined in any order. Functions returning structures are not supported.

`fast` Functions

The compiler reserves a group of eight registers in each ALU for functions declared as `fast`. The reserved registers can then be assigned as required. Because the `fast` registers are never saved or restored, function entry and exit requires almost no overhead. If more than eight registers per ALU are required, other available registers will be assigned as needed. In this case, routine overhead is increased only by the number of registers over eight that must be saved and restored.

Overhead will be further reduced if all non-register variables are stored in static memory , since stack frame allocations/deallocations will not be necessary on function entry/exit.

Do not directly or indirectly call one `fast` function from another, since the `fast` registers are effectively global. `fast` functions apply best to frequently called low-level functions requiring fair amounts of local register storage.

Although `fast` functions minimize overhead, they may not always be necessary. Low level functions using global registers may not need to be fast if the expressions within it are simple enough to not require any extra register assignments. A look at the generated code will help to determine if a function is a good `fast` candidate.

The `fast` data type is only significant when applied to a function declaration. The word `fast` should immediately precede the function name. For example:

```
void fast funca ()
```

or

```
int * fast funcb ()
```

The `STACK_PC` Storage Class Modifier

When a subroutine is called, the jump-to-subroutine instruction pushes the return address onto the stack in the sequencer. The return address is left there until the function returns, at which point it is popped off the stack again. Although this scheme allows very efficient

function calls, it also restricts the maximum combined function call depth and loop statement nesting to the size of the hardware sequencer stack (65).

If you need recursion to arbitrary depths without worrying about overflowing the sequencer stack and you are prepared to pay a penalty of up to seven instructions of extra overhead, then you can declare all functions in a recursive call chain with the storage class modifier STACK_PC. This tells the compiler to pop the return address off the sequencer stack and save it in the software stack, on entry, and to return to the address saved in the software stack, on exit. Here is an example:

```
STACK_PC factorial(n)
{
        if (n==1)
             return 1;
        else
             return n*factorial(n-1);
}
```

At the point of a function call, there is no distinction made between functions declared with STACK_PC and those declared without it. The effect of STACK_PC is completely internal to the function to which it is applied.

NOTE: Using STACK_PC does nothing to prevent software stack overflow. Any function which is to be given the STACK_PC modifier should be as sparing as possible in its declaration of "auto" variables.

Function Prototypes

The TAAC-1 C compiler supports the new ANSI function prototypes. Since these are very new to C language users, explanations may be helpful. The ANSI standard allows you to set up function arguments either the old way, without prototypes, or the new way, with prototypes.

In the old way of declaring functions, the number and type of arguments are known only at the time the function body is declared. A forward or external reference to the function contained no information about the arguments.

In the new way of declaring functions, every declaration of the function must contain information concerning the number and types of arguments, and the information must be the same in every declaration. Every time a function is declared in the new way, you must supply a

list of argument declarations, using the usual syntax for declarators. The declarator may include an identifier, known as a direct declarator, or it may not, which is known as an abstract declarator. The identifier is required when declaring a function body and optional when making a forward or external reference.

There are special rules for functions with no arguments and for functions with a variable number of arguments. A function with no arguments must always be declared with a single `void` argument. A function with variable arguments must always be declared with its last argument reading "`...`". The last argument may be preceded by a list of required argument types.

Any function may be declared using either the old way or the new way, but all declarations of the function must use the same way. The next table provides examples.

When you pass an argument to a function, the compiler checks to see if there is a prototype in scope containing information about the argument. If the compiler finds information about the argument, the compiler converts the actual argument to the type of the prototype argument or else issues an error message if the conversion would violate any of the type conversion rules. If there is no information about the argument, either because no prototype is in scope or because the argument is on the variable part of the list, the compiler must assume that the type of the actual argument is what the callee expects.

For `fast` functions only, if you use function prototyping and the function expects an argument to be a register variable, the compiler passes that argument in the correct register. This applies to function pointers as well, so long as you declare the fast function pointer in the same way as the function it points to.

Table 4-1 *Old and New Ways of Declaring Functions*

Argument Type	*Old Way of Declaring*	*New Way of Declaring*

None	`func();`
	
	`func()`
	`{`
	`}`
		`func(void);`
		
		`func(void)`
		`{`
		`}`

Fixed	`float func();`
	
	`float`
	`func(a, b, c)`
	`int a;`
	`float b;`
	`char *c;`
	`{`
	`}`
		`float func(int, float, char *);`
		
		`float func( int a, float b, char *c )`
		
		`{`
		`}`

Variable	`error();`
	
	`error(s,a1,a2,a3,a4)`
	`char *s;{`
	`}`
		`error(char *s,...);`
		
		`error( char *s, ... )`
		`{`
		`}`

Converting from Float to Double

In the absence of specific prototype information, a `float` formal argument is forced to `double` and `float` actual arguments are converted to `double`. If you want to pass a `float` argument as a `float` and avoid the overhead of conversion to `double`, use the new way of function declaration. Another way to avoid float-to-double conversion is to use the `-fsingle` option when compiling. This option ensures that double precision will not be used anywhere in the module. Non-standard extensions of declaration syntax are described in the next section. These new extensions support the naming of specific registers.

Declaring Global Registers

With separate compilation, you can declare global registers one way for one group of functions and a different way for a different set of functions. If two separately compiled functions are to refer to the same

set of global registers, both source files must have global register declarations which match exactly in number, type, and *order*. This prevents the problem of calling functions, only to have them unexpectedly and wrongly use the caller's global register as a local register.

Function Calls and Argument Size

The default size of a function argument is that of an `int`, which is 32 bits. If a formal argument is a 64-bit quantity and the actual argument is not, and if there is no function prototype in scope defning the type of the argument, then you must make sure to cast the type of the actual argument so that the stack is set up properly for the called function. For instance, suppose you called a function declared as:

```
func( x )
   double x;
{
   ....
}
```

If the actual argument is an `int` or a `float`, the calls should look like this:

```
int x;
func( (double) x );
func( (double) 3 );
```

A better way is to use a function prototype:

```
void function ( double );          /* prototyping */
func ( 3);                         /* call */
func( double x)                    /* actual declaration */
{
   ...
}
```

4.8. Statements

The following are valid statements. Keywords are shown in boldface. Punctuation is required as shown.

expression;
if(*expression*) *statement*
if (*expression*) *statement* **else** *statement*
while (*expression*) *statement*
do *statement* **while** (*expression*);
for (*expression1; expression2; expression3*)

```
   statement
break;
goto label;
continue;
return;
return expression;
case const-expr:
default:
switch( expression ) statement
switchf( expression ) statement
loop ( expression ) statement
{ statement1 statement2 ... statementn }
```

Where:

switch Generates a jump table for a set of cases if the maximum case minus the minimum case plus one divided by the number of cases is less than 1.25. Otherwise, the compiler simply arranges to emit code which tests for each possible value, in the order of appearance. The jump table `switch` is more efficient in terms of space and time.

switchf Same as `switch` except that if the compiler generates a jump table it does not generate code to check the bounds of the jumped value. The compiler assumes the value of `switchf` will invariably select one of the cases and the default will never be selected. Make sure the value you switch on is in range.

loop Not a standard C statement, `loop` allows you to save code space in tight loops by using the sequencer counter to control looping. The statement in the example loops according to the number in the expression. The result of the expression is treated an an unsigned integer, with zero being interpreted as 65536.

If an expression value of zero is to cause no looping, the `loop` statement must be prefixed with an `if` statement. If a count larger than 65535 (0xffff) is given, the actual count executed will be the given count modulo 65536 (i.e., expression & 0xffff).

A `continue` statement inside a `loop` behaves as if the last statement in the loop has just been

executed. In other words, the compiler decrements and tests the count, then either branches to the top of the loop or continues, as indicated by the test.

The total loop nesting depth and function call nesting depth may not exceed the size of the sequencer stack (65).

4.9. Special Coding Techniques

Careful coding in TAAC-1 C comes fairly close to the efficiency of hand-coding. *The best way to reduce code size and improve running time is to use lots of register variables.* However, the overhead of saving and restoring registers may sometimes outweigh the benefit of fast variable access. Using `fast` functions for low-level subroutines often eliminates the overhead of preserving and restoring register states.

Another way of saving space and time is to keep information in registers available to all functions. These global registers need not be saved or restored. Using global registers minimizes the overhead of entering and exiting low-level functions.

Since the compiler does not perform global flow analysis, try to identify frequently used variables and place them in registers. Either declare the variables as register variables or assign them to explicitly declared local register variables in the scopes where the variables are used intensively.

Many programs implement a set of operations requested by a host processor in some sequence. Each operation typically has a small set of frequently used variables, such as the current pixel address. Declare these often-used items as register variables when you have operations implemented completely in one function.

More often than not, a typical operation is implemented using several functions. Multi-function operations work best when a set of global variables are stored in registers available to all the functions in an operation. Since the number of variables you can place in registers is limited, each operation usually cannot have its own set of global register variables. In this case, it is often expedient to declare many global registers and use them for different purposes in different functions.

If code space gets short, try as much as possible to *pass function parameters in global variables,* rather than as function arguments.

Since the compiler does not attempt to identify common sub-expressions and arrange for one-time computing, you should try to do this yourself. If you see a reference to a memory variable, a structure, or an array repeated more than once in a short segment of code, declare a local register variable, evaluate the common expression into it, and use the register in place of the expression.

4.10. Switching Between Standard C and TAAC-1 C

To use UNIX debugging facilities, compile TAAC-1 C programs in the standard C compiler, load the program into a library emulating the built-in functions, and run the program in UNIX.

To switch between C and TAAC-1 C easily, first declare any global registers:

```
globreg int x, y;
```

If using nested `loop` statements, suffix the word `loop` with the level of nesting.

If compiling with the regular C compiler, use the following directives:

```
#define globreg
#define fast
#define RD
#define RC
#define DRAM
#define switchf switch
#define loop(x) for(_ii=0;_ii<x;++_ii)
#define loop0(x) for(_i0=0;_i0<x;++_i0)
#define loop1(x) for(_i1=0;_i1<x;++_i1)
#define loop2(x) for(_i2=0;_i2<x;++_i2)
short _ii,_i0,_i1,_i2;
```

To run TAAC-1 C programs under UNIX, provide a library defining UNIX-based equivalents of each built-in function. For TAAC-1 C, use:

```
#define globreg register
```

4.11. Run-Time Notes

All TAAC-1 C programs start by initializing the AM register, the MO register, and the stack pointer, and jumping to the function `main`, which behaves as if it had been called with no arguments. The compiler translates function returns from `main` into hang instructions, stopping

the functions. The compiler jumps to `main` instead of calling it to save another level of function call.

Function Calls

Always call a function with the number of arguments declared for it. *Maximum function call depth is set by the sequencer stack depth of 65.* Functions which have the `STACK_PC` attribute are not included.

Register Usage

The compiler pre-allocates these registers:

RC63	reserved for the debugger
RC62	scratch (use any time) and function return LSBs
RC61	scratch and function return MSBs
RD63	reserved for the debugger
RD62	reserved for the stack pointer
RD61	scratch
RD60	scratch

During expression evaluation, the compiler requires register placements of all intermediate values. That is to say, no temporary storage of intermediate results in memory takes place - only temporary storage in registers is used. One or two registers are typically needed for intermediate results in complex integer operations. When compiling simple expressions, you can run out of register space if you declare too many register variables. If the compiler runs out of registers while compiling code for an expression, it displays the message "No more registers available." You should either try to simplify the expression or else reduce the number of register variables in use.

Non-Register Variables

The linker supports up to eight different memory types. The compiler uses three of the memory types:

Type 0	program memory
Type 1	Data in scratchpad memory (SRAM)
Type 2	Data in data/image memory (DRAM)

All non-register variables are allocated in fast scratchpad memory. Static and global variables are allocated starting at the low address of scratchpad memory. Function arguments and other automatic variables are allocated on the C stack, which starts at the highest address in scratchpad memory and grows toward zero. Static and global variables can also go into data/image memory (DRAM storage class).

C Stack Format

The stack pointer (SP) starts at the highest address in scratchpad memory and moves towards zero. The stack pointer always points to the last used location on the stack.

On function entry, the compiler creates a new stack pointer by computing:

```
SP = SP - framesize
```

Where `framesize` is the space required for all automatic variables.

To call a function, the compiler pushes arguments onto the C stack, decrementing SP by the argument size. For example, if your program called a function consisting of two one-word arguments, the compiler would produce code to do this:

```
SP = SP - 1
*SP = arg2
SP = SP - 1
*SP = arg1
jump subroutine to function
SP += 2;
```

The jump subroutine instruction pushes the current program counter onto a stack in the sequencer.

On function exit, the compiler restores the stack pointer by computing:

```
SP += framesize
return
```

The return instruction sets the new program counter by popping the sequencer stack.

Local automatic variables are located using positive offsets from the SP. Because argument pushes affect the stack pointer, the compiler keeps track of the total unumber of words pushed and automatically adds the current push size to the offset. The addess of automatic variable `x` is:

```
SP + offset(x) + pushsize
```

Function arguments are located beyond the end of the current stack frame. If `x` is the second argument to a function and the first argument is one word, then the second argument's address is:

```
SP + 1 + pushsize + framesize
```

Where `1` is the offset to `x`. The first argument has an offset of zero.

C Stack Overflow

The run-time stack is set up during run-time initialization so that it starts at the highest address of scratchpad memory and grows "backwards" towards zero. External storage starts at zero (relative to the scratchpad memory start address of 0x30000000) and is allocated towards the highest addresses. This leaves open the possibility that the stack can overflow into the area occupied by external storage. Since this condition is never checked for, it is up to you to keep track of storage usage, to prevent stack overflow. The way to minimize stack usage is to avoid declaring large arrays or structures inside functions unless you first make them `static`.

The Function `atof()`

The compiler uses the C library routine `atof()` for conversion of floating point constants to binary. The function `atof()` does not behave well when given strange inputs such as out-of-range exponents. If you get a strange fault during your compile, it probably occurred when `atof()` was trying to process a malformed floating point number.

4.12. In-Line Assembler Code

You can insert in-line assembler code within C routines by placing assembler code between the escape sequences `/$` and `$/`. The compiler copies all data following `/$` into the assembler output file, until `$/` appears and ends the assembler inclusion. Assembly language entries can occupy single lines:

```
/$ assembler code $/
```

or multiple lines:

```
/$
assembler line 1
assembler line 2
$/
```

Normally, in-line assembler code is used inside the body of a C function. When doing so, you can reference C variables from within the assembler code by using:

```
@name
```

where `name` is the variable name. If `name` identifies a register variable, `@name` will be replaced by a register number assigned by the compiler. For instance, an included assembler line that moves register variable `alpha` to register variable `beta` looks like this:

```
/$ rc_pr rc_a#@alpha rc_c#@beta rc_fyout $/
```

This use of register names is complicated by the fact that the choice of fields in which a register can appear depends on the ALU in which the register resides. When writing in-line code, register locations must be known in advance. Fortunately, the compiler allows you to specify an ALU (RC or RD) during register declaration.

If `name` applies to a non-register variable, the compiler replaces `@name` with an expression used to represent register addresses, as indicated in the next table.

The compiler automatically prefixes variable names with an underscore to avoid conflict with assembler mnemonics.

In-Line Code Hints

On entry to an in-line assembler code section, the compiler guarantees that it will not have any temporary integer ALU registers active.

The MQ register in each ALU and the four scratch registers (RC61, RC62, RD60, RD61) can be used at any time. However, be aware that a number of TAAC-1 library routines (written in assembly language) also use these scratch registers, without saving or restoring them.

The AM and MO registers must be explicitly saved and restored if they are to be modified.

Table 4-2 *Replacing Non-Register Variables in Assembly Code*

Storage Class	`@name` *Replaced By*
external	`_name`
static	`_SSnn`, where `nn` is a compiler-generated number
parameter	`CCnn+k`, where `k` is a constant, `CCnn` is a symbolic constant equated to the "framesize" of the current function (see the section on framesize)
automatic	`k`, a constant

If the stack pointer must be adjusted, then it must be restored before exit. *If your in-line code references compiler-declared variables using `@name`, the stack pointer must not be modified in any way.*

All other external register sources and destinations can be used freely.

4.13. Built-In Functions

The compiler recognizes built-in functions which directly address various elements of the TAAC-1 architecture. These built-in functions are better thought of as macros, since the compiler generates in-line code with knowledge of the current register state of the program.

The include file `builtin.h` defines the mnemonics used in the built-in functions. All programs using the built-in functions must contain the line:

```
#include <taac1/builtin.h>
```

The file comes with TAAC-1 C. A listing of the built-in functions defined in this file appears at the end of this chapter.

In most cases, any argument to a built-in function may be an arbitrary expression. In some cases, an argument may be required to be a constant expression.

For more information on the TAAC-1 architecture, consult the hardware overview chapter.

Built-In Function Summaries

data = input (source);

Allows data reads from one of several TAAC-1 bus sources. The result is an integer. The `source` must be an integer constant, chosen from this list:

`RD_LU`	Read Lookup Table LU Output
`RD_LT`	Read Lookup Table LT Output
`RD_AM`	Read the DRAM Mode Register
`RD_MO`	Read the Miscellaneous Mode Register
`RD_AR`	Read the Address Readback Register
`RD_FS`	Read the Floating Point Status Register

If you are reading floating point values from lookup tables LU or LT, you can use the built-in function `asfloat()` to keep the input data from being converted (incorrectly) to float. For example:

```
float x;
x = input (RD_LT);
```

converts the LT output to float before assigning it to `x`. The next example, however, treats the LT input as a float:

```
x = asfloat (input (RD_LT));
```

See the function `asfloat()` for further details.

output (destination, data);

This function allows you to write to one of several bus destinations. When the destination is LT, `data` must be a one-word type (anything but double). For all other destinations, `data` should be int, char, short, or long. No type conversion is done in any case. The destination must be a constant chosen from this list:

`WR_AM`	Write the DRAM Mode Register
`WR_MO`	Write the Miscellaneous Mode Register
`WR_LR`	Write Lookup Table Input Register LR
`WR_LT`	Write Lookup Table Input Register LT

```
val = cc ( code );
```

The operand must be integer and the result is integer. This function provides access to hardware condition codes. The codes which can be tested are:

CC_VERT	Video in a Vertical Interval
CC_RLTW	High 16 bits of DR < High 16 bits of DW
CC_INTR	Processor Interrupt State
CC_STKW	Sequencer Stack Empty or less than or equal to two spaces left

```
val = abs ( expr );
```

Returns the absolute value of the expression. The type of the result is the same as the type of the argument.

```
val = frac_mul ( expra, exprb );
```

Returns the most significant 32 bits of the 64-bit result obtained from the integer multiplication of `expra` by `exprb`. The operands are coerced to be integer and the result is integer.

```
val = rotl ( data, count );
```

The integer result is obtained by evaluating the expression `data` and performing a left rotate of `count` bits on the result.

```
val = rotr ( data, count );
```

The integer result is obtained by evaluating the expression `data` and performing a right rotate of `count` bits on the result.

```
set_ac ( mask, expr );
```

Loads some or all of the Address Count Register (AC) with the result of `expr` under the control of `mask`. The type of `expr` is coerced to be integer. The possible bits which can be ORed together into `mask` are:

AC_LDX	Load X Counter
AC_LDY	Load Y Counter
AC_LDZ	Load Z Counter

To understand the **X-Y-Z** counters as they relate to 1D, 2D, and 3D addressing, refer to the hardware overview chapter concerning addressing modes and the AC register.

upd_ac (mask);

Updates the Address Count Register (AC) under the control of `mask` with the enforced restriction that you cannot simultaneously specify an increment and decrement for the same counter. The possible bits which can be ORed together into `mask` are:

`AC_INCX`	Increment X Counter
`AC_DECX`	Decrement X Counter
`AC_INCY`	Increment Y Counter
`AC_DECY`	Decrement Y Counter
`AC_INCZ`	Increment Z Counter
`AC_DECZ`	Decrement Z Counter

data = read_ac();

Returns the result of memory read executions using the address in the AC register and the current addresing mode in the DRAM Mode (AM) register. The type of the result is integer. However, the built-in function `asfloat()` can be used to treat the result as a floating point value instead.

write_ac (expr);

Writes the result of `expr` to memory using the address in the AC register and the current addressing mode in the AM register. The argument is coerced to be integer, unless the built-in function `aslong()` is applied to `expr`.

value = asfloat(expr);

This function changes the type of its argument without changing its value. It takes the result of the expression and treats it as a `float`. The argument type may not be `double`, `union`, or `struct`. If the argument is any other type, it is passed through unchanged. The result type is `float`. This built-in function is useful in conjunction with other built-in functions that nominally return integer results. As an example:

```
float x;
x = read_ac ();
```

reads the value stored at the memory location pointed to by the AC register, converts the value to float, and stores it in `x`. In contrast,

```
x = asfloat (read_ac ());
```

reads the same value but stores it in `x` without converting it to float.

`value = aslong( expr );`

This function changes the type of its argument without changing its value. It takes the result of the expression and treats it as a `long`. The argument type may not be `union` or `struct`. If the argument type is `double`, it is first cast to `float`. If the argument is any other type, it is passed through unchanged. The result type is `long`. As an example:

```
float x;
write_ac (x);
```

would convert `x` to an integer before writing it to memory. However,

```
write_ac (aslong (x));
```

would treat `x` as if it were an integer and write it to memory without doing any conversion.

Example Using Built-in Functions

The following program uses the built-in functions to draw a diagonal line in data/image memory:

```
#include <taac1/builtin.h>
main()
{
   t_set_addr_mode (TA_2D);
                              /* set 2D addressing mode */
   set_ac (AC_LDX | AC_LDY, 100);
                              /* load x starting address */
   set_ac (AC_LDZ, 100<<16);
                              /* load y starting address */
   loop (100) {
       write_ac (0xff);       /* write a pixel */
       upd_ac (AC_INCX | AC_INCY);
                              /* increment x address */
       upd_ac (AC_INCZ);      /* increment y address */
```

```
        }
}
```

For the sake of clarity, this example loads the x and y addresses separately, and increments them separately. In actuality the two loads could be combined, as could the two increments--i.e.,

```
set_ac (AC_LDX | AC_LDY | AC_LDZ, 100 | 100 << 16);
```

and

```
upd_ac (AC_INCX | AC_INCY | AC_INCZ);
```

4.14. The Include File

builtin.h

```
#include <taac1/taacdefs.h>
#include <taac1/taregdefs.h>

/* register defs */

/* defines for built-in functions */

/* cc() */

#define _MINCC_ 0
#define CC_IBSY 0
#define CC_VERT 1
#define CC_RLTW 2
#define CC_INTR 3
#define CC_WERR 4
#define CC_STKW 5
#define CC_WBSY 6
#define _MAXCC_ 6

/* input() */

#define _MIN_IN_ 0x10
#define RD_VA    0X10
#define RD_VB    0x11
#define RD_LU    0x12
#define RD_LT    0x13
#define RD_AM    0x14
#define RD_MO    0x15
#define RD_AR    0X16
#define RD_FS    0x17
#define _MAX_IN_ 0x17

/* output() */

#define _MIN_OUT_ 0x20
#define WR_VA     0x20
#define WR_VB     0x21
#define WR_AM     0x22
#define WR_MO     0x23
#define WR_LR     0x24
#define _MAX_OUT_ 0x25

/* load commands for set_ac() */

#define AC_LDX 0x1
#define AC_LDY 0x2
#define AD_LDZ 0x4
```

```
/* incr/decr commands for upd_ac() */

#define AC_INCX 0x8
#define AC_DECX 0x10
#define AC_INCY 0x20
#define AC_DECY 0x40
#define AC_INCZ 0x80
#define AC_DECZ 0x100
```

4.15. The Include File

taacdefs.h

```
#define ON 1
#define OFF 0
#define NULL 0
#define TA_SUCCESS 0
#define TA_FAILURE -1
#define TA_DONTWRITE 0x40000000

/* base address for address banks */

#define TA_VIDEOMEM          0x00000000
#define TA_MICROCODEMEM      0x20000000
#define TA_SCRATCHPADMEM     0x30000000
#define TA_REGISTERS         0x38000000

/* addresses of registers */

#define TA_PROGRAM_COUNTER   0x38000000
#define TA_INTERRUPT_VECTOR  0x38000001
#define TA_INTERRUPT_MASK    0x38000002
#define TA_PROCESSOR_DRDW    0x38000003
#define TA_WS_READBACK       0x38000004
#define TA_WS_MODE           0x38000005
/* values for video control register */

#define TA_VIDEO_EXTERNAL    1      /* pass external video */
#define TA_VIDEO_TAAC        2      /* display TAAC-1 video */
#define TA_VIDEO_MIXEDNOTAAC 3      /* do video mix,nothing
in                                     window */
#define TA_VIDEO_MIXEDTAAC   4      /* do video mix, TAAC-1 in
                                       window */
#define TA_VIDEO_NONE        5      /* display no video */
#define TA_VIDEO_COUNT       5      /* boundary for illegal value */
#define TA_SYNC_EXTERNAL     1      /* pass external sync through */
#define TA_SYNC_GENLOCK      2      /* use TAAC sync, but genlock */
#define TA_SYNC_TAAC         3      /* use TAAC-1 sync, w/ genlock */
#define TA_SYNC_NONE         4      /* do not output sync */
#define TA_SYNC_COUNT        4      /* boundry for illegal value */

/* bits used in the Brooktree command register */

#define TA_USE_LOOKUP    0x40   /* use color look-up table */
#define TA_USE_OVERLAY   0      /* use overlay look-up table */
#define TA_BLINK_OCCULT  0      /* blink occult (75% on, 25% off */
#define TA_BLINK_FAST    0x10   /* blink fast 50/50 */
#define TA_BLINK_MEDIUM  0x20   /* blink medium 50/50 */
#define TA_BLINK_SLOW    0x30   /* blink slow 50/50 */
#define TA_BLINK1_OFF    0      /* do not blink bit 1 of overlay */
#define TA_BLINK1_ON     8      /* blink bit 1 of overlay */
#define TA_BLINK0_OFF    0      /* do not blink bit 0 of overlay */
```

```
#define TA_BLINK0_ON    4       /* blink bit 0 of overlay */
#define TA_MASK1_ON     0       /* mask bit 1 of overlay */
#define TA_MASK1_OFF    2       /* do not mask bit 1 of overlay */
#define TA_MASK0_ON     0       /* mask bit 0 of overlay */
#define TA_MASK0_OFF    1       /* do not mask bit 1 of overlay */

#define TA_MASK0_MSK    0x1     /* masks for enable/disable bits */
#define TA_MASK1_MSK    0x2
#define TA_BLINK0_MSK   0x4
#define TA_BLINK1_MSK   0x8
#define TA_BLINK_MSK    0x30
#define TA_USE_MSK      0x40

/* channel numbers */

#define TA_RED   0
#define TA_GREEN 1
#define TA_BLUE  2
#define TA_ALPHA 3
```

4.16. The Include File

`taregdefs.h`

```
/* AM register definitions */

#define TA_AMWMSK 0xf     /* word mask (write all 4 words) */
#define TA_AMWM 0x100     /* word mask mode */
#define TA_AMBM 0x200     /* bitplane mask mode */
#define TA_AM3S 0x0       /* 3D "slice" mode */
#define TA_AM3D 0x400     /* 3D "dice" mode */
#define TA_AM2D 0x800     /* 2D mode */
#define TA_AM1D 0xc00     /* 1D mode */
#define TA_AMBC 0x1000    /* bounds checking enable */
#define TA_AMSRD 0x2000   /* serial read (vram->shift reg) */
#define TA_AMSWR 0x4000   /* serial write (shift reg->vram) */
#define TA_AMSIN 0x6000   /* serial input mode (processor->shift reg) */

/* AM register masks */

#define TA_AMWM_MASK 0xf       /* mask for word mask */
#define TA_AMBM_MASK 0xf0      /* mask for bitplane mask id */
#define TA_AMWE_MASK 0x100     /* mask for wordmask mode enable */
#define TA_AMBE_MASK 0x200     /* mask for bitplane mask mode enable */
#define TA_AMDIM_MASK 0xc00    /* mask for 1D,2D,3D mode bits */
#define TA_AMBC_MASK 0x1000    /* mask for bounds-checking enable */
#define TA_AMVP_MASK 0x6000    /* mask for serial port enable bits */
#define TA_AMDIAG_MASK 0x8000 /* mask for diagnostic bit */

/* MO register */
/* default used in "cstart" routine */

#ifdef BETA12
#define TA_MO_DEFLT (TA_MOFP_SNG|TA_MOFP_RTZ|\
                     TA_MORD_WORDMODE|TA_MORC_WORDMODE|\
                     TA_MOVA_STRIDE1|TA_MOVB_STRIDE1|\
                     TA_MOMA_NR|TA_MOLU_RECP)
#else
#define TA_MO_DEFLT (TA_MOFP_FAST_MASK|TA_MOFP_SNG|TA_MOFP_RTZ|\
                     TA_MORD_WORDMODE|TA_MORC_WORDMODE|\
                     TA_MOVA_STRIDE1|TA_MOVB_STRIDE1|\
                     TA_MOMA_NR|TA_MOLU_RECP)
#endif

/* vector port fields */

#define TA_MOVA_STRIDE1 0x1     /* bank A stride = 1 */
#define TA_MOVA_STRIDE2 0x2     /* bank A stride = 2 */
#define TA_MOVA_STRIDE3 0x3     /* bank A stride = 3 */
#define TA_MOVA_STRIDE4 0x0     /* bank A stride = 4 */
#define TA_MOVB_STRIDE1 0x4     /* bank B stride = 1 */
#define TA_MOVB_STRIDE2 0x8     /* bank B stride = 2 */
#define TA_MOVB_STRIDE3 0xc     /* bank B stride = 3 */
```

```
#define TA_MOVB_STRIDE4 0x0     /* bank B stride = 4 */

/* RD configuration fields */

#define TA_MORD_BYTE0  0x0    /* byte mode, status from byte 0 */
#define TA_MORD_BYTE1  0x10   /* byte mode, status from byte 1 */
#define TA_MORD_BYTE2  0x20   /* byte mode, status from byte 2 */
#define TA_MORD_BYTE3  0x30   /* byte mode, status from byte 3 */
#define TA_MORD_HALFLO 0x40   /* halfword mode, status from low halfwd */
#define TA_MORD_HALFHI 0x50   /* halfword mode, status from high halfwd */
#define TA_MORD_WORDMODE 0x60 /* word mode */

/* RC configuration fields */

#define TA_MORC_BYTE0  0x0     /* byte mode, status from byte 0 */
#define TA_MORC_BYTE1  0x80    /* byte mode, status from byte 1 */
#define TA_MORC_BYTE2  0x100   /* byte mode, status from byte 2 */
#define TA_MORC_BYTE3  0x180   /* byte mode, status from byte 3 */
#define TA_MORC_HALFLO 0x200   /* halfword mode,status from low halfwd */
#define TA_MORC_HALFHI 0x280   /* halfword mode,status from high halfwd */
#define TA_MORC_WORDMODE 0x300 /* word mode */

/* floating-point fields */

#define TA_MOFP_SNG 0x4000     /* floating pt. single precision */

/* FP rounding modes */

#define TA_MOFP_RTN 0x0       /* round-to-nearest */
#define TA_MOFP_RTZ 0x1000    /* round towards zero */
#define TA_MOFP_RUP 0x2000    /* round up */
#define TA_MOFP_RDN 0x3000    /* round down */

/* MA rounding modes */

#define TA_MOMA_NR 0x0          /* no rounding */
#define TA_MOMA_R30 0x80000     /* round using bit 30 */
#define TA_MOMA_R31 0x100000    /* round using bit 31 */
#define TA_MOMA_R3031 0x180000  /* round using bits 30 and 31 */

/* MO register Lookup table function bits */

#define TA_MOLU_RECP 0           /* reciprocal of floating-pt. value */
#define TA_MOLU_SQRT 0x200000    /* square root of float */
#define TA_MOLU_RSQT 0x400000    /* reciprocal of square root of float */
#define TA_MOLU_ICOS 0x600000    /* cosine of integer */
#define TA_MOLU_ISIN 0x800000    /* sine of integer */
#define TA_MOLU_IRCP 0xa00000    /* reciprocal of integer */

/* MO register masks */

#define TA_MOVAVB_STRIDE_MASK 0xf  /* mask for stride bits in MO */
```

```
#define TA_MORCRD_MASK 0x3f0        /* mask for RC and RD config field */
#define TA_MOFP_CLOCK_MASK 0x400    /* mask for FP clock mode */
#define TA_MOFP_FAST_MASK 0x800     /* mask for FP fast mode */
#define TA_MOFP_ROUND_MASK 0x3000   /* mask for FP round mode */
#define TA_MOFP_CONFIG_MASK 0xc000 /* mask for FP configuration field */
#define TA_MOMA_MASK 0x180000       /* mask for MA round mode field */
#define TA_MOLU_MASK 0xe00000       /* mask for lookup table fctn fld */
```

5

The Assembler (tasm)

5

The Assembler (tasm)

tasm is a two-pass, relocatable assembler for the TAAC-1. It is invoked by the tacc compiler, to convert assembly code into object code.

5.1. tasm Command Syntax

tasm uses this command syntax:

```
tasm [-b] [-l] name
```

where name is the name of a file. If the filename suffix is omitted or is not .asm, tasm adds .asm. The assembler then reads name.asm and writes name.obj.

The -b option tells tasm to omit blanks lines from the listing file. If you have passed the listing file through the tacpp macro preprocessor and made extensive use of conditionals, use the -b option to reduce the size and improve the readability of the listing file.

When the -l option is given, tasm places a listing file containing the values of each instruction in a file named name.lst.

5.2. Using Assembler Commands

Valid variable names use alphabetic characters, dots (.), underbars (_), and dollar signs ($) in any order.

When a single dot is used as a variable name, the assembler replaces the dot with the current program counter value.

A name may be arbitrarily long, but only the first 20 characters of the name are significant.

`tasm` is completely case-sensitive.

Blanks and tabs separate tokens, but are not otherwise significant.

Very long lines can be continued by placing a backslash character at the end of the line, just before the newline character.

Comments start with a semi-colon (`;`). Start comments anywhere on a line. Comments stop at the end of the starting line. Alternatively, a comment can consist of anything between `/*` and `*/`, as in standard C.

All numeric entries must be in C-style binary, decimal, octal, or hexadecimal form.

All binary numbers must start with `0b` or `0B`.

Hexadecimal numbers must start with `0x` or `0X`.

Octal numbers must start with `0`.

Decimal numbers must start with a non-zero digit.

Floating point numbers currently must be represented in an integer format.

5.3. Defining Constants

To associate a name with a constant expression in definitions files, use the format:

```
name=<num_expr>
```

where `name` is the name of the constant and `<num_expr>` is the value assigned to the constant.

Constant names defined this way can be used in numeric expressions during assembly.

5.4. Assembler Input File Format

Any line can begin with a label, which is a name followed by a colon. The effect of defining a label is to define the name as a constant whose value is the current program counter. Multiple labels on a line are allowed. You may define a label only once.

Lines

There are basically three types of assembler lines:

1. Instruction defining
2. Assembler directive
3. Null (containing only labels and/or comments)

Numeric Expressions

The rules for forming numeric expressions are:

```
<num-expr>   <num-expr> '+'   <unary>     (subtraction)
           | <num-expr> '-'   <unary>     (addition)
           | <num-expr> '&'   <unary>     (bitwise and)
           | <num-expr> '|'   <unary>     (bitwise or)
           | <num-expr> '*'   <unary>     (multiplication)
           | <num-expr> '/'   <unary>     (division)
           | <num-expr> '%'   <unary>     (remainder)
           | <num-expr> '>>'  <unary>     (right shift)
           | <num-expr> '<<'  <unary>     (left shift)
           | <unary>
<unary>    = '-' <primary>                (negation)
           | '~' <primary>                (one's complement)
           | <primary>
<primary>  = number
           | name
           | '(' <num-expr> ')'
<constant> = number | name
```

Constant expressions are formed using addition, subtraction, bitwise AND, bitwise OR, multiplication, division, remainder, or negation operators. The assembler evaluates expressions from left to right, except when parentheses alter the order of evaluation. The result is a signed, two's-complement integer.

Any expression involving the use of a label or an external symbol (defined by `.extern`) is said to be a *relocatable* expression. Any other expression is *absolute*.

When one relocatable expression is subtracted from another, both expressions must be relative to the same segment, in which case the result of the expression is absolute. If not, `tasm` generates an "Illegal relocatable expression" error message.

Table 5-1 *Restrictions on Relocatable Expressions*

Left	*Op*	*Right*	*Result*
reloc	'+'	abs	reloc
abs	'+'	reloc	reloc
reloc	'-'	abs	reloc
reloc	'-'	reloc	abs

5.5. Assembler Directives

`.end` — Forces end of input.

`.extern <name-list>` — Identifies externally defined labels.

`.global <name-list>` — Identifies labels which may be referenced in other source files.

`.org <num-expr>` — Sets the current program counter to `<num-expr>`, which must be either absolute or relative to the current segment. In the expression, you must not refer to a constant name or label which is defined after `.org`. All values used in `<num-expr>` must be fully defined in the first pass before `.org`. The new origin is always relative to the start of the current segment.

`.align <num-expr>` — Sets the current pc to the next address which is a multiple of `<num-expr>`. For instance, `.align 4` moves the PC to the next four-word boundary, if the PC is not already at that boundary. `<num-expr>` must be absolute and a power of 2. The start address of the current segment in the current file is also forced to the desired alignment boundary. If there is more then one `.align` directive in a segment in a file, then the segment will be aligned

	according to the maximum `.align` value found.
`.zero <num-expr>`	Sets the next `<num-expr>` entries in the current segment to zero.
`.bss <num_expr>`	This directive reserves the next `<num-expr>` entries in the current segment, but does not assign any initial value to them.
`.title <text>`	`<text>` consists of everything up to the end of the `.title` line. If listing is enabled (see `.list`), `<text>` displays on subsequent page headings.
`.eject`	If listing is enabled (see `.list`), `.eject` causes the next line to appear on a new page.
`.nolist`	Inhibits the generation of the listing file.
`.list`	Reenables the printing of lines to the listing file. `.list` and `.nolist` directives may be nested and simply cancel each other. If more `.nolist` directives than `.list` directives exist, then `tasm` disables the listing.
`.comment <text>`	Writes a comment record, containing the given `<text>`, to the relocatable object file.
`name=<num-expr>`	Gives a symbolic name to a constant. The name may be redefined as required. The `<num-expr>` may contain references to values which are defined later, as long as all values are fully defined by the time `tasm` starts processing the line in the second pass.
`# line-number "filename"`	This type of line is put out by the C Preprocessor. Since `tasm` accepts this type of line as input, it will happily

assemble a `tasm` program which has first been passed through the C Preprocessor `tacpp`. Line numbers and file names in error messages are adjusted to be relative to the original C preprocessor input.

5.6. Segments

The main purpose of having segments is to allow you to deal with different memory types in the same source file.

Segment names are treated as a separate class of names, so it is possible to have both a label and a segment with the same name.

`abs=` Defines an absolute segment which must be loaded starting at the given address. All other segments are relocatable, which means that the linker can place them anywhere in memory. Space for absolute segments is always allocated first by the linker. The linker generates an error message if you attempt to overlap two or more absolute segments.

`memtype=` `memtype` allows you to differentiate between different types of memory - program (instruction) memory, scratchpad memory and data/image memory. The default is program memory.

`memtype=0` specifies program memory (PRAM).
`memtype=1` specifies scratchpad memory (SRAM).
`memtype=2` specifies data/image memory (DRAM).

To create an instruction segment, use:

```
.segment segname,memtype=0
```
(code goes here)
To create a scratchpad segment, use:

```
.segment segname,memtype=1
```
(data goes here)

To create a DRAM segment, use:

```
.segment segname,memtype=2
```

To put a segment at an absolute location, use abs=n, where n is the desired starting address of the segment. For example, to put a particular instruction segment at the beginning of instruction memory, use:

```
.segment segname,memtype=0, abs=0
```

6

The Linker (talink)

6

The Linker (talink)

talink is a two-pass linker that combines relocatable object modules produced by the tasm assembler into an absolute output module suitable for execution on the TAAC-1.

6.1. talink Command Syntax

```
talink [-g][-m mapname][-o outname][-f name] object_file*
       [-Ldirectory][-lfilename][-d start end]
```

`object_file` — Names the object file to be linked. If the filename suffix is omitted, talink automatically appends .obj.

With programs written in C and containing a main(), talink automatically links in the file start.obj, which initializes the C stack pointer, the AM register, and the MO register, and jumps to main(). start.obj is part of the runtime library, rtlib.tlb.

`-g` — Links in routines to support debugging with tadeb.

`-m mapname` — Names the load map output file mapname.map. If -m is not used, the linker outputs a map file of the name outname.map. If -m is used and the mapname includes a ".", the string after the period is replaced with the suffix map.

`-o outname` — Names the output executable file outname.abs. If -o is not used, then the prefix before a "." of the first object filename is used. If no object files are given, the prefix of the first library file name is

used. If -o is used and the outname includes a ".", the string after the period is replaced with the suffix abs.

-f name

Creates an external symbol name, equivalent to one created by a .extern assembler directive, before any object files or libraries are read. Use this option to force an object module containing a global definition for the symbol to be loaded from a library.

-lfilename

Gives the name of an object library created by talib or by concatenating object modules. Only those modules which are needed to resolve previous external references are loaded. Library files are read in the order they appear on the command line, and each library is searched sequentially. The linker decides whether or not to load a particular object module based on the symbols it has already seen. Thus the order of object files and libraries on the command line and the order of object modules within a library file can be significant.

-d start end

Specifies an area of memory in which the linker can allocate space for DRAM variables. start must be a number, which identifies the starting address. end identifies the ending address. All numbers may be in C-style decimal, hex, or octal format. The valid address range is 0x0 to 0x01f ffff. You can specify the -d option multiple times, to allocate more than one memory area.

-Ldirectory_name

Specifies a directory in which the linker should search for libraries. This option applies to all libraries that follow it in the command line. As an example, if the talink command were:

```
talink myfile -L/usr/mydir -lmylib
```

the linker would search in /usr/mydir/mylib.tlb for all library routines.

Currently, the -L option specifies the *only* directory in which the linker will search. That is, if it does not find a routine in the specified directory, it will not look in /usr/lib as well. The TAAC-1 Linker always links the run-time library, rtlib.tlb, into all programs. If you use the -L option to specify an alternate library directory, talink will look in this directory for rtlib.tlb, also. Unless you want to substitute an alternate version of the runtime library, the last option of your talink command should be -L/usr/lib/taac1, as shown in this example:

```
% talink myfile -L/mydir mylib.tlb -L/usr/lib/taac1
```

7

The Object Librarian (`talib`)

7

The Object Librarian (talib)

The talib object librarian creates and manipulates libraries of object modules (files) produced by the tasm relocatable assembler. These libraries can be loaded and searched by the linker talink. Libraries consist of concatenated object modules.

7.1. talib Command Syntax

```
talib [option]* library [module module . . .]
```

library	Tells talib the name of the working library to which all operations are applied. If the filename has no suffix, talib normally appends .tlb. This option must appear exactly once.
module	Names the object module (file) to be edited into the working library. Object modules with the same name as modules already in the library will replace those modules. Other modules are added to the end of the working library. Filenames must include the correct suffix; if the suffix is omitted, talib assumes a suffix of .tlb.
-d module	Removes a module from the working library. If named module does not exist, the option is ignored.
-m module1 module2	module1 moves within the library to the position immediately in front of module2. Both modules must already be present in the library.

`-p module`	Lists the named module . The default module listing contains the name of the module and the time of its creation. Under the `-v` option, the listing includes information about the segments and symbols in the module(s).
`-t`	Lists all modules. The default module listing contains the name of the module and its creation date. Under the `-v` option, the listing includes information about the segments and symbols in the module(s).
`-l symbol`	This option gives the name of a symbol as defined in a `.global` or a `.extern` assembly language directive. `talib` lists the name of any module containing the given symbol.
`-u`	Normally, if an object module added under the `filename` option has the same name as a module already in the library, the existing module is replaced by the new module. Replacement occurs only if the replacement module is newer than the library module. The age of a module is determined by the creation time in its start record.
`-c`	Tells `talib` to ignore the initial contents of the working library. Using this option is equivalent to deleting the library file before running `talib`.
`-v`	Provides verbose output. More specifically, `-v` causes information about segments and symbols to be displayed under the `-p` or `-t` option. This option may be repeated for additional information.

The options that can modify the contents of the working library (`module`, `-d module`, and `-m module1 module2`) are performed in the order they appear in the command line. Occasionally this can be

significant; for instance, if a library contains modules named a, b, and c, in that order, the option pairs:

```
-m a c -m c b
```

and:

```
-m c b -m a c
```

will produce different results.

8

Assembly Language

8

Assembly Language

The TAAC-1 processor employs five buses, two registered ALUs, a multiplier/accumulator, a barrel shifter, a processor lookup table, and a sequencer. Figure 8-2 illustrates the processor architecture. Rounded corners in this diagram represent elements that are not registered, such as the bus transceivers, the barrel shifter output, and the lookup table output.

Each major processor component has a set of assembly language instructions that drive it. This chapter has one section for each major processor component; each section describes the applicable assembly language instructions. Other sections describe methods of memory access, data flow, and bus communication.

You may also wish to refer to the compiler chapter, which contains a section describing in-line code combined with C programs.

8.1. The Processor and Instruction Word

This section provides the instruction word bit map and the processor block diagram, for reference as you read this chapter.

The TAAC-1 processor uses a 200-bit instruction word to coordinate simultaneous operations among processor components. Fields in the instruction word control the TAAC-1 processor components and move data over the local bus and the six processor buses.

Figure 8-1 *Information Carried in the TAAC-1 Instruction Word*

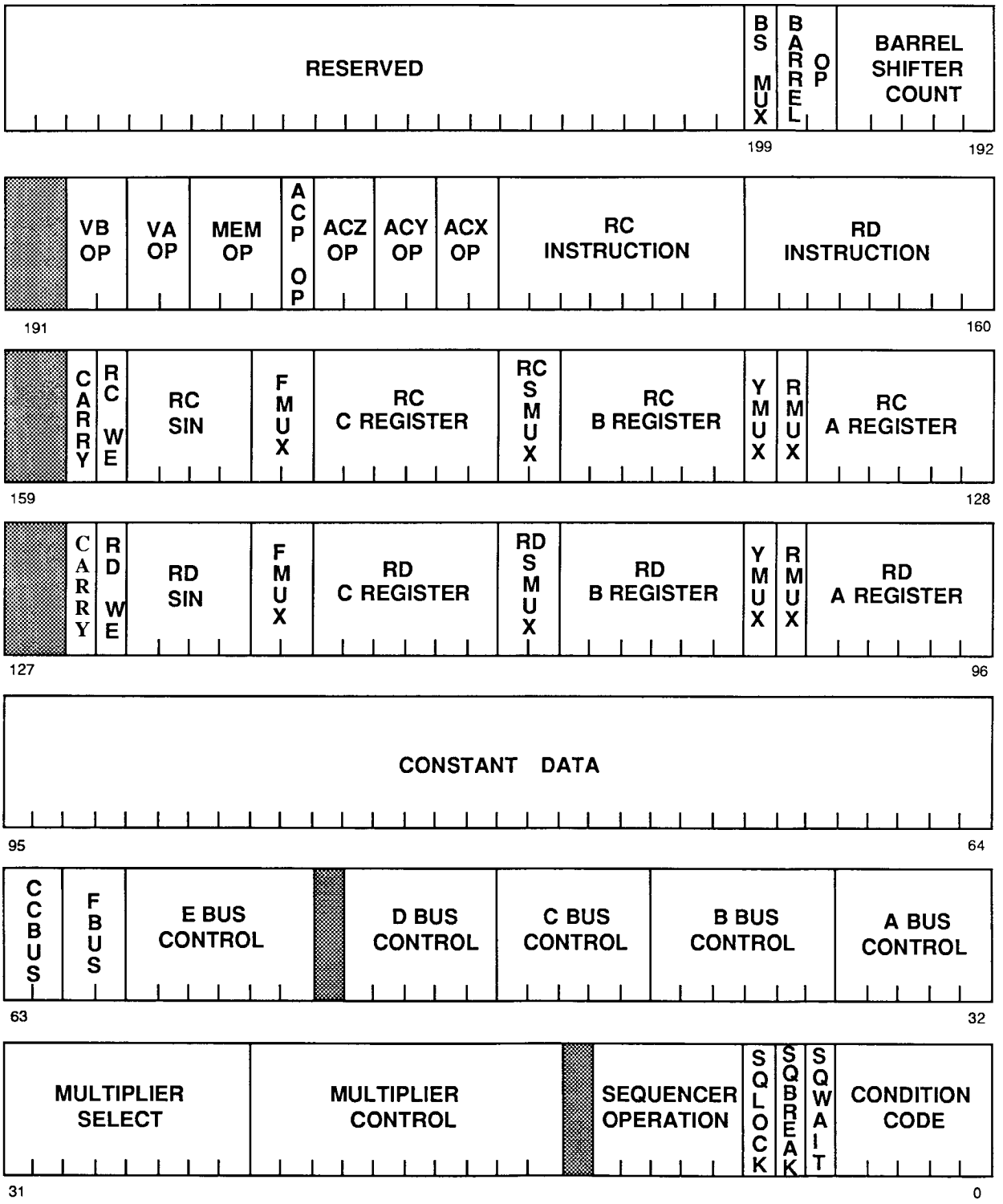
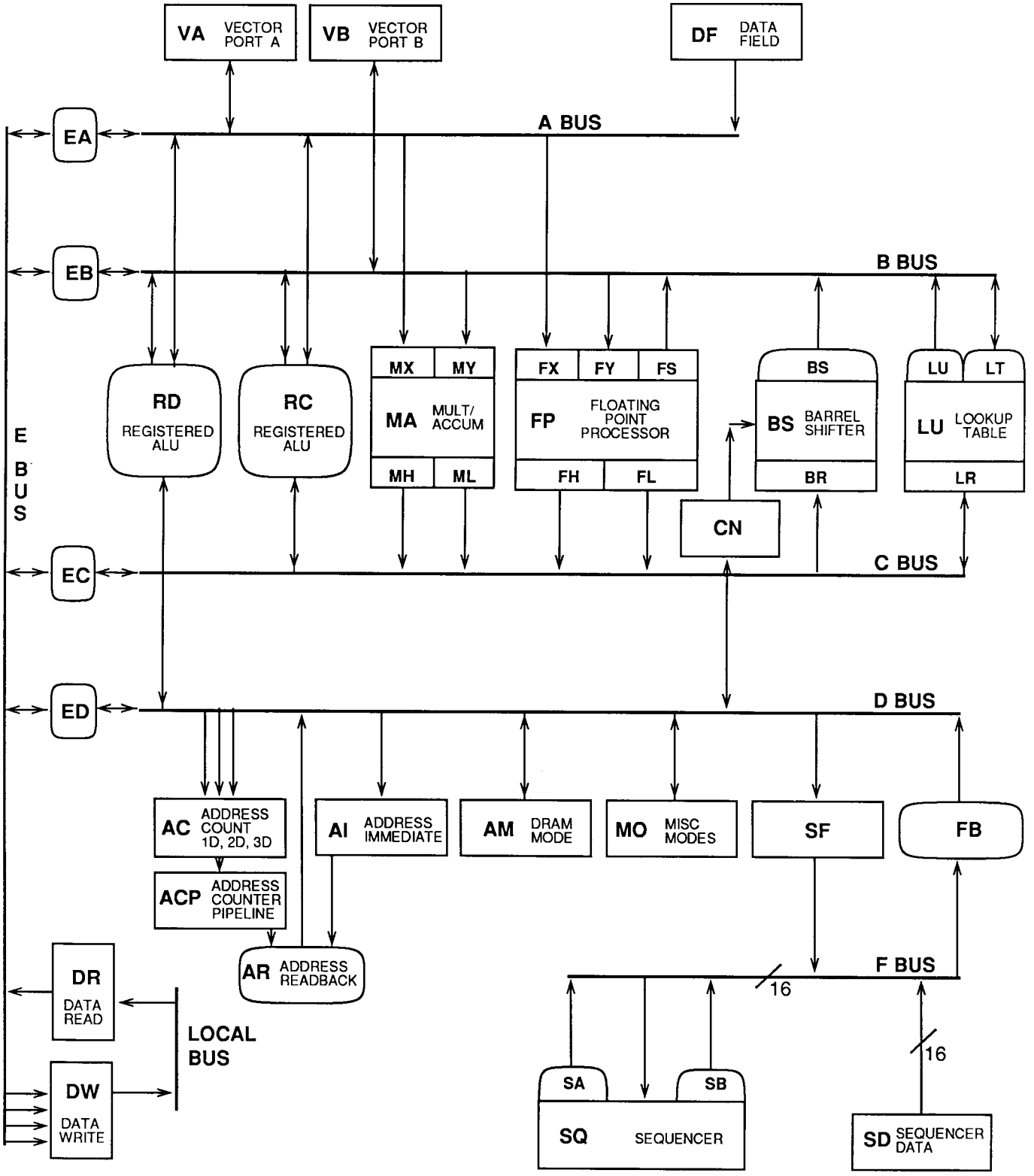


Figure 8-2 *TAAC-1 Processor Architecture*



Default Instruction Word

The default instruction word is defined in the mnemonics file used by the assembler. It contains these fields:

```
bs_sra,15 vbnop vanop mnop macn acznop acynop acxnop rc_rareg\
rc_sbreg rc_nop rd_rareg rd_sbreg rd_nop rd_car0 rd_nowe\
rd_yalu rc_car0 rc_nowe rc_yalu ccrc\
f_sd e_ea d_fb c_lr b_fs a_va cont sqnolock sqnobp sqnowt \
ma_xuns ma_yuns nomult rc_sin#15 rd_sin#15
```

8.2. Sequencer (SQ) Instructions

The sequencer determines the next instruction address, controls the sequencer stack, and controls two register/counters. Sequencer outputs SA and SB are F bus sources. SB always contains the value of Register/Counter RCB. SA varies with the sequencer operation.

Unconditional Instructions

Register/Counter A (RCA) and Register/Counter B (RCB) can be used as loop counters. They hold 16-bit values; a value of 0 is interpreted as a count of 65536.

These instructions affect the output of SA:

`pofb`	Pop the stack and put the value onto the SA bus.
`rdsp`	Put the value of the stack pointer onto the SA bus.
`rdst`	Put the value on the top of the stack onto SA bus.
`pora`	Pop the stack and put the value into RCA. Puts the value on the top of the stack onto SA bus.
`porb`	Pop the stack and put the value into RCB. Puts the value on the top of the stack onto SA bus.

These unconditional instructions put the value of Register/Counter RCA onto the SA bus.

`pura`	Push the value of RCA onto the stack.
`pufb`	Push the value on the F bus onto the stack.
`pump`	Push the value of the Program Counter (MPC) onto the stack.
`ldra`	Load RCA with the value on the F bus.
`ldrb`	Load RCB with the value on the F bus.

`ldsp`	Load the Stack Pointer (SP) with the value on the F bus.
`cont`	No operation (default).

Conditional Jumps

The F bus register (SF) and the sequencer data field (SD) are the F bus sources for conditional jumps. The sequencer data field is taken from the low 16 bits of the Constant Data Field of the instruction word.

`juca`	Decrement RCA. If condition is true and RCA is not zero, jump to the address on the F bus. If condition is true and RCA is zero, continue to the next instruction. If condition is false, jump to the address on the F bus.
`jucb`	Decrement RCB. If condition is true and RCB is not zero, jump to the address on the F bus. If condition is true and RCB is zero, continue to the next instruction. If condition is false, jump to the address on the F bus.
`jura`	If condition is true, jump to the address in RCA, else continue.
`jufb`	If condition is true, jump to the address on the F bus, else continue.
`just`	If condition is true, jump to the address on the top of the stack.
`lpst`	If condition is true, jump to the address on the top of the stack, else pop the stack and continue.
`lpxa`	If condition is true , pop the stack and jump to the address in RCA, else continue.
`lpxf`	If condition is true, pop the stack and jump to the address on the F bus, else continue.
`lpca`	Decrement RCA. If condition is true and RCA is not zero, jump to the address on the stack. If condition is true and RCA is zero, pop the stack and continue. If condition is false, jump to the address on the F bus.
`lpcb`	Decrement RCB. If condition is true and RCB is not zero, jump to the address on the stack. If condition is true and RCB is zero, pop the stack and continue. If condition is false, jump to the address on the F bus.

`lpsb`	Decrement RCB. If condition is true and RCB is not zero, jump to the address on the stack. If condition is true and RCB is zero, jump to the address in RCA. If condition is false, continue.

Conditional Subroutine Calls

`jsra`	If condition is true, push MPC and jump to RCA, else continue.
`jsfb`	If condition is true , push MPC and jump to the address on the F bus, else continue.
`jsaf`	Push MPC. If condition, jump to RCA, else jump to RCB.

Conditional Returns

`retn`	If condition is true, pop the stack and jump to the popped address.

The default condition code is `ccfalse`. Therefore,

```
jufb LABEL
```

would not branch to `LABEL`. For an unconditional branch, use:

```
jufb cctrue LABEL
```

See "The Condition Code Multiplexer" in the next section for more information on condition codes.

Example: Loop Counter Loaded with Constant

The `ldra` instruction loads the loop counter with 10. The `juca` instruction decrements the loop counter; if the count is not zero, it branches to `LOOP`; otherwise, it falls through to the next instruction.

```
      f_sd ldra 10            ;load RCA
LOOP:
      .
      .
      juca cctrue LOOP        ;decrement RCA, jump
```

Example: Loop Counter Loaded with Variable

The `ldra` instruction loads the loop counter with the value stored in RD0. The `juca` instruction decrements the loop counter; if the count is not zero, it branches to `LOOP`; otherwise, it falls through to the next instruction.

```
      rd_a#0 rd_pr d_rdsf
      f_sf ldra                    ;load RCA
LOOP:
      .
      .
      juca cctrue LOOP             ;decrement RCA, jump
```

Example: Branch to Label

Because the condition code is `cctrue`, this instruction branches unconditionally to label `DRAWX`.

```
      f_sd jufb cctrue DRAWX
```

Example: Branch to Computed Address

This is another unconditional branch to the address stored in the Data Read (DR) register.

```
      e_dr d_edsf                  ;SF <- DR
      f_sf jufb cctrue             ;branch
```

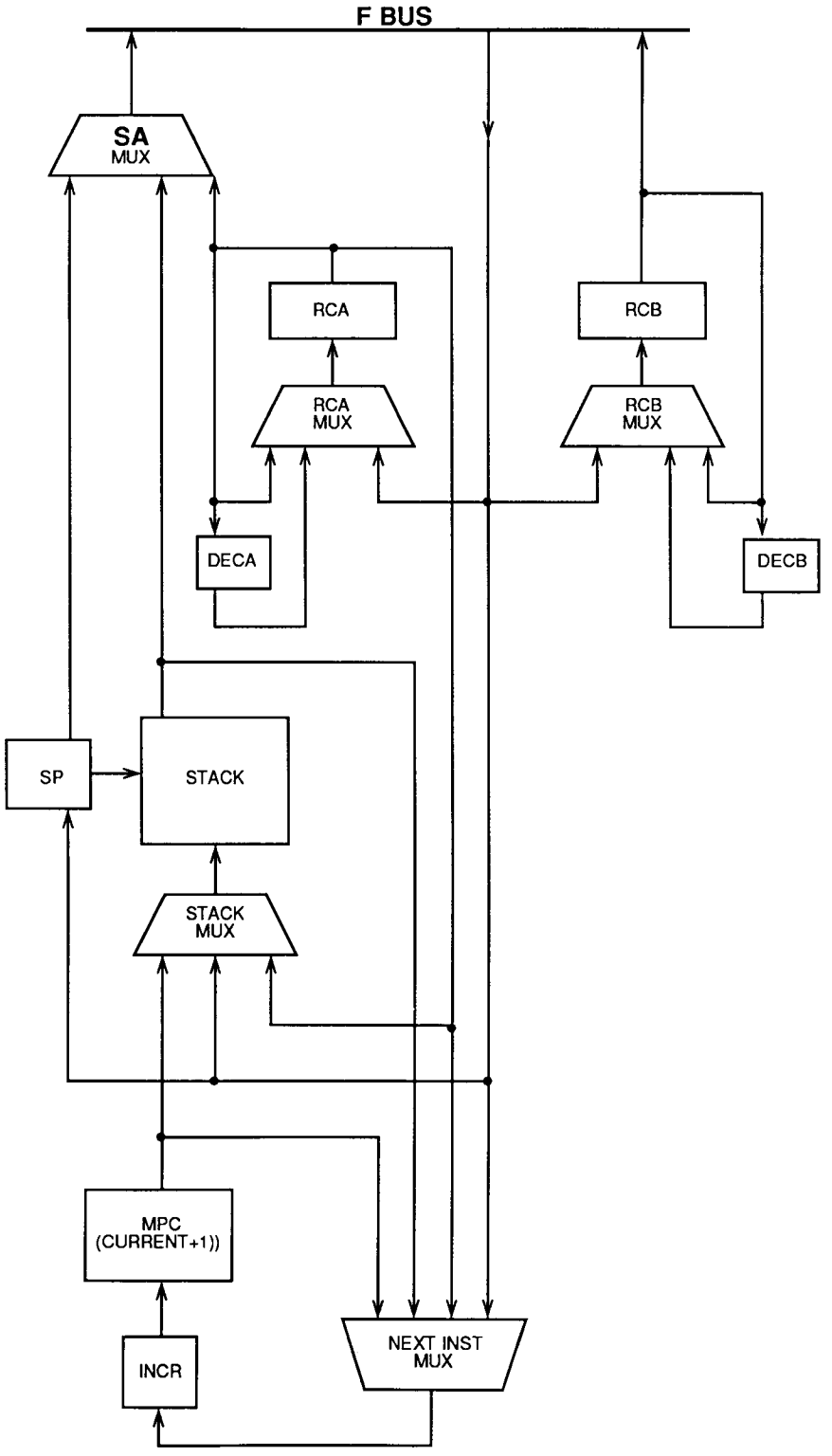
The Condition Code Multiplexer

Registered ALUs RC and RD, along with the floating point processor (FP) are all sources of sequencer statuses ZERO, NEGATIVE, OVERFLOW, and CARRY. These instructions select the sequencer status source:

`ccrc` Tells the sequencer to use the status of RC in the next instruction. Use `ccrc` in the same instruction as the RC operation to be tested. If `sqwait` is specified, the sequencer uses the status from RC in the current *and* in the next instructions, by default.

`ccrd` Tells the sequencer to use the status of RD in the next instruction. Use `ccrd` in the same instruction as the RD operation to be tested. If `sqwait` is specified, the sequencer uses the status from RD in the current *and* in the next instructions, by default.

Figure 8-3 *The Sequencer (SQ)*



`ccfp`	Tells the sequencer to use the status of FP in the next instruction. Use this instruction after any FP compare operation up to and including the next FP operation. `sqwait` does not affect this instruction.
`ccff`	Tells the sequencer to use the status of FP in the current *and* the next instructions. Use this instruction after any FP compare operation up to and including the next FP operation. sqwait does not affect this instruction.

To jump if an RC or RD operation produces a CARRY, use either:

INSTR 0	aluop `ccrc`
INSTR 1	jufb `cccar`

Or:

INSTR 0	aluop `ccrc jufb cccar sqwait`
INSTR 1	the RC status is still valid here

To jump if an FP compares A to be greater than B, use:

INSTR 0	`fp_cmp fp_aax fp_aby`
INSTR 1	`jufb ccagtb ccff`
INSTR 2	the FP status is still valid here

Interrupts

`sqbp`	Generates an interrupt, for debugging.
`sqlock`	Prevents an interrupt at this instruction.

Condition Code

`cctrue`	Always true (also `nccfalse`).
`ncctrue`	Always false (also `ccfalse`) (default).
`ccvert`	Video in vertical interval.
`nccvert`	Video not in a vertical interval.
`ccrltw`	High 16 bits of DR < high 16 bits of DW (unsigned).
`nccrltw`	High 16 bits of DR >= high 16 bits of DW (unsigned).
`ccintr`	Processor in interrupt state.
`nccintr`	Processor not in an interrupt state.
`ccstkw`	Stack is empty or has two or fewer available spaces.
`nccstkw`	Stack is not empty or has more that two available spaces.

These condition codes are useful for testing the status of RC or RD:

`cccar`	CARRY status out of selected processor (FP, RD, or RC).
`ncccar`	No CARRY status out of selected processor.
`ccneg`	NEGATIVE status out of selected processor.
`nccneg`	No NEGATIVE status out of selected processor.
`ccovr`	OVERFLOW status out of selected processor.
`nccovr`	No OVERFLOW status out of selected processor.
`cczero`	ZERO status out of selected processor.
`ncczero`	No ZERO status out of selected processor.
`cccnz`	`ncccar` OR `cczero`.
`ncccnz`	`cccar` AND `ncczero`, same as NOT (`ncccar` OR `cczero`).
`cccoz`	`cccar` OR `cczero`.
`ncccoz`	`ncccar` AND `ncczero`, same as NOT (`cccar` OR `cczero`).
`ccltz`	`ccneg` XOR `ccovr`
`nccltz`	NOT (`ccneg` XOR `ccovr`).
`cclez`	`ccneg` XOR `ccovr` OR `cczero`.
`ncclez`	NOT (`ccneg` XOR `ccovr` OR `cczero`).

These condition codes are useful for testing the status of the `fp_cmp` instruction in FP:

`ccagtb`	The A input to the ALU was greater than the B input.
`ccaleb`	The A input was less than or equal to the B input.
`ccaneb`	The A input was not equal to the B input.
`ccaeqb`	The A input was equal to the B input
`ccaltb`	The A input was less than the B input
`ccageb`	The A input was greater than or equal to the B input.

8.3. Constant Data Field

The Constant Data field is a 32-bit value specified in the instruction word. It is a source on the A bus for immediate data. To use the data field, specify:

```
a_df <value>
```

The least significant 16 bits of the same field are used for sequencer data (SD), to specify branch destinations.

8.4. ALU (RC, RD) Instructions

The ALUs require nine instruction fields for control. Two instruction fields specify source registers. Source registers are inputs to the R and S multiplexers for ALU operations. Both of the source register instruction fields are available on the A and B buses, as sources. To define the source registers, use:

`r?_a#n`	The A source register is register number n (0 to 63).
`r?_b#n`	The B source register is register number n (0 to 63).

Another instruction field specifies the destination register:

`r?_c#n`	The C destination register is register number n (0 to 63).

When you are writing in-line code within a C program, you can reference register variables by using:

```
@name
```

where `name` is the variable name, instead of a register number (for example, `rc_a#@x`). See the compiler chapter for details.

Four instruction fields select the data path through the ALU. Data paths allow operands to come from various sources:

`r?_rareg`	R operand is the A register (default).
`r?_rabus`	R operand is the A bus.
`r?_sbreg`	S operand is the B register (default).
`r?_sbbus`	S operand is the B bus.
`r?_smq`	S operand is the MQ register.
`r?_yalu`	Y bus output is the ALU/shifter output (default).
`r?_ymq`	Y bus output is the MQ register.
`r?_fabus`	A bus value written to the register file.
`r?_fbbus`	B bus value written to the register file.
`r?_fyout`	Y bus value written to the register file.
`r?_f?bus`	C or D bus value written to the register file. C bus for RC, D bus for RD.
`r?_nowe`	No value written to the register file (default).

sin Field

One particular instruction field has variable meaning depending on the ALU instruction. This field is a four-bit constant used for selection of bytes and as fill bits for shift operations:

`r?_sin#n`	Where n is a value from 0 to 15.

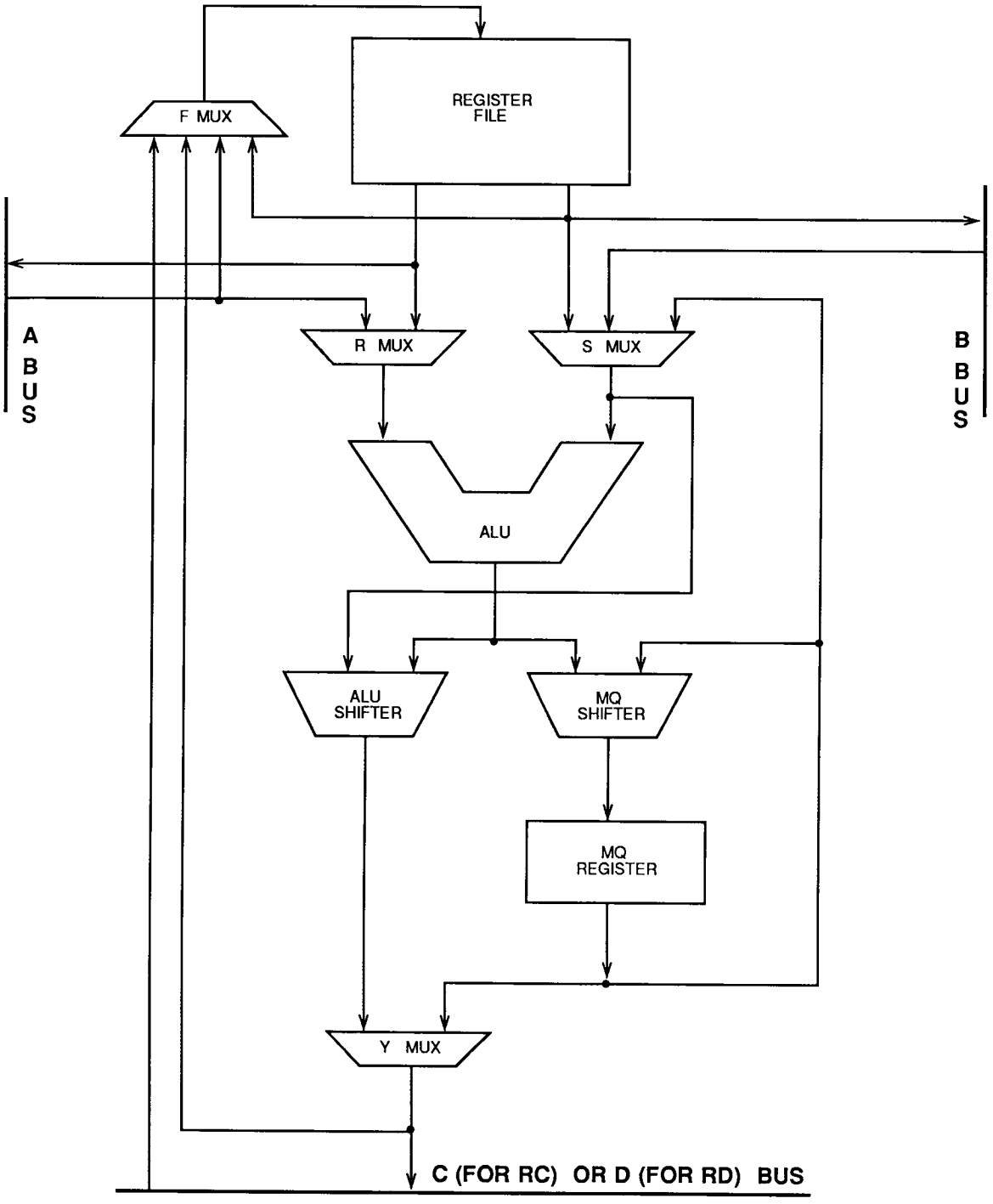
The defaults are `rc_sin#15` and `rd_sin#15`.

ALU/Shifter Operations

The most complex instruction field controls ALU/shifter operation. The field allows independent shifter and ALU operations in some instructions, while others combine the two parts for more complicated operations. Standard C syntax is used here: ~ represents NOT, | is OR, ^ is XOR, and & is AND. These simple ALU operations may be combined with a shifter operation:

`r?_add`	ALU out = R + S
`r?_addinc`	ALU out = R + S + 1
`r?_subr`	ALU out = S - R
`r?_subs`	ALU out = R - S
`r?_subrdec`	ALU out = S - R - 1
`r?_subsdec`	ALU out = R - S - 1
`r?_ps`	ALU out = S
`r?_pr`	ALU out = R
`r?_ms`	ALU out = -S
`r?_mr`	ALU out = -R
`r?_ocs`	ALU out = ~S
`r?_ocr`	ALU out = ~R
`r?_incs`	ALU out = S + 1
`r?_incr`	ALU out = R + 1
`r?_xor`	ALU out = R ^ S
`r?_and`	ALU out = R & S
`r?_or`	ALU out = R | S
`r?_nand`	ALU out = ~ (R & S)
`r?_nor`	ALU out = ~ (R | S)
`r?_andnr`	ALU out = ~R & S
`r?_nop`	no operation (default)

Figure 8-4 *ALUs RC and RD*



Example: Add RC0 and RC1, Write Result to RC2.

The `rc_fyout` instruction is necessary for the result to be written to RC2.

```
rc_rareg rc_a#0 rc_sbreg rc_b#1 rc_add rc_fyout rc_c#2
```

These shift operations *must be accompanied by an ALU operation*:

`r?_sra`	Shift out = ALU out shifted right arithmetic 1.
`r?_srad`	Shift out = ALU out shifted right arithmetic 1, fill with shift out of MQ. MQ = MQ shifted right logical 1, fill with shift out of ALU.
`r?_srl`	Shift out = ALU out shifted right logical 1, fill with ~(SIN bit 0).
`r?_srld`	Shift out = ALU out shifted right logical 1, fill with ~(SIN bit 0). MQ =MQ shifted right logical 1, fill with shift out of ALU.
`r?_sla`	Shift out = ALU out shifted left arithmetic 1, fill with ~(SIN bit 0).
`r?_slad`	Shift out = ALU out shifted left arithmetic 1, fill with shift out of MQ. MQ = MQ shifted left logical 1, fill with ~(SIN bit 0).
`r?_slc`	Shift out = ALU out shifted left circular 1.
`r?_slcd`	Shift out = ALU out shifted left logical 1, fill with shift out of MQ. MQ = MQ shifted left logical 1, fill with shift out of ALU.
`r?_src`	Shift out = ALU out shifted right circular 1.
`r?_srcd`	Shift out = ALU out shifted right logical 1, fill with shift out of MQ. MQ = MQ shifted right logical 1, fill with shift out of ALU.
`r?_mqsra`	Shift out = ALU out. MQ = MQ shifted right arithmetic 1.
`r?_mqsrl`	Shift out = ALU out. MQ = MQ shifted right logical 1. Fill with ~(SIN bit 0).
`r?_mqsll`	Shift out = ALU out. MQ = MQ shifted left logical 1, fill with ~(SIN bit 0).
`r?_mqslc`	Shift out = ALU out. MQ = MQ shift left circular.
`r?_loadmq`	Shift out = ALU out. MQ = ALU out.
`r?_pass`	Shift out = ALU out (default).

Example: Add RC1, RC2, Shift, Write to RC3

Add RC1 and RC2, shift the result to the right one bit, and write the result to RC3:

```
rc_rareg rc_a#1 rc_sbreg rc_b#2 rc_add rc_sra rc_fyout rc_c#3
```

Operations on Selected Bytes

The next set of more complicated operations must be the only ones performed by the ALU. Some of these operations work only on selected bytes. The selected bytes have zero in the corresponding bit of `sin`. Two common operations are `r?_addi` and `r?_subi`, which allow you to add or subtract, respectively, a constant between 0 and 15.

`r?_set0` The register addressed by B is both the source and destination for this instruction. The selected bytes of the B register are ANDed with a mask formed by an iverted concatenation of the low four bits of the C register address. (~C#-C0: :A#-A0).

`r?_set1` The register addressed by B is both the source and destination of this instruction. The selected bytes of the B register are ORed with a mask formed by concatenating the low four bits of the C register address and the low four bits of the A register address (C3-C0: :A3-A0).

`r?_tb0` Same as `R?_SET0` except that if the operation did not change the value (all selected bits were already zero), then the `ZERO` status bit is set. Also write enable to the register file is internally disabled.

`r?_tb1` Same as `R?_SET1` except that if the operation did not change the status the value (all selected bits were already one), then the `ZERO` status bit is set. Also write enable to the register file is internally disabled.

`r?_abs` Computes the absolute value of the S operand.

`r?_smtc` Signed magnitude/two's complement conversion of the S operand.

`r?_addi` Add the low four bits of the A register address to the S operand. Using this function, a constant from 0 to 15 can be added.

`r?_subi` Subtract the low four bits of the A register address from the S operand. Using this function, a constant from 0 to 15 can be subtracted.

`r?_badd` Computes R + S in selected bytes. Unselected bytes pass S unaltered. If the bytes are adjacent, carry flows between them.

`r?_bsubs` In the selected bytes, compute R - S. In unselected bytes, pass S unaltered. If bytes are adjacent, carry flows between them.

`r?_bsubr`	Computes S - R in selected bytes. Unselected bytes pass S unaltered. If the bytes are adjacent, carry flows between them.
`r?_bincs`	Computes S + 1 in selected bytes. Unselected bytes pass S unaltered. If the bytes are adjacent, carry flows between them.
`r?_bms`	Computes -S in selected bytes. Unselected bytes pass S unaltered. If the bytes are adjacent, carry flows between them.
`r?_bocs`	Computes ~S in selected bytes. Unselected bytes pass S unaltered.
`r?_bxor`	Computes S ^ R in selected bytes. Unselected bytes pass S unaltered.
`r?_band`	In the selected bytes, compute S & R. In unselected bytes, pass S unaltered.
`r?_bor`	Computes S | R in selected bytes. Unselected bytes pass S unaltered.
`r?_nop`	Output is zero (default).

8.5. Barrel Shifter (BS) Instructions

The barrel shifter performs left and right logical and arithmetic shifts, using a constant shift value or a variable from the count (CN) register. The constant shift value comes from a five-bit field in the instruction word. Using the barrel shifter with a constant value requires two steps:

1. Load barrel shifter input register (BR).
2. Shift and read the barrel shifter output.

The output of the barrel shifter is not registered, so it must be read in the same instruction cycle as the command that shifts it. Using a shift command does *not* affect input register BR.

CN register shifts operate exactly like constant shifts except that the shift count comes from the CN register instead of the instruction word. CN is not valid until two cycles after it is loaded. Since a right shift of zero produces unexpected results (see below), you may want to test the count when you are loading the CN register and branch around the shift when the count is zero.

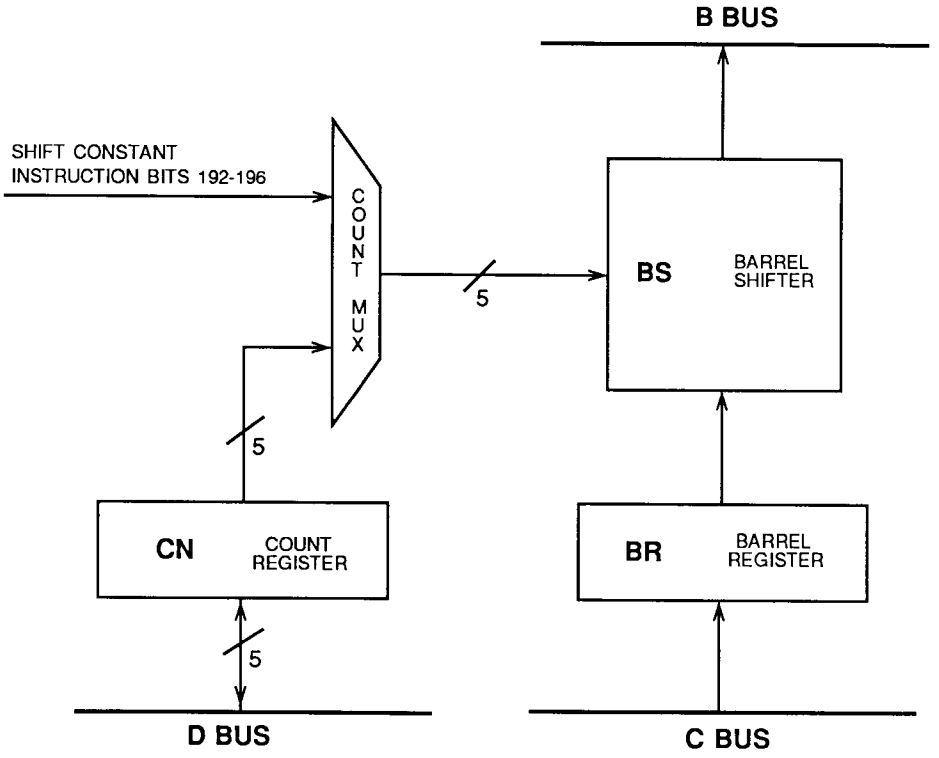
When the CN register is read, only the low five bits are defined. To get the actual count of CN, AND it with 0x1f.

The following instructions load the barrel register and CN register and read the barrel shifter output:

`c_??br`	load barrel register
`b_bs or b_bs??`	read output from barrel shifter
`d_??cn`	load CN register
`d_cn or d_cn??`	read CN register

Where `??` represents a valid source or destination. See the B, C, and D bus sections at the end of this chapter for a summary of valid sources and destinations.

Figure 8-5 *The Barrel Shifter (BS)*



Shift Using a Constant

`bs_sll,n`	Shift left logical n bits. Fill vacated bits with zeroes.
`bs_srl,n`	Shift right logical -n bits. Fill vacated bits with zeroes.
`bs_slc,n`	Shift left circular n bits.
`bs_sra,n`	Shift right arithmetic -n bits.

Example: Shift Using a Constant

Shift contents of RC7 >> 15:

```
rc_pr rc_rareg rc_a#7 c_rcbr        ;load barrel shifter
b_bs bs_srl,-15 rc_fbbus rc_c#7     ;write output to RC7
```

Shift Using the CN Register

`bs_sllcn`	Shift left logical CN bits. Fill vacated bits with zeroes.
`bs_srlcn`	Shift right logical -CN bits. Fill vacated bits with zeroes.
`bs_slccn`	Shift left circular CN bits.
`bs_sracn`	Shift right arithmetic -CN bits.

Example: Shift Using the CN Register

Shift contents of RC7 left by a variable number (stored in RD1):

```
rd_pr rd_rareg rd_a#1 d_rdcn        ;load CN with variable
                                    ;in RD1
rc_pr rc_rareg rc_a#7 c_rcbr        ;load barrel shifter
b_bs bs_sllcn rc_fbbus rc_c#7       ;write output to RC7 -
                                    ;NOTE CN not valid
                                    ;until this cycle
```

Interesting Effects Using the Barrel Shifter

`bs_srl,0`	Always outputs a zero. Useful as a zero constant generator.
`bs_sra,0`	Always outputs a -1 if BR is a negative number and 0 if BR is positive.

8.6. Multiplier/ Accumulator (MA) Instructions

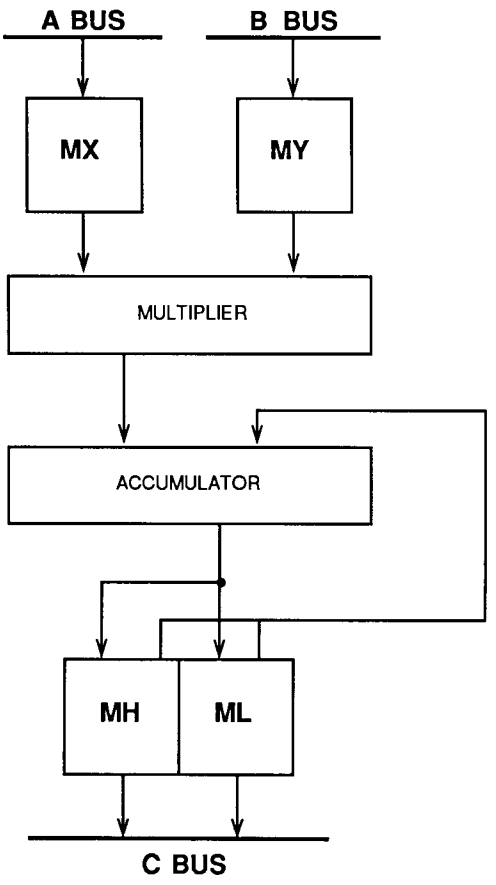
MA multiplies two 32-bit numbers. MA also passes one 32-bit number and stores it or adds it to a 64-bit accumulation register. MX and MY are operands, MH is the high 32 bits of the output, ML is the low 32 bits of output. Using the multiplier requires a minimum of three instructions:

1. Load MX and MY (these can be loaded in the same instruction)
2. Perform the operation

3. Read MH or ML

These instructions can be pipelined. MH and ML are valid until the next multiplier operation.

Figure 8-6 *The Multiplier/Accumulator (MA)*



Valid instructions to load the multiplier or read its output are:

`a_??mx`	load MX
`b_??my`	load MY
`c_ml or c_ml??`	read ML (low 32 bits of output)
`c_mh or c_mh??`	read MH (high 32 bits of output)

Where `??` represents a valid source or destination. See the A, B, and C bus sections at the end of this chapter for a summary of valid sources and destinations.

MA operations include:

`ma_pas`	MH/ML = MX
`ma_neg`	MH/ML = - MX
`ma_mpas`	MH/ML = MX * MY
`ma_mneg`	MH/ML = - (MX * MY)
`ma_madd`	MH/ML = MH/ML + (MX * MY)
`ma_msub`	MH/ML = MH/ML - (MX * MY)
`ma_msua`	MH/ML= (MX * MY) - MH/ML

MA does both signed and unsigned multiplication, with an unsigned default. If MX is signed, use the instruction `ma_xsign`. If MY is signed, use the instruction `ma_ysign`. Sign instructions must be expressed in addition to the operation.

The multiplier control fields are overlapped with the control fields for the floating point processor (FP), so that both processors cannot be active during the same instruction. The default for this field is `nomult`.

Example: Signed Multiply

Perform a signed multiply of RC1 by RC2 and put the results in RC12:

```
a_rcmx rc_a#1 b_rcmy rc_b#2
ma_mpas ma_xsign ma_ysign
c_ml rc_c#12 rc_fcbus
```

Example: Multiply/Accumulate

Multiply and accumulate RC5 = (RC1 * RC2) + (RC3 * RC4):

```
a_rcmx rc_a#1 b_rcmy rc_b#2
a_rcmx rc_a#3 b_rcmy rc_b#4\
ma_mpas ma_xsign ma_ysign
ma_madd ma_xsign ma_ysign
c_ml rc_c#5 rc_fcbus
```

8.7. Lookup Table (LU) Instructions

The lookup table is a set of ROMs and RAMs used to store precomputed values that accelerate some calculations. The ROMs are used by the math subroutines provided with the TAAC-1 and are generally not useful for other programs. See the mathematical functions for examples, in the TAAC-1 library chapter.

The RAMs are very useful for application-dependent calculations. The low 13 bits of LR are an address into the 8K by 32-bit RAM, which can be read or written through LT. To write to the lookup table RAM, load the address into the LR. On the next cycle (or a subsequent cycle),

write the value to LT. To read from the lookup table RAM, load the address into the LR; the result is available from LT on the next cycle. Results remain valid until the LR is changed.

The following instructions load or read the lookup tables:

`c_??lr`	load lookup table address
`b_??lt`	load lookup table RAM (LT)
`b_lu or b_lu??`	read lookup table ROM (LU)
`b_lt or b_lt??`	read lookup table RAM (LT)

Where ?? represents a valid source or destination. See the B and C bus sections at the end of this chapter for a summary of valid sources and destinations.

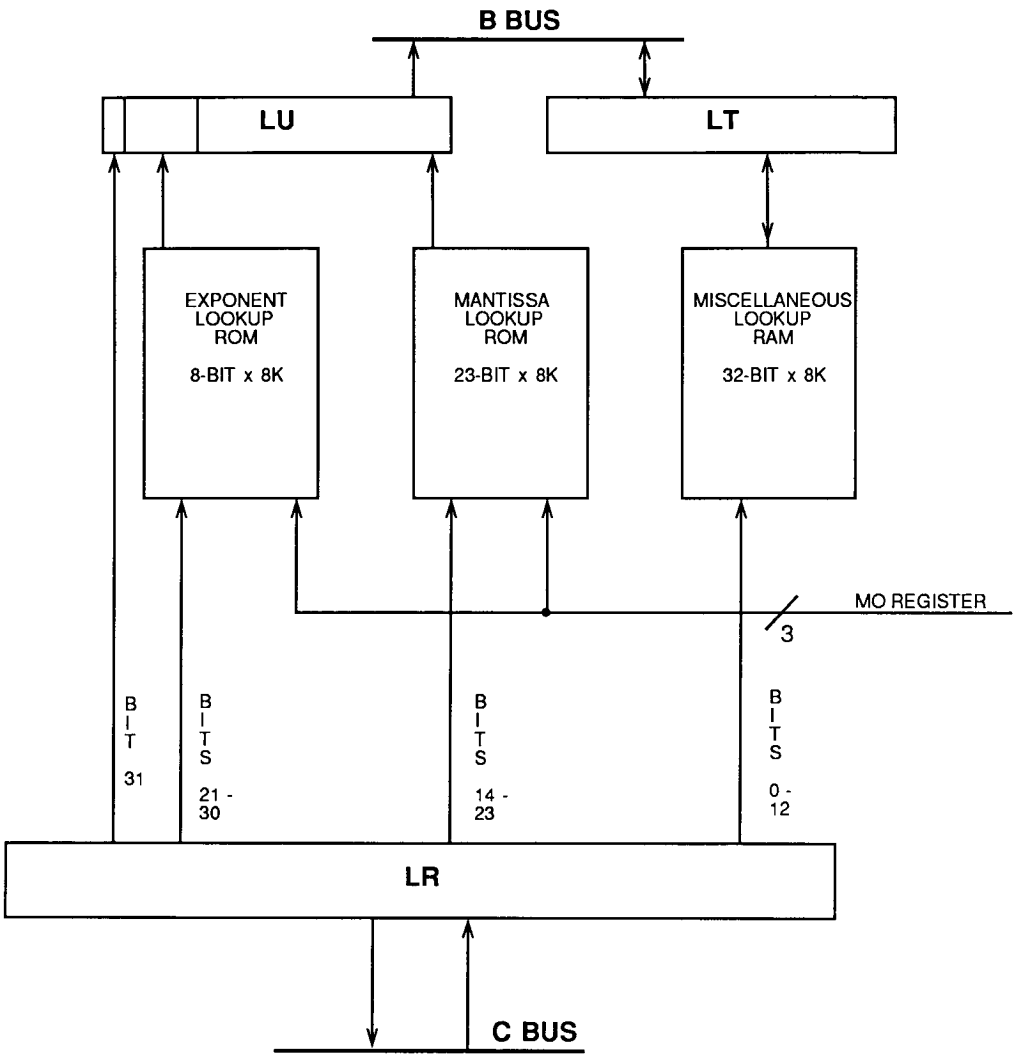
Example: Write to Lookup Table RAM

```
rc_pr rc_rareg rc_a#5 c_rclr    ;load LR with address in RC5
rc_b#7 b_rclt                   ;load LT with value in RC7
```

Example: Read from Lookup Table RAM

```
rc_pr rc_rareg rc_a#5 c_rclr    ;load LR with address in RC5
b_lt rc_fbbus rc_c#6            ;write LT value to RC6
```

Figure 8-7 *The Lookup Table*



8.8. Floating Point Processor (FP) Instructions

The Floating Point Processor (FP) performs both single and double precision ALU and multiplication operations. The ALU and multiplication operations can be independent or simultaneous. To use the Floating Point Processor, it is important to understand that:

- The FP is clocked only in cycles containing a floating point instruction.

- When you load the FX or FY input registers, you must clock the Floating Point Processor by specifying a floating point operation in the same cycle. For example, the instruction:

```
rc_a#0 a_rcfx rc_b#1 b_rcfy
```

  would not load FX and FY. A correct instruction might contain:

```
rc_a#0 a_rcfx rc_b#1 b_rcfy fp_mul
```

- Each time you clock the FP, it latches the floating point instruction for the next cycle, from the next instruction to be executed. This means that a floating point multiply and/or ALU operation must be directly preceded by a floating point operation (of any kind), which latches the instruction for the multiply and/or ALU operation into the FP. For example, the sequence of instructions:

```
rc_a#0 a_rcfx rc_b#1 b_rcfy fp_mul  ;load FX and FY
cont
fp_mul fp_max fp_mby                ;multiply
```

  would produce undefined results, because the multiply operation was not directly preceded by a floating point operation of some kind.

- Every time the FP is clocked, it destroys the SUM and PRODUCT registers, as well as the FH and FL output registers. If a multiply operation is executed by itself (for example, `fp_mul`), the contents of the SUM register are undefined. Similarly, if an ALU operation is executed by itself (for example, `fp_add`), the contents of the PRODUCT register are undefined. The instruction:

```
fp_paspasa  fp_mas fp_aap
```

  swaps the SUM and PRODUCT registers and can be used to keep the contents of these registers from being destroyed.

To use the Floating Point Processor, follow these steps:

1. Load FX and FY. These registers can be loaded in the same cycle or separate cycles. Specify a floating point operation in the same instruction, to clock the input into the FX and/or FY registers.
2. Perform the floating point operation. Because the first floating point operation in a series of contiguous floating point operations produces undefined results, this instruction must be *directly*

preceded by an instruction containing a floating point operation (of any kind). If this is a simultaneous ALU/multiplier operation, this instruction must also specify whether the sum or product should be written to the output registers FL and FH.

3. Read the result in FL or FH. For single-precision operations, the result is contained in FH. For double-precision operations, FL contains the low 32 bits of the result; FH contains the high 32 bits. The results are also contained in the SUM and/or PRODUCT registers.

The following instructions load the FX and FY registers and read the result:

`a_??fx`		`<fp-op>`	load FX register
`b_??fy`	or	`<fp_op>`	load FY register
`c_fh`	or	`c_fh??`	read FH - high 32 bits of result (entire result for single precision operations)
`c_fl`	or	`c_fl??`	read FL - low 32 bits of result

Where ?? represents a valid source or destination. See the A, B, and C bus sections at the end of this chapter for a summary of sources and destinations. <fp_op> represents the floating point operation that must be specified to load the FX and FY registers.

Double Precision Operations

To perform double precision operations, it is important to understand that:

- when you load the FX and FY registers, load the high half (upper 32 bits) of the double precision word first, by sourcing them onto the A and/or B buses. Do not specify fx or fy as a destination, but do specify a dummy floating-point operation (single precision, such as fp_mul).
- load the low half of the double precision word second, specifying fx and/or fy as a destination, and specifying a dummy floating-point operation.
- preceding every double precision operation, there must be *two* cycles that contain floating-point operations. (These cyles may be the FX,FY load cycles, dummy operations, or other floating-point operations.) The second of these operations must be a double

- precision operation.
- in order for a double precision multiply (fp_dmul) to complete, it must be followed by a dummy no-op cycle; using fp_mul is sufficient.
- in chained or simultaneous operations, specifically fp_dpaspasa, both the sum and product paths need the no-op delay after the operation, not just the product path.

Refer to the Floating Point Processor Instructions earlier in this section for more information.

The following sequence of instructions performs a double precision multiply:

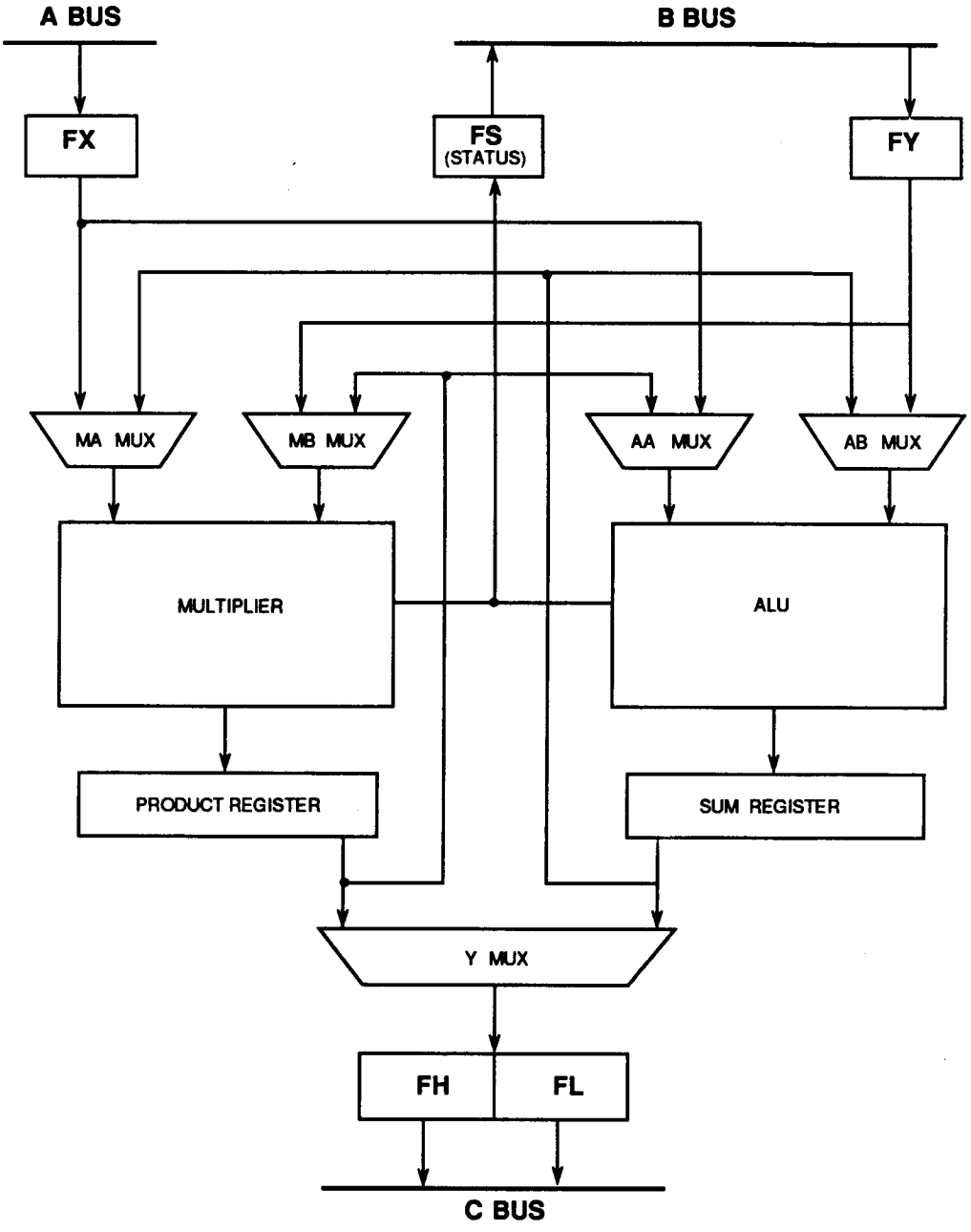
```
a_rc rc_a#OP1_HI b_rc rc_b#OP2_HI fp_mul fp_max fp_mby
                                  ;load high 32 bits of X and Y
                                  ;dummy operation (single)
a_rcfx rc_a#OP1_LO b_rcfy rc_bOP2_LO fp_dmul fp_max fp_mby
                                  ;load low 32 bits of X and Y
                                  ;dummy operation (double)
fp_dmul fp_max fp_mby             ;multiply
fp_mul fp_max fp_mby              ;allow multiply to complete
c_fh rc_fcbus rc_c#PROD_HI        ;store product high 32 bits
c_fl rc_fcbus rc_c#PROD_LO        ;store product low 32 bits
```

As another example, the following instruction sequence performs a double-to-integer conversion:

```
a_rc rc_a#OP1_HI fp_mul fp_max fp_mby
                                  ;load high 32 bits of operand,
                                  ;dummy floating-pt operation
a_rcfx rc_a#OP1_LO fp_dmul fp_max fp_mby
                                  ;load low 32 bits of operand
                                  ;dummy floating-pt operation
                                  ;(double precision)
fp_dinta fp_aax                   ;convert to integer
c_fh rc_fcbus rc_c#RESULT         ;store result from sum
                                  ;register (high 32 bits)
```

For another example, refer to `ddiv.asm`, the source code for the double precision divide routine in the runtime library.

Figure 8-8 *The Floating Point Processor (FP)*



Independent Single-Operand ALU Operations

Single Precision

`fp_absa`	FH = SUM = |AA|
`fp_pasa`	FH = SUM = AA
`fp_nega`	FH = SUM = -AA
`fp_flta`	FH = SUM = (float)AA (AA is an integer)
`fp_inta`	FH = SUM = (int)AA (AA is a float)
`fp_sgla`	FH = SUM = (float)AA (AA is a double)
`fp_wrap`	FH = SUM = wrap(AA)
`fp_unwe`	FH = SUM = unwrap(AA) (AA is exact)
`fp_unwi`	FH = SUM = unwrap(AA) (AA is inexact)
`fp_unwr`	FH = SUM = unwrap(AA) (AA is rounded)

Double Precision

`fp_dabsa`	FH/FL = SUM = |AA|
`fp_dpasa`	FH/FL = SUM = AA
`fp_dnega`	FH/FL = SUM = -AA
`fp_dflta`	FH/FL = SUM = (double)AA (AA is a float)
`fp_dinta`	FH = SUM = (int)AA (AA is a double)
`fp_dbla`	FH/FL = SUM = (double)AA (AA is an integer)
`fp_dwrap`	FH/FL = SUM = wrap(AA)
`fp_dunwe`	FH/FL = SUM = unwrap(AA) (AA is exact)
`fp_dunwi`	FH/FL = SUM = unwrap(AA) (AA is inexact)
`fp_dunwr`	FH/FL = SUM = unwrap(AA) (AA is rounded)

Independent Double-Operand ALU Operations

Absolute value operations can be combined in the same instruction with independent double-operand ALU operations.

Single Precision

`fp_add`	FH = SUM = AA + AB
`fp_suba`	FH = SUM = AB - AA
`fp_subb`	FH = SUM = AA - AB
`fp_cmp`	sets compare condition codes
`fp_aabs`	AA = |AA|
`fp_babs`	AB = |AB|

Double Precision

`fp_dadd`	FH/FL = SUM = AA + AB
`fp_dsuba`	FH/FL = SUM = AB - AA
`fp_dsubb`	FH/FL = SUM = AA - AB

`fp_dcmp`	sets compare condition codes
`fp_aabs`	AA = |AA|
`fp_babs`	AB = |AB|

Independent Multiplier Operations

Absolute value operations can be combined in the same instruction with independent multiplier operations.

Single Precision

`fp_mul`	FH = PRODUCT = MA * MB
`fp_mulwra`	FH = PRODUCT = MA * MB (MA is wrapped)
`fp_mulwrb`	FH = PRODUCT = MA * MB (MB is wrapped)
`fp_aabs`	MA = |AA|
`fp_babs`	MB = |AB|

Double Precision

`fp_dmul`	FH = PRODUCT = MA * MB
`fp_dmulwra`	FH = PRODUCT = MA * MB (MA is wrapped)
`fp_dmulwrb`	FH = PRODUCT = MA * MB (MB is wrapped)
`fp_aabs`	MA = |AA|
`fp_babs`	MB = |AB|

Simultaneous ALU and Multiplier Operations with Selectable Output

`fp_outp` and `fp_outs` determine which output (sum or product) is passed to the FH/FL registers. It must be specified in the same instruction as a simultaneous ALU and multiplier operation.

Single Precision

`fp_muladd`	PRODUCT = MA * MB, SUM = AA + AB
`fp_mulsuba`	PRODUCT = MA * MB, SUM = AB - AA
`fp_mulsubb`	PRODUCT = MA * MB, SUM = AA - AB
`fp_mulnega`	PRODUCT = MA * MB, SUM = -AA
`fp_mulsub2`	PRODUCT = MA * MB, SUM = 2 - AA
`fp_mulpasa`	PRODUCT = MA * MB, SUM = AA
`fp_pasadd`	PRODUCT = MA, SUM = AA + AB
`fp_passuba`	PRODUCT = MA, SUM = AB - AA
`fp_passubb`	PRODUCT = MA, SUM = AA - AB
`fp_paspasa`	PRODUCT = MA, SUM = AA
`fp_pasnega`	PRODUCT = MA, SUM = -AA
`fp_passub2`	PRODUCT = MA, SUM = 2 - AA

Double Precision

`fp_dmuladd`	PRODUCT = MA * MB, SUM = AA + AB
`fp_dmulsuba`	PRODUCT = MA * MB, SUM = AB - AA
`fp_dmulsubb`	PRODUCT = MA * MB, SUM = AA - AB
`fp_dmulnega`	PRODUCT = MA * MB, SUM = -AA
`fp_dmulsub2`	PRODUCT = MA * MB, SUM = 2 - AA
`fp_dmulpasa`	PRODUCT = MA * MB, SUM = AA
`fp_dpasadd`	PRODUCT = MA, SUM = AA + AB
`fp_dpassuba`	PRODUCT = MA, SUM = AB - AA
`fp_dpassubb`	PRODUCT = MA, SUM = AA - AB
`fp_dpaspasa`	PRODUCT = MA, SUM = AA
`fp_dpasnega`	PRODUCT = MA, SUM = -AA
`fp_dpassub2`	PRODUCT = MA, SUM = 2 - AA

Output Select

`fp_outp`	FH/FL = PRODUCT
`fp_outs`	FH/FL = SUM

Input Multiplexer Controls

`fp_max`	MA = X input
`fp_mas`	MA = SUM register
`fp_mby`	MB = Y input
`fp_mbp`	MB = PRODUCT register
`fp_aax`	AA = X input
`fp_aap`	AA = PRODUCT register
`fp_aby`	AB = Y input
`fp_abs`	AB = SUM register

Example: Add RD1 to RD2 (single precision)

```
a_rdfx rd_a#1 b_rdfy rd_b#2 fp_mul  ;load RD1 and RD2
fp_add fp_aax fp_aby               ;add
c_fh e_ec d_ed rd_fdbus rd_c#3     ;store result in RD3
```

*Example: Multiply/ Accumulate RC1 * RC2 + RC3 * RC4 (sgl precision)*

```
a_rcfx rc_a#1 b_rcfy rc_b#2 fp_mul  ;load RC1 and RC2
a_rcfx rc_a#3 b_rcfy rc_b#4 fp_mul fp_max fp_mby
                                    ;multiply RC1*RC2,
                                    ;load RC3 and RC4
fp_max fp_mby fp_aap fp_mulpasa     ;multiply RC3*RC4,
                                    ;SUM=PRODUCT
fp_abs fp_aap fp_add                ;PRODUCT=
                                    ;SUM+PRODUCT
c_fh rc_fcbus rc_c#5                ;store result in RC5
```

FP Status Register

The floating point status word is available on the B bus in the cycle after each operation. The next table contains FP status bit definitions.

Table 8-1 *Floating Point Status Word Bit Definitions*

Bit	*Definition*
0	A NaN has been input to the multiplier or the ALU, or an invalid operation has been requested.
1	The result of an operation is inexact.
2	Overflow
3	Underflow
4	The multiplier output is a wrapped number or the ALU output is a denorm.
5	The input to the multiplier is a denorm.
6	The mantissa of a wrapped number has been altered due to rounding.
7	Status was generated by multiplier.
8	reserved
9	A NaN or a denorm has been input from the B bus.
10	A NaN or a denorm has been input from the A bus.

8.9. Memory Access

TAAC-1 memory access runs through vector ports VA and VB, for sequential DRAM access, or through the Data Read and Data Write registers (DR and DW), for random access. The TAAC-1 has two address registers. The AI register (Address Immediate) accepts only 1D addresses. The AC register (Address Count) allows 1D, 2D, or 3D addressing.

Random Access Using the AI Register

The insructions for reading and writing using the AI register are:

`mreadai`	read at address stored in AI register
`mwriteai`	write to address stored in AI register

To read from a particular address, load the AI register with an address and use the `mreadai` instruction. The value addressed goes to Data Read Register DR.

To write to an address, load the AI register with the address and use

the `mwriteai` instruction.

Example: Read Word at Address 0x17456

```
a_df e_ea d_edai 0x17456          ;load AI with 0x17456
mreadai                           ;read addressed value to DR
e_dr a_ea rc_c#12 rc_fabus        ;store value in RC12
```

Example: Write Contents of RC12 to Address Contained in RD0

```
rd_a#0 rd_pr d_rdai rc_b#12 b_rc e_ebdw
                                  ;AI=RD0,DW=RC12
mwriteai                          ;write
```

Random Access Using the AC Register

The Address Count Register (AC register) allows random memory access in 1, 2, or 3 dimensions. AC address reads and writes use three steps:

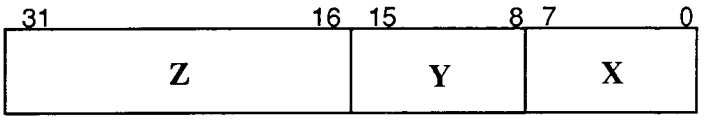
1. Load, increment, or decrement AC register.
2. Use the `macp` instruction to calculate new address, based on the addressing mode set in the AM register. `macp` uses the contents of the AC register at the end of the previous cycle to generate an effective address, which it stores in the Address Count Pipeline (ACP) register.
3. Write or read, using effective address generated by `macp`. DW contains the value to be written; DR receives the data read.

These instructions need not be consecutive, but they must occur in this order. They can be pipelined, to allow single-instruction reads or writes. (See the second example at the end of this section.)

AC has its own instructions for reading the D bus, allowing separate reads of different bytes on the bus. Instructions `acxld`, `acyld`, and aczld enable reads into the **X**, **Y**, and **Z** fields of the Address Register/Counter, as shown in the next diagram.

NOTE: **X**, **Y**, and **Z** are *only field symbols* and are *not* the same as x, y, and z addresses.

Figure 8-9 *Format of AC Register Fields*



The following instructions are available for AC reads and writes:

`mreadac`	read at address stored in AC register
`mwriteac`	write to address stored in AC register
`macp`	generate effective address from AC register
`acxld`	load **X** field of AC register
`acyld`	load **Y** field of AC register
`aczld`	load **Z** field of AC register
`acxi`	increment **X** field of AC register
`acyi`	increment **Y** field of AC register
`aczi`	increment **Z** field of AC register
`acxd`	decrement **X** field of AC register
`acyd`	decrement **Y** field of AC register
`aczd`	decrement **Z** field of AC register

The next table summarizes instructions used to load, increment, and decrement the AC register.

Addressing Modes

Bits 10 and 11 of the DRAM Mode Register (AM register) set the addressing mode for the AC register:

0 3-dimensional (slice)
1 3-dimensional (dice)
2 2-dimensional
3 1-dimensional

One-Dimensional Addressing

One-dimensional addressing passes the address without change.

Table 8-2 *Summary of AC Register Instructions*

Function	*D Bus*	*Instructions*

1D Addressing
Load address
Increment address
Decrement address
	1D address	`acxld acyld aczld`
		`acxi acyi aczi`
		`acxd acyd aczd`

2D Addressing
Load x address
Load y address
Increment x address
Increment y address
Decrement x address
Decrement y address
	x address
	y address << 16
		`acxld acyld`
		`aczld`
		`acxi acyi`
		`aczi`
		`acxd acyd`
		`aczd`

3D Addressing
Load x address
Load y address
Load z address
Increment x address
Increment y address
Increment z address
Decrement x address
Decrement y address
Decrement z address
	x address
	y address << 8
	z address << 16
		`acxld`
		`acyld`
		`aczld`
		`acxi`
		`acyi`
		`aczi`
		`acxd`
		`acyd`
		`aczd`

NOTE: Instructions can be combined to affect all three AC fields simultaneously.

Two-Dimensional Addressing

Two-dimensional addressing rearranges the address to accommodate pixel addressing:

zzzyyzzzzzzzzxxxxxxxx
(3) (2) (8) (8)

Where z = low eleven bits of **Z**
y = low two bits of **Y**
x = all eight bits of **X**

In 2D mode, the **Z** field holds the y coordinate and **X/Y** field the x coordinate. There are effectively 10 bits of spatial x and 11 bits of spatial y resolution (1024 x 2048).

Three-Dimensional Addressing

3D "dice mode" addressing rearranges addresses to allow efficient use of memory for 3D cubes:

zyxzyxzyxzyxzyxzyxzyx

Where x = low seven bits of **X**
y = low seven bits of **Y**
z = low seven bits of **Z**

The next table helps you build and size cubes for your application.

Table 8-3 *Video Interpretation of "Dice Mode" Data Cubes*

Cube Size	*X Size*	*Y Size*	*Total Words*	*Number of Cubes available*
2	8	1	8	256K
4	64	1	64	32K
8	256	2	512	4K
16	256	16	4K	512
32	256	128	32K	64
64	1024	256	256K	8
128	1024	2048	2048K	1

In 3D "slice mode" addressing, the **X**, **Y**, and **Z** fields of the AC register are loaded in the same manner as "dice mode," but the address is passed through for memory access without bit reordering. This mode is most useful in indexing through 256 x 256 images, such as CT scan data.

Timing of Random Memory Access

The number of cycles it takes a read or write to complete depends on the address register being used (AI or AC) and the memory type being accessed.

- When the AI register is used, scratchpad memory (SRAM) reads and writes take one cycle to complete; data/image memory (DRAM) accesses require three processor cycles.
- When the AC register is used, SRAM access again takes one cycle

to complete. For DRAM reads or writes within the same four-word block (when all but the low two address bits are the same), all accesses after the first take one cycle to complete. For DRAM reads or writes within the same 1024-word block (all but the low 10 address bits are the same), all accesses after the first take two cycles to complete. All other random DRAM reads and writes take three processor cycles.

Addressed data is valid in the Data Read Register (DR) one instruction (not necessarily one processor cycle) after the read. If you are reading from DRAM and use DR in the next instruction, the TAAC-1 stops the processor clock until the read has actually completed. Otherwise there is no delay unless the next instruction contains another memory access. That is to say, while the read is taking place, the processor can be performing other instructions, such as computations, so long as these instructions do not involve DR or another memory access.

An sqwait instruction used in conjunction with the read forces the processor to wait for completion of the read before executing another instruction. sqwait data-read instructions look like this:

```
a_df e_ea d_edai 0x17456              ;load AI with 0x17456)
mreadai sqwait e_dr a_ea rc_c#12 rc_fabus
                                      ;read value into RC12
```

Two conditions cause a processor delay after a write:

- The write will not complete in the current cycle, and the current instruction changes the DW register, the address registers (AI or AC), or the address mode (AM) register.
- The operation will not complete in the current cycle, and the next instruction contains another memory access.

`sqwait` has no effect on writes.

Example: Write a Pixel

Write a pixel to DRAM using the AC register and 2D addressing (x address stored in RD2, y address stored in RD1). Note that before every write (or read) to a new address, you must use the `macp` instruction to latch the address into the ACP.

```
#include <taac1/builtin.h>
#define RED 0xff
```

```
d_am rd_fdbus rd_c#0          ;RD0 = AM register contents
rd_and a_df rd_rabus rd_sbreg rd_b#0 rd_fyout rd_c#0\
   TA_AMDIM_MASK              ;mask out 1D/2D/3D bits
rd_or d_rdam a_df rd_rabus rd_sbreg rd_b#0 TA_AM2D
                              ;set AM reg. for 2D mode
   .
   .
a_rd c_ecbr e_ea rd_a#1
rd_or rd_sbbus b_bs bs_sll,0x10 rd_rareg rd_a#2\
   acxld acyld aczld          ;load y<<16 | x
a_df a_eadw RED\              ;load DW with shade
   macp                       ;latch address into ACP
mwriteac                      ;write pixel
```

Example: Single-Instruction Loop

Write a line of 100 pixels. The x address is again stored in RD2, y address in RD1.

```
#include <taac1/builtin.h>
#define RED 0xff

;(set AM register for 2-D mode as above)
      .
      .
   a_rd rd_a#1 e_ea c_ecbr
   rd_or rd_sbbus b_bs bs_sll,0x10 rd_rareg rd_a#2\
      acxld acyld aczld\
         f_sd ldra 100            ;load y<<16 | x address
                                  ; and loop count
   a_df e_eadw RED\               ;load data word w/shade
      macp\                       ;latch next address
         acxi acyi                ;increment to next addr+1
XLOOP:
   mwriteac\                      ;write pixel
      macp\                       ;latch next address
         acxi acyi\               ;increment to next addr+1
            f_sd juca cctrue XLOOP ;loop back
```

8.10. Addressing Memory with the Vector Ports

The vector ports are used for fast sequential access to DRAM memory. DRAM memory is divided into 1024-word segments called *pages*. When you are reading from DRAM, a single read loads the serial shift register with the contents of a page of memory. Then, from the vector ports, the TAAC-1 processor has access to a new 32-bit word from

the shift register on every processor cycle. The shift register can be loaded only on page (1024-word) boundaries, but the starting address you specify sets a pointer into the shift register, on a four-word boundary.

Writing to DRAM, you can load the shift register, via the vector ports, with a new word every processor cycle. Then a single write writes the entire contents of the shift register to DRAM memory.

Instructions to write to or read from the vector ports are:

`vast`	start VA (vector port A) read or write operation
`vard`	read from shift register to VA
`vawr`	write from VA to shift register
`a_va??`	VA is source on A bus
`a_??va`	VA is destination on A bus
`vbst`	start VB (vector port B) read or write operation
`vbrd`	read from shift register to VB
`vbwr`	write from VB to shift register
`b_vb??`	VB is source on B bus
`b_??vb`	VB is destination on B bus

Reading Data through the Vector Ports

1. Load DRAM Mode Register AM bits 13 and 14 with 1 for *shift register load* mode. The `builtin.h` include file defines this bit setting as `TA_AMSRD`.
2. Do a read from the starting address. The lower two bits of this address are ignored, placing the effective address on a four-word boundary. This read loads the shift register from the data/image memory.
3. Change AM register bits 13 and 14 back to 0, for random access mode, if needed.
4. Do a read from another page in data/image memory, to ensure that the shift register has been completely loaded.
5. To read data from bank A:

   Use vast to start the read from VA
   Do a second `vast` (for hardware reasons)
   The first word is now available in VA. Use `a_va` as a source, and `vard` to read the next word, until the end of the page (1024-word boundary).

To read data from bank B:
Use `vbst` to start the read from VB
Do a second `vbst`
The first word is now available in VB. Use `b_vb` as a source, and `vbrd` to read the next word, until the end of the page.

6. Return to step one as needed.

See the next example and the clahe/map demo source code for examples of vector port reads.

Writing Data through the Vector Ports

1. Load AM register bits 13 and 14 with a 3 (`TA_AMSIN`), for *serial input* mode.
2. Write to the desired address. This sets the starting address. The lower two bits of this address are ignored, producing an effective address on a four-pixel boundary.
3. Change AM register bits 13 and 14 back to 0, random access mode, if needed.
4. To write to bank A:
   Write the first word of data to the VA register.
   Use `vast` for the first write and load the VA register with the second word of data.
   Then use `va` as a destination and `vawr` as needed until the end of the page.

   To write to bank B:
   Write the first word of data to the VB register.
   Use `vbst` for the first write and load the VB register with the second word of data.
   Then use `vb` as a destination and `vbwr` as needed until the end of the page.
5. Load AM register bits 13 and 14 with a 2 (`TA_AMSWR`), for *shift register store* mode.
6. Do a write to the desired address. This writes the entire contents of the shift register to data/image memory. Note that if you do not want to change the entire page of memory, you must first load the shift register (using a serial read) from image memory, then use the vector port to write to the desired addresses, then write the entire page back to DRAM.
7. Return to step one as needed.

Refer to the `t_erase` function in the TAAC-1 library for an example of vector port writes.

Example: Vector Port Reads

Read 100 words from Bank B using vector port B and write them to Bank A (using the AC register) starting at address 0:

```
#include <taac1/builtin.h>
#define SRC 0x100000            /* data source */
#define DST 0x0                 /* data destination */
#define LPCOUNT 100             /* count */

;load shift register from DRAM
   a_df TA_AMSRD e_ea d_edam    ;set AM for serial read
   a_df SRC e_ea d_edai
      mreadai\                  ;shift register <- DRAM
      a_df TA_AM1D e_ea d_edam  ;set AM register for 1D

;read from another page to make sure shift register load is
;done
   a_df 0 e_ea d_edai
   mreadai\
      a_df DST e_ea d_ed acxld acyld aczld
                                ;AC register <- dest address

;begin vector port operation
   vbst\                        ;start read
      macp\                     ;latch address into ACP
         acxi acyi aczi         ;increment address
   vbst\                        ;second vbst needed
      f_sd ldra LPCOUNT-1       ;load loop counter
   vbrd b_vb e_ebdw             ;DW <- first word, read next
READLOOP:
   vbrd b_vb e_ebdw\            ;DW <- VB, read next
      mwriteac\                 ;write to destination
         macp\                  ;latch new dest address
            acxi acyi aczi\
               juca cctrue READLOOP
                                ;increment to next src
                                ;address and loop back
DONE:
   mwriteac                     ;write last word
```

8.11. DRAM Mode Register

The DRAM Mode (AM) Register controls bitmask and wordmask selection and enable, addressing modes for the AC register, data bounds checking (hardware z-buffering), and the access mode for

data/image memory. Bit assignments are shown in the next table. Refer to the hardware chapter for a further description of these functions. The include file `<taac1/taregdefs.h>` contains contant definitions for these fields.

Table 8-4 *Bit Assignments in DRAM Mode Register AM*

Bits	*Mode Definition*
0-3	Word mode mask for random writes to DRAM (1 = write enable) Channel mode mask for vector port writes to DRAM (1 = channel enable)
4-7	Bitplane mask select
8	Enable word/channel mode mask
9	Enable bitplane masked write
10, 11	Addressing Mode 0 = 3D Addressing - "Slice" Mode 1 = 3D Addressing - "Dice" Mode 2 = 2D Addressing 3 = 1D Addressing
12	Write enable Data Bounds Checking, when 1, write if DW <= DR
13, 14	DRAM Access Mode 0 = random access 1 = shift register load (DRAM → shift register) 2 = shift register store (shift register → DRAM) 3 = enable serial write (processorshift → register)
15	1 = disable read into DR

8.12. Miscellaneous Mode Register

The Miscellaneous Mode (MO) register controls:

- the stride (address increment amount) for the vector ports
- RC and RD modes (byte, halfword, word)
- modes for the floating-point processor (FP)
- rounding modes for the integer multiplier/accumulator
- lookup table functions

Table 8-5 *Bit Assignments in Mode Register MO*

Bit Number	*Register Values*	*Assignment*
0, 1	0	VA stride (address increment amount) of 4
	1	stride of 1 (default)
	2	stride of 2
	3	stride of 3
2, 3	0	VB stride of 4
	1	stride of 1 (default)
	2	stride of 2
	3	stride of 3
4 - 6	0	RD configuration, byte mode, status from byte 0
	1	byte mode, status from byte 1
	2	byte mode, status from byte 2
	3	byte mode, status from byte 3
	4	halfword mode, status from low halfword
	5	halfword mode, status from high halfword
	6	word mode (default)
	7	reserved
7 - 9	0	RC configuration, byte mode, status from byte 0
	1	byte mode, status from byte 1
	2	byte mode, status from byte 2
	3	byte mode, status from byte 3
	4	halfword mode, status from low halfword
	5	halfword mode, status from high halfword
	6	word mode (default)
	7	reserved
10		FP clock mode (default = 0)
11		FP fast mode (default = 1)
12, 13	0	FP round mode, towards nearest
	1	round towards zero (default)
	2	round towards infinity
	3	round towards negative infinity
14, 15		FP configuration (default = 1)
16 - 18	none	reserved
19, 20	0	MA round mode, no rounding (default)
	1	round using bit 30
	2	round using bit 31
	3	round using bits 30 and 31
21 - 23	0	Lookup Table function, 1 / LR (default)
	1	SQRT(LR)
	2	1 / SQRT(LR)
	3	SIN(LR) (fixed point)
	4	COS(LR) (fixed point)
	5	I / LR (fixed point)
	6-7	reserved

NOTE: MO default is defined in `taregdefs.h` as `TA_MO_DEFLT`.

RC and RD modes control carry between bytes or halfwords. Byte mode disables carry between bytes and controls which byte the status will come from. Halfword mode disables carry between halfwords and controls which halfword the status will come from.

The hardware overview chapter provides a fuller description of the MO register. Bit assignments are shown in preceding table. The include file `<taac1/taregdefs.h>` contains constant definitions for these fields.

8.13. Data Flow

In general, all bus sources drive all bus destinations, either on one bus or between different buses through the E bus. Data flow over a bus is specified by:

```
<bus>_ssdd
```

Or:

```
<bus>_ss
```

Where <bus> is the bus name (`a`, `b`, `c`, `d`, `e` or `f`), `ss` represents the source, and `dd` represents the destination. Operations without destinations use `<bus>_ss`.

Data Path Restrictions

The rules on data patch restrictions for wide instruction word architectures are quite complex. With its multitude of processor buses and processor elements, the TAAC-1 is no exception. Rather than define a complete specification of worst-case hardware timings (processor propagation times, register setup times, gate delays, etc.) that might equate into a complex set of programming rules, the TAAC-1's data flow restrictions can be summarized by the following conservative guidelines:

- Operands for integer ALUs RC and RD should be sourced only from registers on the A, B, or E buses, or from the register file.
- Destinations for integer ALUs RC and RD should be limited to registers on the C and D buses, respectively, or the E bus.

The TAAC-1 C compiler follows these rules. A slightly more aggressive policy permits any register-to-register data path, as long as the data passes through no more than one integer ALU. It is possible, but not recommended, to create even longer data paths that may seemingly provide correct numerical results, but fail in obscure

ways. For example, the processor might fail to take a conditional branch properly, because of a setup time violation of the condition code register in the previous, long data path instruction.

Registered and Unregistered Paths

- EA, EB, EC, and ED are not registers, so the instruction:

```
a_df e_ea d_edam
```

  moves data from the data field to DRAM Mode Register AM.

- FB is not a register, so the instruction:

```
f_sd d_fb acxld acyld
```

  loads the **X** and **Y** fields of the AC register with the low 16 bits of the sequencer.

- SF is a register, so the instruction:

```
e_dr d_edsf f_sf jufb cctrue
```

  does not jump to the contents of DR. This requires two instructions:

```
e_dr d_edsf                    ;SF=DR
f_sf jufb cctrue
```

- LR is registered but LU and LT are not, so using the lookup table requires two instructions:

```
e_dr c_eclr                    ;LR=DR
b_lu rc_fbbus rc_c#23          ;RC23= LU
```

- The ALUs don't need to be registered, so this instruction is valid:

```
a_va b_vb rd_rabus rd_sbbus rd_add d_rd e_eddw
                               ;DW= VA+VB
```

- BR is registered, but BS is not, so using the barrel shifter requires two instructions:

```
e_dr c_ecbr                    ;BR=DR
b_bsvb bs_sll,5                ;VB=BS= BR << 5
```

- The CN register must be loaded two instructions before a shift, so a variable shift requires 3 instructions:

```
e_dr d_edcn                    ;CN=DR
a_df e_ea c_ecbr               ;BR=DF
b_bsvb bs_sll,5                ;VB=BS = BR << 5
```

8.14. The A Bus

Sources

VA	Vector Port A. Useful for reading sequential data in the lower bank of video memory. Once started, this port can read one 32-bit word per instruction without delay for 1024 words.
DF	Data Field. 32-bit data from the current instruction.
EA	E bus to A bus transceiver. Transfers 32-bit values to and from the E bus.
RC	Registered Accumulator C. Grabs the value in the A register of RC specified by `rc_a#n`.
RD	Registered Accumulator D. Grabs the value in the A register of RD specified by `rd_a#n`.

Explicit Destinations

VA	Vector Port A. Useful for writing sequential data in the lower bank of video memory. Once started, this port can write one 32-bit word per instruction without delay for 1024 words.
FX	Floating Point Processor X Operand.
MX	Multiplier/Accumulator X Operand.

Implicit Destinations

EA	E bus reads A bus using `e_ea??` as a source.
RC	Registered Accumulator C can grab the A bus for operations using `rc_rabus` and write the A bus to a register using `rc_fabus`.
RD	Registered Accumulator D can grab the A bus for operations using `rd_rabus` and write the A bus to a register using `rd_fabus`.

Most combinations of source and destination are valid on the A bus. The only exception is a source VA with a destination VA, which makes no sense. As a rule, the A bus command (or any bus, for that matter) follows a standard of `a_ssdd`, where `ss` is the source and `dd` is the destination. Operations without destinations use `a_ss`.

Valid A Bus Instructions

	a_dfva	a_eava	a_rcva	a_rdva
a_vafx	a_dffx	a_eafx	a_rcfx	a_rdfx
a_vamx	a_dfmx	a_eamx	a_rcmx	a_rdmx
a_va	a_df	a_ea	a_rc	a_rd

8.15. The B Bus

Sources

VB	Vector Port B. Useful for reading sequential data in the lower bank of video memory. Once started, this port can read one 32-bit word per instruction without delay for 1024 words.
EB	E bus to B bus transceiver. Transfers 32-bit values to and from the E bus.
RC	Registered Accumulator C. Grabs the value in the B register of RC specified by `rc_b#n`.
RD	Registered Accumulator D. Grabs the value in the B register of RD specified by `rd_b#n`.
FS	Floating Point Status.
BS	The shifted output of the barrel shifter.
LU	Lookup Table PROM output.
LT	Lookup Table RAM output.

Explicit Destinations

VB	Vector Port B. Useful for writing sequential data in the lower bank of video memory. Once started, this port can write one 32-bit word per instruction without delay for 1024 words.
FY	Floating Point Processor Y Operand.
MY	Multiplier/Accumulator Y Operand.
LT	Lookup Table RAM.

Implicit Destinations

EB	E bus reads B bus using EB as a source.
RC	Registered Accumulator C can grab the B bus for operations using `rc_sbbus` and write the B bus to a register using `rc_fbbus`.
RD	Registered Accumulator D can grab the A bus for operations using `rd_sbbus` and write the B bus to a register using `rd_fbbus`.

Valid B Bus Instructions

```
b_ebvb   b_rdvb   b_rcvb            b_bsvb   b_luvb   b_ltvb   b_fsvb
b_ebfy   b_rdfy   b_rcfy   b_vbfy   b_bsfy   b_lufy   b_ltfy
b_ebmy   b_rdmy   b_rcmy   b_vbmy   b_bsmy   b_lumy   b_ltmy
b_eblt   b_rdlt   b_rclt   b_vblt   b_bslt
b_eb     b_rd     b_rc     b_vb     b_bs     b_lu     b_lt
```

8.16. The C Bus

Sources

EC	E bus to C bus transceiver. Transfers 32-bit values to and from the E bus.
RC	Registered Accumulator C. Values come from the Y multiplexer.
RCH	Registered Accumulator C. High 16 bits of RC's Y bus, copied into the high and low halves.
RCL	Registered Accumulator C. Low 16 bits of RC's Y bus, copied into the high and low halves.
FH	Floating Point Processor. High 32 bits of the double-precision result or all of the single-precisions result
FL	Floating Point Processor. Low 32 bits of the double precision result.
MH	Multiplier/Accumulator. High 32 bits of result
ML	Multiplier/Accumulator. Low 32 bits of result.
LR	Lookup Register, for readback.

Explicit Destinations

BR	Barrel Register, the Barrel Shifter input.
LR	Lookup Register. Lookup address for the lookup table.

Implicit Destinations

EC	E bus can read the C bus using EC as a source.
RC	RC can grab the value off the C bus using `rc_fcbus`.

Valid C Bus Instructions

```
c_ecbr   c_mhbr   c_mlbr   c_fhbr   c_flbr   c_rcbr   c_lrbr
c_eclr   c_mhlr   c_mllr   c_fhlr   c_fllr   c_rclr
c_ec     c_mh     c_ml     c_fh     c_fl     c_rc     c_lr     c_rcl    c_rch
```

8.17. The D Bus

Sources

ED	E Bus Transceiver. Transfers 32-bit values to and from the E bus.
RD	Registered Accumulator D. Values come from the RD's Y bus.
RDH	Registered Accumulator D. High 16 bits of RD's Y bus, copied into the high and low halves.
RDL	Registered Accumulator L. Low 16 bits of RD's Y bus , copied into the high and low halves.
CN	Barrel Shift Count Register. 5-bit readback source.
MO	Mode Register. 24-bit readback source.
FB	F Bus Transceiver. Reads the 16-bit value off of the F bus.
AR	Address Readback Register. Contains the address of the last memory address read or written with AC or AI. This is for use by the debugger only.
AM	DRAM Mode Register. 16-bit readback source.

Explicit Destinations

AI	Address Immediate Register. Specifies ordinary linear addresses.
MO	Mode Register. Controls miscellaneous processor functions.
AM	Addressing Mode Register. Controls memory access functions.
CN	Barrel Shift Count Register.
SF	F Bus Register. Cannot be read until the instruction after it has been loaded.

Implicit Destinations

ED	E bus can read the D bus using ED as a source.
RD	RD can read the D bus using `rd_rdbus`.
AC	AC can read the D bus using `acxld`, `acyld`, and/or `aczld`.

Valid D Bus Instructions

```
d_edai   d_arai                                          d_rdai
d_edam                                          d_fbam   d_rdam
d_edmo                                                   d_rdmo
d_edcn                                                   d_rdcn
d_edsf           d_amsf                         d_fbsf   d_rdsf
d_ed     d_ar    d_am    d_mo    d_cn    d_fb   d_rd     d_rdh    d_rdl
```

8.18. The E Bus

Sources

EA	E bus to A bus Transceiver. Transfers data to and from the A bus.
EB	E bus to B bus Transceiver. Transfers data to and from the B bus.
EC	E bus to C bus Transceiver. Transfers data to and from the C bus.
ED	E bus to D bus Transceiver. Transfers data to and from the D bus.
CD	Special. Reads 16 high bits of the C bus and the 16 low bits of the D bus.
DC	Special. Reads 16 high bits of the D bus and the 16 low bits of the C bus.
DR	Data Read Register. Contains data read in the last read instruction. DR is not valid until after the memory read instruction.

Explicit Destinations

DW Data Write Register. Can be written in 7 different ways:

DW	Write enable to all 32 bits of DW.
WL	Write enable to bits 0-15 of DW.
WH	Write enable to bits 16-31 of DW.
WR	Write enable to bits 0-7 (Red) of DW.
WG	Write enable to bits 8-15 (Green) of DW.
WB	Write enable to bits 16-23 (Blue) of DW.
WA	Write enable to bits 24-31 (Alpha) of DW.

Implicit Destinations

EA	A bus can read the E bus using EA as a source.
EB	B bus can read the E bus using EB as a source.
EC	C bus can read the E bus using EC as a source.
ED	D bus can read the E bus using ED as a source.

Valid E Bus Instructions

```
e_drdw   e_eadw   e_ebdw   e_ecdw   e_eddw   e_cddw   e_dcdw
e_drwh   e_eawh   e_ebwh   e_ecwh   e_edwh   e_cdwh   e_dcwh
e_drwl   e_eawl   e_ebwl   e_ecwl   e_edwl   e_cdwl   e_dcwl
e_drwl   e_eawr   e_ebwr   e_ecwr   e_edwr   e_cdwr   e_dcwr
e_drwg   e_eawg   e_ebwg   e_ecwg   e_edwg   e_cdwg   e_dcwg
e_drwb   e_eawb   e_ebwb   e_ecwb   e_edwb   e_cdwb   e_dcwb
e_drwa   e_eawa   e_ebwa   e_ecwa   e_edwa   e_cdwa   e_dcwa
e_dr     e_ea     e_eb     e_ec     e_ed     e_cd     e_dc
```

8.19. The F Bus

Sources

SD	Sequencer Data. Reads the 16 low bits of the instruction word data field.
SF	F Bus Register, read to the F bus.
SA	Sequencer A bus. Reads register/counter A, the stack pointer, or a value off the stack, depending on the sequencer instruction.
SB	Sequencer B bus. Reads the current value of register/counter B.

Implicit Destinations

FB	D bus can read the F bus using FB as a source.
SQ	Sequencer can read the F bus using F bus as a source.

Valid F Bus Instructions

```
f_sd     f_sf     f_sa     f_sb
```

Figure 8-10 *Instruction Summary*

RC and RD operand select

r?_a#n	A src reg n
r?_b#n	B src reg n
r?_c#n	C dst reg n
r?_rareg	R operand = A reg
r?_rabus	R operand = A bus
s?_sbreg	S operand = B reg
r?_sbbus	S operand = B bus
r?_smq	S operand = MQ reg
r?_yalu	Y mux=alu shifter output
r?_ymq	Y mux = MQ reg
r?_fabus	Write A bus to reg
r?_fbbus	Write B bus to reg
r?_fyout	Write Y mux outp to reg
r?_f?bus	Write C or D bus to reg
r?_nowe	write nothing to reg

Shifts (with alu op)

r?_sra	alu >> 1 arithmetic
r?_srad	alu >> 1, MQ >> 1
r?_srl	alu >> 1 logical
r?_srld	alu >> 1, MQ >> 1
r?_sla	alu << 1 arithmetic
r?_slad	alu << 1, MQ << 1
r?_slc	alu <<> 1
r?_slcd	alu << 1, MQ << 1
r?_src	alu <>> 1
r?_srcd	alu >> 1, MQ >> 1
r?_mqsra	MQ >> 1 arithmetic
r?_mqsrl	MQ >> 1 logical
r?_mqsll	MQ << 1 logical
r?_mqslc	MQ <<> 1
r?_loadmq	MQ = alu
r?_pass	alu out (default)
r?_sin#n	Selects fill for shift

Sequencer Operations

pofb	pop stack, SA = stack
rdsp	SA = stack ptr
rdst	SA = stack
pora	pop stack, RCA = stack
porb	pop stack, RCB = stack
pura	push RCA onto stack
pufb	push F bus onto stack
pump	push MPC onto stack
ldra	RCA = F bus
ldrb	RCB = F bus
ldsp	SP = F bus
cont	noop
juca	RCA--, cond. jump = F bus
jucb	RCB--, cond. jump = F bus
jura	cond. jump = RCA
jufb	cond. jump = F bus
just	cond. jump = stack
lpst	cond. jump = stack or pop stack and continue
lpxa	cond. pop stack, jump= RCA
lpxf	cond. pop stack,jump = F bus
lpca	RCA--; 3-way branch
lpcb	RCB--; 3-way branch
lpsb	RCB--; 3-way branch
jsra	cond. push MPC, jump = RCA
jsfb	cond. push MPC, jump = F bus
jsaf	cond. push MPC, jump = RCA or F bus
retn	cond. pop stack,jump = stack

ALU operations

r?_add	R + S
r?_addinc	R + S + 1
r?_subr	S - R
r?_subs	R - S
r?_subrdec	S - R - 1
r?_subsdec	R - S - 1
r?_ps	S
r?_pr	R
r?_ms	-S
r?_mr	-R
r?_ocs	~S
r?_ocr	~R
r?_incs	S + 1
r?_incr	R + 1
r?_xor	R ^ S
r?_and	R & S
r?_or	R | S
r?_nand	~(R & S)
r?_nor	~(R | S)
r?_andnr	~R & S

ALU byte operations

r?_set0	set reg B selected bytes & mask
r?_set1	set reg B selected bytes | mask
r?_tb0	test reg B selected bytes & mask
r?_tb1	test reg B selected bytes | mask
r?_abs	absolute value of S
r?_smtc	S signed-magnitude -> twos comp.
r?_addi	S + low 4 bits of A reg address
r?_subi	S - low 4 bits of A reg address
r?_badd	R + S, in selected bytes
r?_bsubs	R - S, in selected bytes
r?_bsubr	S - R, in selected bytes
r?_bincs	S + 1, in selected bytes
r?_bms	-S, in selected bytes
r?_bocs	~S, in selected bytes
r?_bxor	S ^ R, in selected bytes
r?_band	S & R, in selected bytes
r?_bor	S | R, in selected bytes
r?_nop	output is zero
r?_sin#n	n is byte-select mask

Interrupts

sqbp	generate interrupt for debug
sqlock	prevent interrupt at this instruction

Constant Data Field

a_df	A bus=data field
f_sd	F bus=data field (16 bits)

Condition Code Multiplexer

ccrc	status from RC
ccrd	status from RD
ccfp	status from FP
ccff	status from FP

Condition Codes (FP,RC,RD)

(n<cond.code> negates sense)

cctrue	always true
ccfalse	always false
ccvert	vertical interval
ccintr	interrupt state
ccintr	interrupt state
ccrltw	DR < DW (upper 16 bits)
ccintr	interrupt state
ccstkw	stack empty or <= 2 spaces
cccar	carry out
ccneg	negative
ccovr	overflow
cczero	zero
cccnz	ncccar OR cczero
cccoz	cccar OR cczero
ccltz	ccneg XOR ccovr
cclezccneg XOR ccovr OR cczero	

FP condition codes (from fp_cmp)

ccagtb	A > B
ccaleb	A <= B
ccaneb	A <> B
ccaeqb	A == B
ccaltb	A < B
ccageb	A >= B

Figure 8-11 *Instruction Summary (continued)*

Barrel Shifter (BS)

c_??br	load BR
b_bs??	read BS
d_??cn	load CN
d_cn	read CN
bs_sll,n	<< n logical
bs_srl,n	>> -n logical
bs_slc,n	<< n circular
bs_sra,n	>> -n arithmetic
bs_sllcn	<< CN logical
bs_srlcn	>> -CN logical
bs_slccn	<< CN circular
bs_sracn	>> -CN arithmetic
bs_srl,0	output is 0
bs_sra,0	output is -1 if BR<0, output is 0 if BR>=0

Multiplier/Accumulator (MA)

a_??mx	load MX
b_??my	load MY
c_ml??	read ML
c_mh??	read MH
ma_pas	MH/ML = MX
ma_neg	MH/ML = - MX
ma_mpas	MH/ML = MX * MY
ma_mneg	MH/ML = -(MX * MY)
ma_madd	MH/ML = MH/ML + (MX * MY)
ma_msub	MH/ML = MH/ML - (MX * MY)
ma_msua	MH/ML = (MX * MY) - MH/ML
ma_xsign	MX is signed
ma_ysign	MY is signed
nomult	no multiply

Random Memory Access

mreadai	read value at AI addr
mreadac	read value at AC addr
mwriteai	write to AI address
mwriteac	write to AC address
macp	generate effective address from AC
acxld	load AC X field
acyld	load AC Y field
aczld	load AC Z field
acxi	increment AC X field
acyi	increment AC Y field
aczi	increment AC Z field
acxd	decrement AC X field
acyd	decrement AC Y field
aczd	decrement AC Z field

Vector Port Memory Access

v?st	begin read/write
v?rd	read shift reg to va or vb
v?wr	write va or vb to shift reg

Floating Point Processor (FP) (sgl precision)

a_??fx <fp_op>	load FX
b_??fy <fp_op>	load FY
c_fh??	read FH
c_fl??	read FL
fp_absa	FH = SUM = |AA|
fp_basa	FH = SUM = AA
fp_nega	FH = SUM = -AA
fp_flta	FH = SUM = (float)AA
fp_inta	FH = SUM = (int)AA
fp_dbla	FH/FL = SUM = (double)AA
fp_sgla	FH = SUM = (float)AA
fp_wrap	FH = SUM = wrap(AA)
fp_unwe	FH = SUM = unwrap(AA)
fp_unwi	FH = SUM = unwrap(AA)
fp_unwr	FH = SUM = unwrap(AA)
fp_max	MA = X input
fp_mas	MA = SUM register
fp_mby	MB = Y input
fp_mbp	MB = PRODUCT register
fp_aax	AA = X input
fp_aap	AA = PRODUCT register
fp_aby	AB = Y input
fp_abs	AB = SUM register
fp_pasnega	PRODUCT = MA, SUM = -AA
fp_passub2	PRODUCT = MA, SUM = 2-AA
fp_outp	FH = PRODUCT
fp_outs	FH = SUM
p_add	FH = SUM = AA + AB
fp_suba	FH = SUM = AB - AA
fp_subb	FH = SUM = AA - AB

Lookup Table (LU)

c_??lr	load LR	b_??lt	load LT
b_lu??	read LU	b_lt??	read LT

fp_cmp	sets compare cond. codes
fp_aabs	AA = |AA|
fp_babs	AA = |AB|
fp_mul	FH = PRODUCT = MA * MB
fp_mulwra	FH = PRODUCT = MA * MB (wrap A)
fp_mulwrb	FH = PRODUCT = MA * MB (wrap B)
fp_aabs	MA = |AA|
fp_babs	MA = |AB|
fp_muladd	PRODUCT = MA * MB, SUM = AA + AB
fp_mulsuba	PRODUCT = MA * MB, SUM = AB - AA
fp_mulsubb	PRODUCT = MA * MB, SUM = AA - AB
fp_mulnega	PRODUCT = MA * MB, SUM = -AA
fp_mulsub2	PRODUCT = MA * MB, SUM = 2 - AA
fp_mulpasa	PRODUCT = MA * MB, SUM = AA
fp_pasadd	PRODUCT = MA, SUM = AA + AB
fp_passuba	PRODUCT = MA, SUM = AB - AA
fp_passubb	PRODUCT = MA, SUM = AA - AB
fp_paspasa	PRODUCT = MA, SUM = AA

Default word

bs_sra,15 vbnop vanop mnop macn acznop acynop acxnop \
rc_rareg rc_sbreg rc_nop rd_rareg rd_sbreg rd_nop rd_car0\
rd_nowe rd_yalu rc_car0 rc_nowe rc_yalu ccrc f_sd e_ea\
d_fb c_lr b_fs a_va cont sqnolock sqnobp sqnowt ma_xuns\
ma_yuns nomult rc_sin#15 rd_sin#15

Figure 8-12 *Data Flow Summary*

The A Bus
Sources:

VA	Vector port A
DF	Data field
EA	A bus data
RC	RC A reg (rc_a#n)
RD	RD A reg (rd_a#n)

Explicit Destinations:

VA	Vector port A
FX	FP X operand
MX	MA X operand

Implicit Destinations:

EA	E bus uses e_ea??
RC	RC uses rc_rabus or rc_fabus
RD	RD uses rd_rabus or rd_fabus

The D bus
Sources:

ED	E bus data
RD	RD Y bus
RDH	RD Y bus high 16 bits
RDL	RD Y bus low 16 bits
CN	BS Count reg
MO	Mode reg
FB	F bus data
AR	Address Readback reg
AM	AM reg

Explicit Destinations:

AI	AI reg
MO	Mode reg
AM	AM reg
CN	BS Count reg
SF	F bus reg

Implicit Destinations:

ED	E bus uses e_ed??
RD	RD uses rd_fdbus
AC	AC uses acxld, acyld, aczld

The B Bus
Sources:

VB	Vector port B
EB	E bus data
RC	RC B reg (rc_b#n)
RD	RD B reg (rd_b#n)
FS	FP status
BS	BS output
LU	LU output
LT	LT output

Explicit Destinations:

VB	Vector port B
FY	FP Y operand
MY	MA Y operand
LT	LT input

Implicit Destinations:

EB	E bus uses e_eb??
RC	RC uses rc_sbbus or rc_fbbus
RD	RD uses rd_sbbus or rd_fbbus

The E Bus
Sources:

EA	A bus data
EB	B bus data
EC	C bus data
ED	D bus data
CD	C bus high 16 bits and D bus low 16 bits
DC	D bus high 16 bits and C bus low 16 bits
DR	Data Read (DR) register

Explicit Destinations:

DW	Data Write (DW) register
WL	DW bits 0-15
WH	DW bits 16-31
WR	DW bits 0-7
WG	DW bits 8-15
WB	DW bits 16-23
WA	DW bits 24-31

Implicit Destinations:

EA	A bus uses a_ea??
EB	B bus uses b_eb??
EC	C bus uses c_ec??
ED	D bus uses d_ed??

The C bus
Sources:

EC	E bus uses e_ec??
RC	rc_c#n
RCH	rc_c#n (Y bus high)
RCL	rc_c#n (Y bus low)
FH	FP high result
FL	FP low result
MH	MA high result
ML	MA low result
LR	LR readback

Explicit Destinations:

BR	BS input
LR	LR address

Implicit Destinations:

EC	E bus uses e_ec??
RC	RC uses rc_fcbus

The F Bus
Sources:

SD	Sequencer Data
SF	F bus reg
SA	Sequencer A bus (RCA, stack ptr, or stack)
SB	Sequencer B bus (RCB)

Implicit Destinations:

FB	D bus uses d_fb
SQ	Sequencer

Data Flow
Bus command format:
<bus>_ss or
<bus>_ssdd

where

<bus>	a,b,c,d,e, or f
ss	source
dd	destination

9

Utilities

9

Utilities

This chapter describes these TAAC-1 utility programs:

- `ras2taac` Writes a Sun rasterfile to a rectangular region of the TAAC-1 frame buffer and optionally displays it.
- `taabs2o` Converts TAAC-1 absolute file to Sun object file.
- `tachan` Enables or disables TAAC-1 video channels.
- `taclear` Clears TAAC-1 image memory to specified colors.
- `tadeb` TAAC-1 process debugger.
- `tainit` Initializes the TAAC-1.
- `taload` Loads IFF image file into TAAC-1 data/image memory.
- `tamakedef` Generates host include file from TAAC-1 link map
- `tamon` Accesses TAAC-1 processor and memory.
- `taprof` Profiles TAAC-1 program execution time.
- `tashow` Displays TAAC-1 image memory on single-monitor configurations.
- `tarun` Opens the TAAC-1, loads and executes a TAAC-1 program.
- `taread` Reads TAAC-1 data/image memory into image file in IFF format.
- `tatool` Provides a viewport into TAAC-1 image memory for single monitor configurations. It also sets and modifies video keying parameters.
- `tatxt2o` Converts text file to Sun object file.

The chapter also describes the following Image File Format (IFF) utilities:

- `iffcreate` prepends header to an image file
- `iffreorder` reorganizes the pixels of an IFF file
- `iffgetkey` returns values of keywords in an IFF file
- `iffsetkey` changes/adds keyword/value pairs to an IFF file
- `iffaddkey` adds keyword/value pairs to an IFF file
- `iffstrip` removes the header from an IFF file
- `iffcutout` crops an IFF file
- `ras2iff` converts a binary Sun rasterfile to an IFF file
- `iffcolor` converts an 8-bit IFF file to a 24-bit color IFF file

These IFF utilities are described in the `taload` section of the chapter.

9.1. `ras2taac`, Write Sun Rasterfile to TAAC-1 Memory

`ras2taac` writes a Sun rasterfile to a rectangular region of TAAC-1 data/image memory and optionally displays it in a window. Usage:

```
% ras2taac [-n] [-x xaddr] [-y yaddr] rasterfile
```

Where:

- `-n` specifies that no TAAC-1 window is to be created on the host for display.
- `-x` specifies the image's starting x address in TAAC-1 data/image memory; the default value is zero.
- `-y` specifies the image's starting y address in TAAC-1 data/image memory; the default value is zero.
- `rasterfile` specifies the input Sun rasterfile. This file format is described in the Pixrect Reference Manual.

9.2. taabs2o, Convert .abs File to Sun Object File

The utility `taabs2o` converts a TAAC-1 executable in absolute file (`.abs`) format to a host object file so that it can be linked into a host program. The object file (`.o`) that is created should be added to the object file list in the link command. To download and execute the TAAC-1 program, call ta_runm() or ta_runb() from the host program, as described in the host library chapter of the *TAAC-1 Software Reference Manual.*

Command Syntax

To invoke `taabs2o`, type:

```
% taabs2o filename.abs   filename.o symbol
```

Where:

- `filename.abs` specifies the input TAAC-1 absolute file.
- `filename.o` specifies the output Sun object file.
- `symbol` is an optional argument used when you are linking multiple .abs files into a host program.

`taabs2o` produces an object file that contains the TAAC-1 code as if it were an array of bytes. If you specify a symbol, this array is named "`<symbol>data`." You can then use this symbol as an argument to ta_runb(), a host routine which allows you to switch between multiple TAAC-1 programs. For a detailed example of switching between multiple TAAC-1 programs, see Appendix C of the *TAAC-1 Software Reference Manual*.

If you plan to link only one TAAC-1 executable into a host program, you can omit the third argument in your call to `taabs2o`. In this case, the array is named "`_ta_absdata.`" You can use the host routine ta_runm() to download and execute a single TAAC-1 program.

Example

To link a single TAAC-1 program with a host program, follow these steps:

Given `sunfile.c` and `taacfile.tc`:

```
% tacc taacfile.tc
% talink taacfile.obj
```

```
% taabs2o taacfile.abs taacfile.o
% cc -c sunfile.c
% cc -o sunfile sunfile.o taacfile.o
```

To load the linked-in TAAC-1 program, `sunfile.c` would call host routines `ta_ldm(tahandle)` or `ta_runm(tahandle)`.

9.3. tachan, TAAC-1 Channel Selection Tool

The utility `tachan` provides the means to selectively enable and disable TAAC-1 video channels.

Command Syntax

To invoke `tachan`, enter:

```
% tachan [-01234rgbafnv]
```

Where:

- `-0, -1, -2, -3` designate enabled channels. Entering `tachan -0 -3` enables channels 0 and 3.
- `-r, -g, -b, -a` provides exactly the same function as the above options with alternate names (red, green, blue, alpha).
- `-4` or `-f` enables all channels .
- `-n` disables all channels (no image).
- `-v` prints out the `tachan` version number and date.

If no options are specified, `tachan` enables all channels.

Functions Calls

`ta_open()`	Opens communication with the TAAC-1.
`ta_set_channel_select()`	Enables the designated channels.
`ta_close()`	Closes communication with the TAAC-1.

Channel 0 is also referred to as the red channel and corresponds to least significant image bits 0-7. Similarly, channels 1 (green), 2 (blue) and 3 (alpha) correspond to image bits 7-15, 16-23, and 24-31, respectively.

9.4. `taclear`, TAAC-1 Clear Tool

The utility `taclear` clears one or both banks of TAAC-1 image/data memory to a specified value.

Command Syntax

To invoke `taclear`, enter:

```
% taclear [-abtv] [-x hexnum] [-c colorname]
```

Where:

- `-a`, `-b`, and `-t` specify bank A, bank B, or both banks to be cleared. The default is bank A.
- `-x hexnum` specifies a value to write to the selected bank(s). `hexnum` should be a valid hexadecimal, 24-bit integer (without the 0x prefix), such as `-x ff00ff`.
- `-c colorname` specifies a color to write to the selected bank(s), where `colorname` may be one of the following recognized colors: black, red, green, yellow, blue, magenta, cyan, or white. For example, `taclear -c red` would clear to red (assuming the colormap is loaded for full color).
- `-v` prints out the `taclear` version number and date.

`taclear` with no arguments clears bank A to a value of zero (black).

When used with `tashow`, `taclear` verifies simple video functionality of the TAAC-1.

9.5. tadeb, TAAC-1 Debugger

`tadeb` is a window-based utility used for debugging both C and assembly language programs that run on the TAAC-1. It allows you to load and run a TAAC-1 program, to look at memory, to set breakpoints or single step, and to read and write memory, including register variables. Beginning with the TAAC-1 2.3 software release, you must use the `-g` option when you are linking TAAC-1 programs in order to run the debugger. Otherwise the register dump routine is not included in the link.

Because only one host process can effectively communicate with the TAAC-1 at a time, `tadeb` cannot be used to debug a host and TAAC-1 process concurrently. In this case, the host process can be used to initiate the TAAC-1 process, but must be stopped before the debugger is invoked.

Command Syntax

To invoke `tadeb`, type:

```
% tadeb [-i] [-v]
```

Where:

- `-i` starts the debugger without initializing the TAAC-1. This is useful when the code you wish to debug has already been loaded and started by another host program.
- `-v` prints out the `tadeb` version number and date.

User Interface

Upon initiation, tadeb creates two overlapping windows as shown in the sample screen in Figure 9-1. The main debugger window is the larger of the two and is used for command control and display of data. The other smaller window on the right is the source code display window and is used to examine code and track code listings while single-stepping through a program.

Figure 9-2 summarizes the usage of the individual panels within each of two major windows. The main window is broken down into a main command panel that spans the full width at the top and eight larger panels below, arranged in two columns of four.

These are the primary controls of the main command panel:

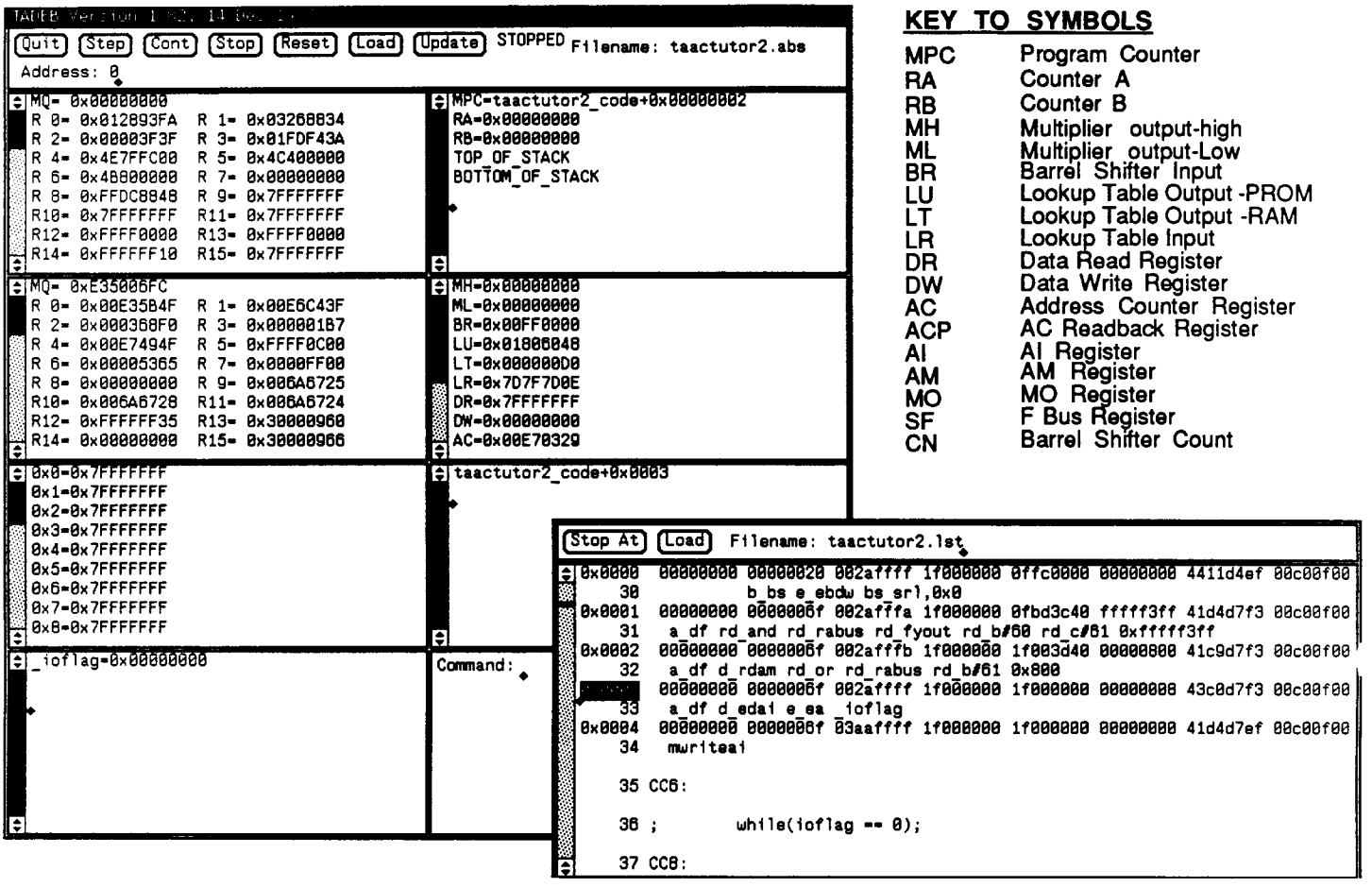
Quit	Stops the debugger task.
Step	Single-steps the TAAC-1 processor.
Cont	Starts the TAAC-1 processor from the current program counter.
Stop	Stops the TAAC-1 processor.
Reset	Resets the program counter and stack pointer without changing any other variable values.
Load	Loads the executable and updates the display panels if a corresponding map file exists.
Update	Updates the display and TAAC-1 memories based on editing of applicable windows.
Address	Expression or address of starting memory values in panel below RD register panel.
Filename	Name of TAAC-1 executable file, `<name>.abs`.
Processor State	TAAC-1 Processor State - Normally reads either Stopped or Running. ?? indicates that there is no register dump code reference in the linker map file, `<name>.map`.

The TAAC-1's integer ALUs, RC and RD, each contain 64 general usage registers which are displayed in the two upper left display panels. Below these panels on the left is the memory display panel pointed to by the address expression defined in the top (main) command panel.

The lower left panel displays memory contents by symbol, variable name, expression or explicit address, as defined in the command entry panel on the lower right. The later panel is used for entering command lines to the debugger. Recognized commands include: `help`, `quit`, `step`, `cont`, `stop`, `load`, `update`, `debug <filename>`, `display <expression>` and `undisplay <expression>`. While most commands mimic the button controls in the top command panel, `display` and `undisplay` are unique and are used to add and remove memory

locations whose contents are displayed in the lower left panel. Expressions can consist of global symbols referenced in the

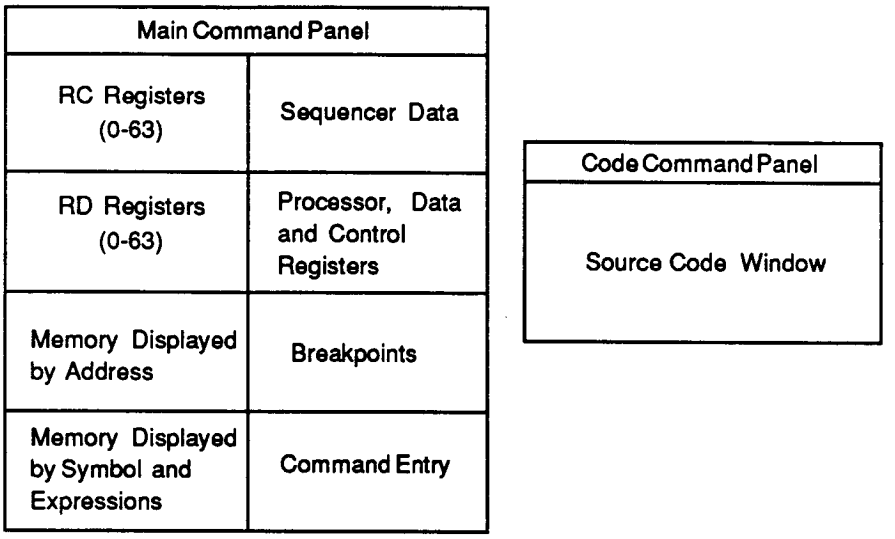
Figure 9-1 *Sample* `tadeb` *Window*



KEY TO SYMBOLS

MPC	Program Counter
RA	Counter A
RB	Counter B
MH	Multiplier output-high
ML	Multiplier output-Low
BR	Barrel Shifter Input
LU	Lookup Table Output -PROM
LT	Lookup Table Output -RAM
LR	Lookup Table Input
DR	Data Read Register
DW	Data Write Register
AC	Address Counter Register
ACP	AC Readback Register
AI	AI Register
AM	AM Register
MO	MO Register
SF	F Bus Register
CN	Barrel Shifter Count

Figure 9-2 `tadeb` *Window Layout*



executable's map file and/or numeric expressions and operators (the debugger must be able to resolve any expression into a memory address). The help command pulls up a window which summarizes the available commands. To remove this window, select the Done button.

The top right panel of the main window displays sequencer data such as the current program counter and the internal sequencer counters RA and RB.

`tadeb` displays stack information between `TOP_OF_STACK` and `BOTTOM_OF_STACK`. Currently, all stack values are displayed as program addresses, expressed as offsets from the start of a code segment.

Major processor data and control registers are displayed in the panel below the sequencer information, while breakpoints can be defined in the remaining panel.

The separate and distinct source code window can be used to look at program listing files and trace code execution as the processor is single-stepped. This window also provides a convenient method of entering breakpoints by highlighting the instruction and using the STOP AT control to include a breakpoint.

Usage Notes

The following sections describe the steps involved in using the debugger.

Loading a Program

1. Run `tadeb` by entering:

   ```
   % tadeb
   ```

2. Move the cursor to the `Filename` prompt. Enter the program name (`<name>.abs`) or use the right mouse button to pull down a menu of all the executable files in the current directory. Once the name is entered by either means, press the Load button.

3. `tadeb` loads the file, but the PC will be held at 0 and the screen will not be updated until you select Step, Cont (continue), or Reset.

4. `tadeb` uses the link map generated by `talink` for its addresses, so the `.map` file must be current and in the same directory as the program. If the map file cannot be found the top four panels will not be initialized after the load and the processor status display may show question marks.

5. If the .abs file you selected was not linked with the `-g` option, `tadeb` will display an error message. You must relink the program before proceeding.

Debugging a Program Without Loading It

If you have a program that is downloaded from the host and uses input from the host, it is possible to debug it without loading it inside `tadeb`.

1. Run and exit the host program (the host program and `tadeb` cannot operate concurrently).

2. Run `tadeb -i`. Enter the filename but select Update instead of Load. This allows `tadeb` to read the link map but will not load the program. Select Stop to stop the processor. You can then look at memory or single-step the process.

3. To rerun the program without loading it, select Reset, which halts the processor, resets the program counter to zero and resets the sequencer stack. Reset does not change any program variables.

4. You may want to change either the host code, the TAAC-1 code, or both, so that the TAAC-1 program waits for a flag to be set before proceeding. Then you can set the flag by writing to memory inside `tadeb`.

Running a Program Without Setting Breakpoints

1. Run `tadeb` and load the TAAC-1 program.
2. Select Cont (continue). The processor status message in the top control panel will change from Stopped to Running.
3. To stop the program, select Stop. A TAAC-1 C program, when it is finished, hangs in a one-instruction loop.

Setting a Breakpoint and Running a Program

1. Enter the breakpoint address in the breakpoint window. The address can be an absolute address, a global symbol defined in the link map, or an expression using symbols and constants (note that the C compiler prefaces subroutine names with an underscore). You can enter multiple breakpoints, separated by one or more spaces or lines. The breakpoint window can be edited, scrolled, and saved to a file. An easier way to set breakpoints is to highlight any portion of a line of code in the source code window and press the Stop At button. The breakpoint address will appear in the breakpoint window. Examples of valid breakpoints:

```
0x47
_main
_main+0x1f
```

2. Select Cont. When the breakpoint is reached, `tadeb` highlights the correct breakpoint, unless the breakpoint falls on an instruction containing a branch. The processor status will now indicate Stopped in the top command panel. The instruction at the breakpoint will have been executed, and the program counter (MPC) will point to the next instruction to be executed after the breakpoint. If the breakpoint is within a code segment that is defined with an assembler listing file in the operating directory, the source code window will display the listing file and highlight the breakpoint instruction.
3. Breakpoints are not allowed on instructions containing floating point operations, because the floating point processor must be clocked during the cycle immediately preceding its operation. Stopping for a breakpoint would interfere with this sequence. The debugger generate an error for these breakpoints.

4. You can specify a count with a breakpoint, and `tadeb` will stop only when it has reached the breakpoint "count" times. For example, to stop at the fifth time the program calls subroutine `subx`, enter:

```
_subx#5
```

Single-Stepping

1. Select Step. This causes a breakpoint at each instruction. Instructions that disallow interrupts, including floating point instructions, are stepped over.

Looking at Memory

1. Enter the address at the Address: prompt in the command panel at the top of the main window. You can enter:
   - The name of a global symbol
   - An absolute address
   - An expression using global symbols and constants

   Entering an invalid address causes a message window to appear. If you press the right mouse button near the Address entry line, a pull-down menu of recent addresses will appear, simplifying the process of changing and rechecking old addresses.
2. The first 30 words of memory starting at the given address will be displayed in the lower left window. The words are scrollable and their values can be modified with standard window editor techniques. Selecting Cont, Step, or Update rewrites any variables that might have been changed.
3. Pressing the right mouse button inside the memory window allows you to select a different radix for the display.

Function Arguments and Automatic Variables

1. At the entrance to a function, the compiler subtracts the framesize (the size needed to hold all the automatic variables, including locations needed to save any registers) from RD62, and then jumps to "_S.<subroutine-name>". To look at an automatic variable within a subroutine, use the address

```
RD62 + stkloc offset [+ pushsize]
```

2. To look at an argument, use the address

```
RD62 + framesize + argloc offset [+ pushsize]
```

`pushsize` applies only if the compiler has begun to push arguments onto the stack, in preparation for calling another subroutine. For each argument the compiler pushes onto the stack, it subtracts the size of that argument from the stack pointer.

To understand how to read local variables, it helps to look at the assembly code for a function that is reading or writing a variable. The following example has comments inserted:

```
/* the C code */
sub(a,b,c)
int a,b,c;
{
      int w,x,y,z;
      y = b;
}

/* the assembly language source */
;sub(a,b,c)
;int a,b,c;
      .global_sub,_S.sub
_S.sub: ; start of sub
;{
; a at argloc 0 (0x0) size 1        argloc contains the offsets for function
; b at argloc 1 (0x1) size 1        arguments
; c at argloc 2 (0x2) size 1
;     int w,x,y,z;
; w at stkloc 0 (0x0) size 1        stkloc contains the offsets for automatic
; x at stkloc 1 (0x1) size 1        variables
; y at stkloc 2 (0x2) size 1
; z at stkloc 3 (0x3) size 1
;     y = b;

;first reading b, located at SP(in RD62) + framesize(=4) + arglog offset(=1)
       a_df d_rdai rd_add rd_rabus rd_b#62 CC6+0x1
       mreadai\
;
;now writing y, located at SP(RD62) + stkloc offset(=2)
            d_rdai rd_addi rd_a#2 rd_b#62
       e_drdw
       mwriteai
;
;this is the subroutine exit - it restores any registers it needs to
;and then adds the framesize back into the stack pointer
CC7:
```

```
        d_rdai rd_addi rd_fyout rd_a#4 rd_b#62 rd_c#62\
              retn cctrue
;
;this is the subroutine entrance - it subtracts the framesize from the
;stack pointer, saves any registers it needs to, and jumps to _S.sub
_sub:
        d_rdai rd_subi rd_fyout rd_a#4 rd_b#62 rd_c#62\
              f_sd jufb cctrue _S.sub

CC6=4 ;this is the framesize = size of all the automatic variables
;}
```

Using the Display Window

1. In the command line of the lower right window, enter a global variable name or an expression which is translatable into a valid memory address. If the expression is illegal, the command line will remain static awaiting correction. `tadeb` can handle indirection operators for pointer variables. Examples of valid command line syntax:

   ```
   display _ioflag
   display _array+0x10
   display 0x30000007
   display *_p
   ```

2. To select an alternate radix for a variable, select the variable with the left mouse button and then press the right mouse button for the radix pull-down menu. In this window, the radix can be individually selected for a variable, as opposed to the other windows, where the radix applies to all variables in the window.

3. Memory values displayed cannot be edited or updated from this window.

4. To remove entries, use the undisplay <expression> command.

Exiting `tadeb`

Select Quit.

General Information

- You can change the radix of the display in any window. Select the

right mouse button for the radix pull-down menu. The radix is changed on a per-window basis except in the lower left memory display window, which allows radix customizing of each entry.

- Double precision can be selected in either the memory window or the display window. In the display window, you can specify that a particular variable is double precision and it will be displayed in that mode. In the memory window, selecting the double precision radix means that, starting with the first address shown in that window, every two words will be considered to contain a double precision value, and will be displayed that way. Double precision values are displayed with a leading "D" to distinguish them from single-precision values. Note that the double precision radix cannot be selected for any other windows.
- All the windows are scrollable except the lower-right command line panel.
- All 64 registers are displayed for each ALU. RD62 contains the C stack pointer. These registers can be modified using standard `textedit` techniques. Selecting Step, Cont, or Update updates the register.
- The MPC (program counter) shows the address of the *next* instruction to be executed. The address shown is relative to the start of its code segment.
- All stack values between `TOP_OF_STACK` and `BOTTOM_OF_STACK` are currently expressed as offsets from the start of a code segment. However, the stack can contain other kinds of values, such as loop counts, and so this display can be misleading.

9.6. `tainit`, TAAC-1 Initialization Tool

The utility `tainit` provides a simple and quick means of initializing the TAAC-1.

To invoke `tainit`, enter:

```
% tainit [-vstm]
```

Where:

- `-v` prints out the tainit version number and date. It performs no other functions.
- `-s`, `-t`, `-m` set Sun, TAAC-1, or mixed video, respectively. Note that once you are in TAAC-1 video, you are "blind" with respect to Sun video, so do not move the mouse or you will have difficulty returning to Sun or mixed video with `tainit` or `tashow`.
- `tainit` without options initializes the TAAC-1 and sets the video mode according to the configuration file.

Function Calls

`ta_open()`	Opens communication with the TAAC-1 and verifies that the TAAC-1 is accessible.
`ta_init()`	Initializes the state of the TAAC-1, including default video and colormaps. Sets video mode according to configuration file. In single-monitor mode, the default video is Sun video and sync. In dual-monitor mode, the default video and sync formats are as defined in `$TAAC1/hardware/taconfig.<hostname>`. See the `ta_init()` host library entry for more details.
`ta_set_video()`	Sets the video and sync modes.
`ta_close()`	Closes communication with the TAAC-1.

9.7. `taload`, Image File Loader

`taload` writes an image file (in IFF format) to TAAC-1 data/image memory. To invoke `taload`:

```
% taload [-f imagefile] [-x xoffset] [-y yoffset]
```

Where:

- `imagefile` is the name of an image file stored in IFF format (described below). If you do not specify a filename, taload uses standard input.
- `xoffset` and `yoffset` specify a starting address, in TAAC-1 data/image memory, to which the image will be written; the default is (0, 0).

IFF Image File Format

The IFF Image File Format is a flexible and extensible format for associating useful information with a data set. The approach, utilities, and libraries are not limited to the storage of images, although several conventions have been established for two-dimensional multi-band data.

Format

The basic format of an IFF file is an ASCII header terminated by formfeed and newline characters, followed by the image data. The header begins with a "magic number" (the string "ncaa") identifying the file as an IFF file. This magic number may be placed into /etc/magic for use with the utility file(1). A ^L (Control/L) terminates the header. This character allows you to examine the header using more(1).

The remainder of the header consists of keyword and value pairs. The syntax of a pair is a keyword, an equals sign (=), the value string, and a terminating semicolon (;). Some examples:

```
bands=4;
size=512 512;
title=Famous Mandrill Picture;
```

Keywords should not contain embedded blanks, equals signs (=), backslashes (\), or semicolons (;). Semicolons in value strings must be represented by a backslash followed by a semicolon (\;). A backslash character is represented by two consecutive backslashes (\\).

Image Keywords

A set of five keywords has been chosen to represent the minimum amount of data that must accompany an image. These keywords are:

`rank` dimensionality of the data (2 for images, 3 for volumes, etc.)

`size` number of samples along each dimension. For 2D images, size refers to X and Y size, in pixels.

`bands` number of bands in the image (1 for monochrome, 3 for RGB, etc.)

`bits` number of bits for each band

`format` format of the image data. The IFF format specifies how the image data is stored in the format keyword. Currently-defined formats are:

"base" - a conventional format used in some of the image handling utilities. The base format stores pixels in raster order. All bands of a pixel are stored together--band interleaved as opposed to band sequential. Each pixel band is stored in an integral number of bytes; no bit packing is done. A typical TAAC-1 (32-bit) image file is stored in base format as AGBRABGR...

"Block Pseudo" - the format used for a2bitmovie frames. This format has two additional keywords: `colormapsize` and `colormap`. `colormapsize` is normally 256. `colormap` signifies the presence of an ASCII colormap in the image data file. For more information about a2bitmovie, refer to the Demo chapter of the *TAAC-1 Software Reference Manual.*

Keyword values that contain more than one piece of information, such as for size and bits, are represented by separating the parameters with white space. The header for a 512x512 24-bit color image might look like this:

```
ncaa
rank=2;
size=512 512;
bands=3;
bits=8 8 8;
format=base;
^L
```

Additional information may be included in the header simply by adding more keyword and value pairs. Binary data should be represented in hexadecimal.

IFF Utilities

There are a few utilities that deal with any type of IFF file, not specific to images. These allow the creation of IFF files, modification of keywords, and the removal of the IFF header.

In general, these utilities read IFF files from standard input and write IFF files to standard output, allowing several operations to be piped. The use of the -f option allows input from a file.

IFF files compressed with compress(1) can be used as input provided they end in the conventional .Z extension.

There is no special utility to list the header of an IFF file, since the formfeed header termination allows examination with more(1).

% iffcreate [-f rawfile] [-h headerfile] [key=val]

`iffcreate` prepends an image file header to standard input or the file rawfile, writing the result to standard output. Keyword/value pairs may be read in from an existing image file as well as added on the command line. The following command creates an IFF file from a 32-bit 512x512 data file:

```
% iffcreate -f myfile "rank=2" "size=512 512" "bands=4"
   "bits=8 8 8 8" "format=base" > myfile.iff
```

Alternatively, you can create a file using a header from another IFF file with the same characteristics:

```
% iffcreate -f myfile -h $TAAC1/demo/data/images/button.iff
   > myfile.iff
```

% iffreorder [-f imagefile] [-b bandnumber]

`iffreorder` reorganizes the pixels of an image file taken from standard input or the file imagefile. It may be used to extract a band or bands and reorder the components. Bands are numbered from 0 to n-1, in the order that they are stored in the file. For TAAC-1 images, the ordering is typically ABGR, so A is band 0, B is band 1, G is band 2, and R is

band 3. Any number of pairs of -b options and band numbers may be specified. The following command takes a 32-bit input IFF file with colors ordered ABGR and produces a 24-bit output file with colors ordered RGB:

```
% iffreorder -f myfile_abgr.iff -b 3 -b 2 -b 1
   > myfile_rgb.iff
```

The following command extracts only the green component of a 32-bit input IFF file with colors ordered ABGR:

```
% iffreorder -f myfile_abgr.iff -b 2 >myfile_green.iff
```

% iffgetkey [-f imagefile] [keyword]

`iffgetkey` returns values of keywords specified on the command line from an image file taken from standard input or the file imagefile. Keyword values are separated by newlines. The following command returns the contents of the bands and bits keywords:

```
% iffgetkey -f myfile.iff bands bits
```

% iffsetkey [-f imagefile] [key=val]

`iffsetkey` changes/adds keyword/value pairs to the image file taken from standard input or the file imagefile, writing the result to standard output. If the key value contains any white space, the entire keyword/value pair must be enclosed in double quotation marks. The following command changes the title keyword of an image header:

```
% iffsetkey -f myfile.iff "title=Ray-traced Sun Logo" >
   titled_image.iff
```

% iffaddkey [-f imagefile] [key=val]

`iffaddkey` adds keyword/value pairs to the image file taken from standard input or the file imagefile, writing the result to standard output. If the key value contains any white space, the entire keyword/value pair must be enclosed in double quotation marks.

```
% iffstrip [-f imagefile]
```

`iffstrip` removes the header from an image file taken from standard input or the file imagefile, writing the result to standard output. The following command removes the IFF header from a file:

```
% iffstrip -f myfile.iff > orig_file
```

Other TAAC-1 IFF Utilities

Besides `taload` and `taread`, there are additional utilities that deal specifically with TAAC-1 images. The most common types of TAAC-1 images are eight-bit monochrome and 32-bit color. Examples of TAAC-1 image headers:

Monochrome

```
ncaa
rank=2;
size=512 512;
bands=1;
bits=8;
format=base;
^L
xxxx...
```

32-bit Color

```
ncaa
rank=2;
size=512 512;
bands=4;
bits=8 8 8 8;
format=base;
^L
ABGRABGR...
```

```
% iffcutout [-f imagefile] [-x xoffset] [-y yoffset]
    [-X xsize] [-Y ysize]
```

This utility crops an image taken from standard input or the file imagefile, writing the result in the IFF format to standard output.

```
% ras2iff [-O onval] [-o offval]
```

This utility reads a binary Sun rasterfile from standard input and writes a color image in the IFF format to standard output. It handles 1-, 8-, and 24-bit rasterfiles. 1-bit rasterfiles are treated specially by ras2iff, by converting them to 32-bit color IFF files for compositing purposes.

The colors to represent binary on and off may be specified using the switches. The defaults are offval=0x00000000 (black) and onval=0xff00ffff (yellow with alpha).

```
% iffcolor
```

This utility reads a single-band, 8-bit IFF file from standard input and writes a 24-bit color IFF file to standard output, passing the data through the header colormap, if present.

See Also

`taread`, read rectangular region of TAAC-1 data/image memory.
The Hostlib-Image File Functions chapter of the *TAAC-1 Software Reference Manual.*

9.8. `tamakedef`, Include File Generator

`tamakedef` generates an include file for a host program containing the addresses of all global symbols in a TAAC-1 program. It reads the `.map` file generated by the TAAC-1 linker `talink`, extracts the names and addresses of functions and global variables, and produces an include file containing each global symbol name, prefixed by `TC_`, and its address in TAAC-1 memory. For example, if the `.map` file contained a global symbol:

```
_ioflag 0x30000000
```

the include file generated by `tamakedef` would contain the line:

```
#define TC_ioflag 0x30000000
```

This include file allows host programs to read or write TAAC-1 global variables using the same symbolic names as in the TAAC-1 programs, without worrying about where these variables are located in TAAC-1 memory. If you use `tamakedef`, structure your Makefile dependencies so that the host program depends on the include file, which depends in turn on the `.map` file from the linker.

Command Syntax

To invoke `tamakedef`, enter:

```
% tamakedef map_file include_file [-cd]
```

Where:

- `map_file` is the name of the `.map` file produced by the TAAC-1 linker `talink`.
- `include_file` is the name of the host include file to be generated by `tamakedef`.
- `-c` writes to the output file only if it will change. This option is useful for Makefile dependencies; if you re-link your TAAC-1 program but none of the global addresses has changed, this option keeps your host program from being re-compiled unnecessarily.
- `-d` extracts only data addresses from the link map. Function addresses are ignored.

9.9. tamon, TAAC-1 Monitor

`tamon` is a low-level tool that allows direct access to TAAC-1 memory. You can also use `tamon` to initialize TAAC-1 registers, to load and run programs, and to perform certain diagnostic functions. To invoke the monitor, enter:

```
% tamon
```

Commands

Type `H` or `?` to display the `tamon` help menu. Specify commands in upper or lower case.

Reading

R	Reads from the current address.
RF	Reads a floating point value.
RN	Reads from the next address.
RP	Reads from the previous address.

Writing

W	Writes to the current address.
WN	Writes to the next address.
WP	Writes to the previous address.
WR	Writes, then immediately reads.
W2V	Writes two values to the current address.
W2A	Writes the current value to two addresses.

Video

VIDL	Set the sync up.
VM	Set the video mode.

Setting Slave Mode Register (SMR) Bits

BREAK/NOBREAK	Controls the break bit.
STEP/NOSTEP	Controls the step bit.
GO/STOP	Controls the clock enable bit.
ZERO/NOZERO	Controls the zero bit.
RESET/NORESET	Controls the reset bit.
JAM/NOJAM	Controls the jam bit.
MASTER/NOMASTER	Controls the master bit.
1D/2D	Controls the addressing mode bit.
VRAM/MCR/SR/REG	Controls the memory type bits.
INITSMR	Writes 0 to the slave mode register.

Setting Monitor Attributes

A	Sets the current address.
C	Sets the repeat count.
QUIET	Prints small set of informational messages.
V	Sets the value to write.
VERBOSE	Prints all informational messages.

Memory Checks

DP	Does pattern checking on a range of memory.
DPMC	Does pattern checking specifically for program (instruction) memory.
UA	Does a unique address check on a range of memory.

Miscellaneous

G	Goes to an instruction address.
H or ?	Prints this help message.
L	Loads an assembly program.
S	Provides status information.
ACI	Loads an assembly code instruction.
INIT	Initializes the TAAC-1.

Useful Functions

1. *VM - Sets the Video Mode.* You can select :
   - TAAC-1 video (the TAAC-1 takes over the entire screen).
   - Sun video (the Sun takes over the entire screen).
   - Mixed video (TAAC-1 video in a Sun window).
2. *INIT - Initializes TAAC-1 Registers.* It sets the video mode (VM) to Sun video.
3. *SR, VRAM, REG, MCR - Select the TAAC-1 memory type for reading and writing.* Only one memory type can be accessed at a time. Choices are:

SR	Scratchpad memory.
VRAM	Data/image memory.
REG	Video registers.
MCR	Program memory.

4. *A - Sets the Current Address* in the selected memory type. Use an address relative to the start of that memory type. For example, SR memory begins at 0x30000000. Therefore, the first SR address would be zero.

- To read memory, type R, which returns a hex value, or RF, which returns a floating point value. RN reads the next address; RP reads the previous address. RN and RP return hex values only.
- To write to memory:

  Type V to enter the value to be written.
  Type W to write the value.
  Type WN to write the same value to the next address;
  Type WP to write to the previous address.

5. *S - Displays Status Information.* Status information includes the current address, value, and SMR bit settings.

6. *L - Loads a TAAC-1 Program.* The sequence to use for loading a TAAC-1 program is:

ZERO	Holds the processor at zero. If the processor is running (the GO bit is set), it will execute instruction 0 repeatedly.
GO	If the processor is stopped, type GO to start it.
L	Loads the program.
NOZERO	Begins program execution.

7. *VIDL - Initializes the Sync Registers.* In response to the `filename` prompt, enter the full pathname of the appropriate sync file. Refer to *TAAC-1 Application Accelerator - Software Installation Guide* (800-2441-xx) for information about sync files. INIT also initializes the sync registers, so if you call INIT, you do not need to use VIDL.

9.10. `taprof`, TAAC-1 Profiler

`taprof` analyzes TAAC-1 program execution, to help you determine where a program is spending most of its time, so that those parts of the program can be optimized. The profiler provides information concerning the percentage of time spent in a given subroutine or over an arbitrary range of addresses. `taprof` can be run only on systems that support SunView.

Command Syntax

To invoke the profiler, enter

```
% taprof  [-c -s -d]
```

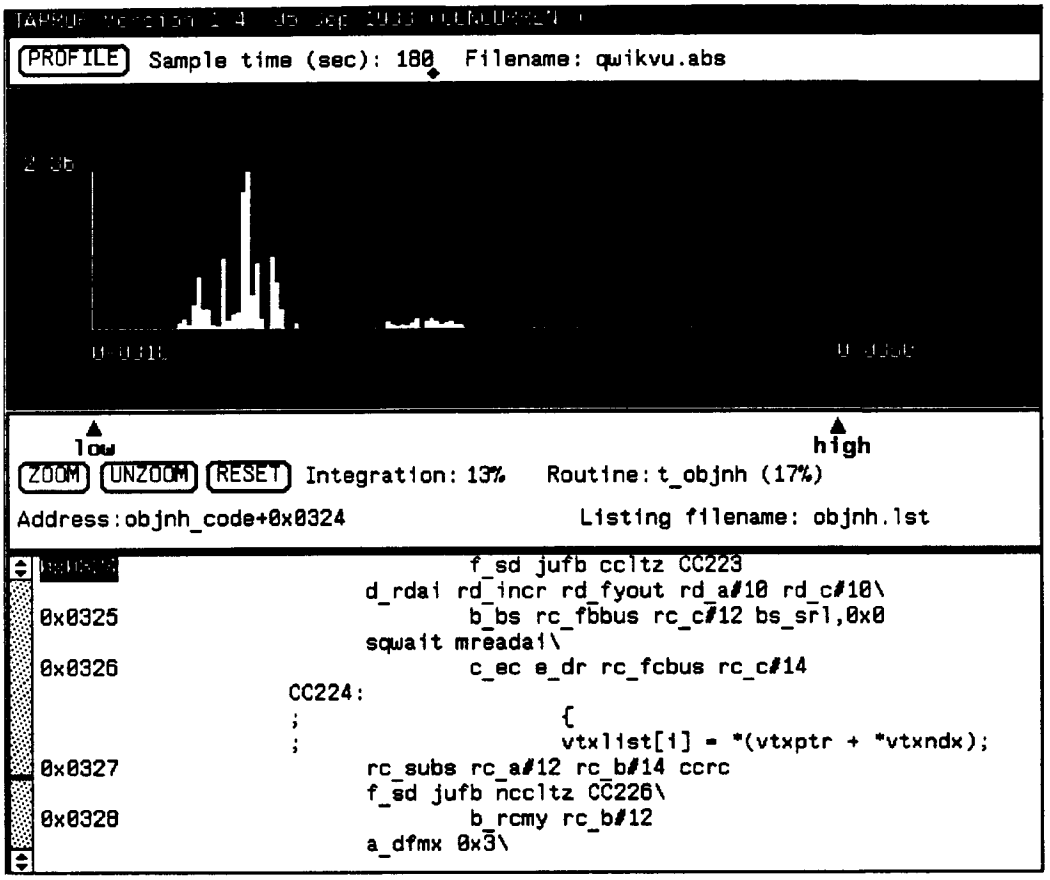
Where:

- `-c` specifies concurrent mode (the default), in which the profiler runs concurrently with a host program/TAAC-1 program. In this mode, data is collected at a rate of 50-100 samples per second. The host/TAAC-1 program is invoked separately, either before or after invoking `taprof`.
- `-s` specifies standalone mode, in which `taprof` loads and runs a TAAC-1 program without a host application program. In this mode, the sample rate is 400,000-500,000 samples per second. Standalone mode produces an accurate profile more quickly than concurrent mode; however, this mode requires that the TAAC-1 program run by itself, which may require some rewriting. Currently, standalone mode sets Sun-only video for single-monitor configurations; therefore, you will not be able to see the program running in a `tatool` window while the profiler is executing.
- `-d` specifies debug mode, which can be used to test the user interface. This mode does not run or profile a TAAC-1 program; it can be invoked on workstations without a TAAC-1.

User Interface

When invoked, `taprof` creates a Sunview window, as shown in the figure on the next page. The top half of the window displays profiling information; the bottom half, a listing file, if one is available.

Figure 9-3 *taprof Window*



Controls

Filename

Enter the full pathname of the TAAC-1 program being profiled. Pressing the right mouse button in this field causes a list of `.abs` files to be displayed. Move the mouse to highlight the file you wish to select and release the mouse button. If `taprof` is executing in standalone mode, pressing the Profile button causes `taprof` to download this file and begin execution. In either standalone or concurrent mode, the profiler looks in the specified directory for the `.map` file corresponding to the TAAC-1 absolute filename. It uses the `.map` file information to display subroutine names and listing files.

Sample time

Enter the number of seconds that you wish the profiler to gather samples. In standalone mode,

a Sun 3/160 gathers approximately 400,000 samples per second, and a 3/260 and 4/260 gather approximately 500,000 samples per second.

Profile

When you press this button, the profiler loads and runs the TAAC-1 file specified in Filename (in standalone mode) and begins gathering samples. When the sample time has elapsed, it displays the profiling information, including a graph of vertical lines representing relative frequency spent at each instruction address within the region of interest (see Low and High Pointers). If the region of interest is large, each line on the graph will represent more than one address. For greater precision, use the Zoom button to decrease the address range.

Scale Number

Although not generally useful, the scale number is displayed in the upper left corner of the graph. It indicates the percentage of time spent in the tallest bar visible in the display.

Low and High Pointers

The low and high pointers define the current region of interest to be within the address range specified. To move the low pointer, position the cursor (within the profile graph display) at the appropriate point and press the left mouse button. To move the high pointer, position the cursor and press the middle mouse button.

Integration

Integration displays the percentage of time spent within the current region of interest.

Routine

The routine display shows the name of the routine to which the cursor is currently pointing (based on the `.map` file information), and the percentage of time spent in that routine.

Zoom Button

When you press the Zoom button, the region of interest selected by the Low and High Pointers becomes the complete viewed area. This allows you to select progressively narrower regions of interest.

Unzoom Button	The Unzoom button undoes the effect of the previous Zoom. Zoom uses a stack which allows multiple zooming and unzooming.
Reset Button	The Reset button returns the display to the original full range.
Address	The Address display shows the address currently pointed to by the cursor, in terms of a code segment name and an offset from the start of that code segment. When the region of interest is large, several lines of code will be mapped to the same vertical line. In that case, the offset will be to the lowest address represented.
Minimum and Maximum Address	At the base of the profile graph, `taprof` displays the low and high boundaries (as absolute addresses) of the area being displayed.
Listing Filename	Displays the name of the file displayed in the listing file window.
Listing File Window	If `taprof` finds the appropriate listing file, it displays the listing file for the address pointed to by the Low Pointer, and highlights the listing file address for that instruction. Again, if the region of interest is large, this will be the lowest of a sequence of addresses. If there is no listing file corresponding to a selected address, the Listing File Window is blank. Note that, unlike the Routine and Address displays, the Listing File Window changes only when you change the Low Pointer, not every time you move the cursor.
Menu	The Zoom, Unzoom and Reset functions are also available through a menu selection. To see the menu, press the right mouse button.

9.11. taread, read TAAC-1 Data/Image Memory

`taread` writes a 32-bit image file (in IFF format) from TAAC-1 data/image memory to standard output. To invoke `taread`:

```
% taread [-x xoffset] [-y yoffset] [-X xsize] [-Y ysize]
```

Where:

- `xoffset` and `yoffset` specify a starting address, in TAAC-1 data/image memory, from which the image will be read; the default is (0, 0).
- `xsize` and `ysize` specify the image size in pixels. The default is (512, 512).

You can pipe the output of taread through the utility `iffreorder`, to extract and/or reorder the bands of an image. For example, to produce an 8-bit image file from the red channel of TAAC-1 memory (using default starting address and size):

```
% taread | iffreorder -b 3 > myfile_red.iff
```

To extract a 24-bit image (BGR), again using the default starting address and size:

```
% taread | iffreorder -b 1 -b 2 -b 3 > myfile_bgr.iff
```

See Also

`taload`, load rectangular region of TAAC-1 data/image memory, for descriptions of IFF format, the `taload` utility, and associated IFF utilities.

9.12. tarun, TAAC-1 Program Execution Tool

The utility `tarun` provides a simple means of loading and executing a program on the TAAC-1.

Command Syntax

To invoke `tarun`, enter:

```
% tarun [-v] filename.abs
```

- `-v` prints out the `tarun` version number and date.
- filename.abs is an absolute load file produced by the TAAC-1 linker `talink`.

Note that `tarun` does not initialize the TAAC-1. To initialize before running `tarun`, use `tainit`.

Function Calls

`ta_open()`	Opens communication with the TAAC-1.
`ta_run()`	Stops the TAAC-1 processor, loads the `.abs` file, resets the program counter and stack pointer, and begins execution.

9.13. `tashow`, TAAC-1 Show Tool

The utility `tashow` is used to switch from Sun to TAAC-1 video. `tashow` is useful in single-monitor configurations for quick looks at TAAC-1 data/image memory. In addition, `tashow` permits selection of TAAC-1 video from data/image memory bank A (0<=x<=1023, 0<=y<=1023), bank B (0<=x<=1023, 1024<=y<=2047), or from any specified y address in TAAC-1 data/image memory.

Command Syntax

To invoke `tashow`, type:

```
% tashow [-abstmwnvZzq] [-y yaddress]
```

Where:

- `-a` and `-b` specify display of the A or B banks, respectively. These take effect in TAAC-1 video. The default is bank A.
- `-t`, `-s`, and `-m` specify display of TAAC-1, Sun, or mixed video, respectively. The default is TAAC-1 video.
- `-y` specifies the y address of TAAC-1 video to be displayed at the upper left corner of the Sun monitor. The value of the y address must be between 0 and 2047 (inclusive), where 0 is the top of TAAC-1 image memory and 1024 is the beginning of bank B.
- `-w` and `-n` specify vertical wrap and no wrap, respectively. This has effect only if the `yaddress` option is invoked. If wrap is selected, the TAAC-1 video displayed will wrap within the bank containing the specified y address. If `nowrap` is specified, the display will not wrap within a bank, but will wrap at line 2047 back to line 0.
- `-v` prints out the `tashow` version number and date.
- -Z zooms the display by a factor of two in both the x and y directions. `tashow` uses the hardware zoom feature of the TAAC-1; pixels are replicated beginning with the upper left corner of the display (location 0, 0).
- `-z` turns off hardware zoom.
- `-q` turns off interactive mode. This option allows you to set up an alias that performs one or more `tashow` functions and then exits `tashow`.

IMPORTANT NOTE: If you display TAAC-1 video in interactive mode, be sure not to move the cursor from the window in which the command was entered. If you do it will be difficult to give the command to return to Sun or mixed video or to exit `tashow`.

The TAAC-1 reset button, located on the backpanel, will always restore video for single-monitor configurations.

Entering `tashow` with no arguments is equivalent to specifying bank A and TAAC-1 video.

Interactive Commands

When `tashow` is invoked without the `-q` option, it is in an interactive mode, accepting single key entries (each followed by `Return`). This should make operation simpler when using TAAC-1 video control. `tashow` recognizes these characters (organized by function):

a	bank A
b	bank B
y	y address displayed at top of Sun monitor
s	Sun video control
t	TAAC-1 video
m	mixed video
w	vertical wrap within a bank
n	no vertical wrap within a bank
Z	turn on hardware zoom by two
z	turn off hardware zoom
q	quit

All other characters are ignored.

9.14. `tatool`, TAAC-1 Tool

In single-monitor configurations, the utility `tatool`:

- Provides a quick SunView window into TAAC-1 data/image memory using the TAAC-1 video keying feature. This can be useful if you want to get started with the TAAC-1 without first developing a background in SunView.
- Provides an interactive method for video keying setup, using the Video Editor window.
- Optionally loads and starts execution of TAAC-1 load files while simultaneously supporting the TAAC-1 image window.

In dual-monitor configurations, `tatool` provides a simple method of identifying dual-monitor operation as the default and selecting the new video and sync formats.

Command Syntax

`tatool` is a window-based utility is invoked by entering:

```
% tatool [filename.abs][-t][-v]
```

Where:

- `filename.abs` is an absolute load file produced by the TAAC-1 linker, `talink`, which will be loaded and started on the TAAC-1. The `-t` option disables TAAC-1 communications, preventing the loading and execution of TAAC-1 load files.
- `-t` creates a host window for TAAC-1 images, with no TAAC-1 communication link. Since the TAAC-1 can be controlled by only a single Unix process at a time, this option creates a viewing window for TAAC-1 video that coexists with other Sun/TAAC-1 processes that may not include windowing software.
- `-v` prints the `tatool` version number and date.

More on the `-t` Option

The `-t` option is used to differentiate between a `tatool` window that is actively linked to the TAAC-1 and one that is not.

Without the `-t` option, a single area of TAAC-1 data/image memory tracks the host window. When you move the host window, the TAAC-

1 image in the window stays the same. A TAAC-1 image memory coordinate (generally 0,0) is tied to the top left corner of the host window. Using this mode, you can invoke the Video Editor to select single- or dual-monitor modes, adjust video keying, change image centering and select new video sync parameters.

With the `-t` option, the TAAC-1 data/image memory is fixed with respect to the upper left corner of the screen, instead of the top left corner of the host window. Therefore, moving the host window displays a different part of the TAAC-1 video frame. It should be noted that the TAAC-1 video frame is often partially blanked (or black), so that roaming around with the `tatool` window may not reveal all the contents of TAAC-1 data/image memory. You *cannot* invoke the Video Editor in this mode.

Since there is absolutely no communication with the TAAC-1 when using the `-t` option, TAAC-1 video control must already be in the mixed video mode, with the keying registers properly loaded, for TAAC-1 video to be inserted into the `tatool` window. If TAAC-1 video control is in external video mode or the keying registers are not properly initialized, the window will show only the keying color defined for the host canvas in the file `$TAAC1/hardware/taconfig.<hostname>`.

Full-screen keying mode

When you invoke `tatool -t` and make the `tatool` window iconic, moving the cursor into the icon results in "full-screen keying" mode, in which TAAC-1 video appears over the entire screen. When you move the cursor out of the icon space, normal Sun video returns. Points to note about full-screen keying mode:

- When you use the right mouse button to select the frame menu from the tatool icon, the TAAC-1 video will appear to "bleed through" the Sun video. This is an effect of the implementation and not a hardware problem.
- To use this full-screen keying mode on systems equipped with a 10-bit frame buffer, you must start SunView with the `-8bit_color_only` option. Otherwise, the TAAC-1 video will not be visible where the overlay colors are being used. For more information about the `-8bit_color_only` option, see the SunView man page.)

X windows version

The directory `$TAAC1/demo/src/txtool` contains source code and a Makefile for an X-windows version of `tatool`. `txtool` is a preliminary version only; it does not understand all the command line arguments that most X applications understand. However, it incorporates most of the features of `tatool`, with these exceptions:

- you must use the right mouse button and the pull-down menu, rather than the Escape key, to invoke the Video Editor;
- the Video Editor does not allow you to switch between single- and dual-monitor configurations. You must edit the appropriate sync file, instead.

The Makefile for `txtool` requires the X libraries to be visible to your machine. "Make install" builds `txtool` and installs it in the directory `$TAAC1/demo/bin`. To invoke `txtool`:

1. Log in; do not start up the Suntools environment.
2. Enter

   ```
   % xinit; kdb_mode -a
   ```

3. When the X console window starts up, enter

   ```
   % txtool [-geometry geometry_specs ]
   ```

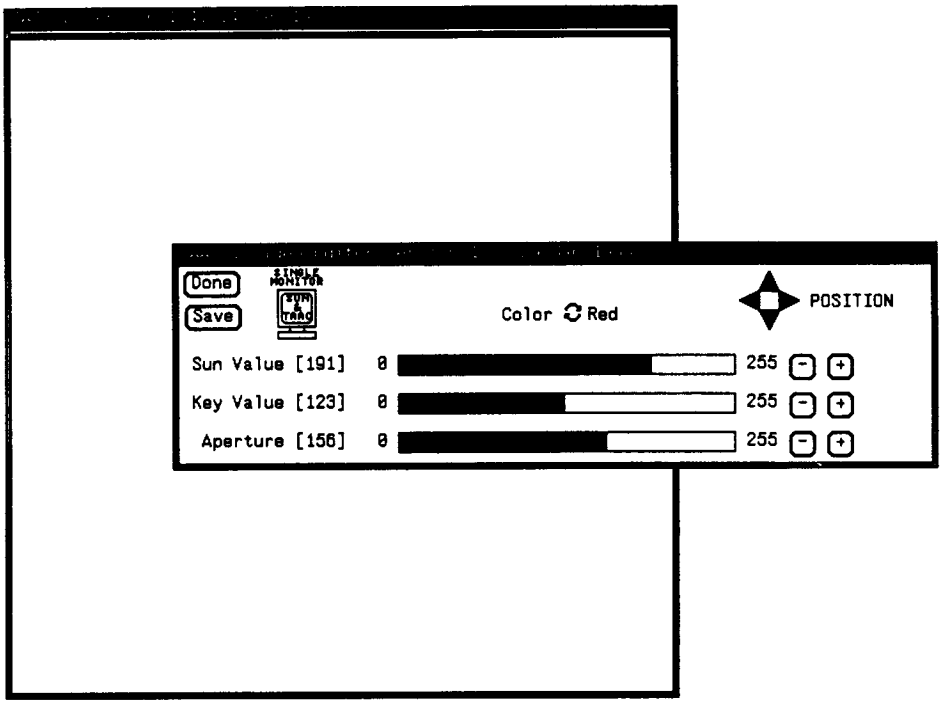
   `txtool` understands standard X-window geometry specifications. If you omit the `-geometry` option, use the mouse buttons to place and size the window..

The Video Editor

If `tatool` is initiated *without* the `-t` option, the Video Editor window can be used to set up video keying in single-monitor configurations. In dual-monitor configurations, the Video Editor permits the switch to dual-monitor operation and lets you enter the sync source and video formats of your second monitor.

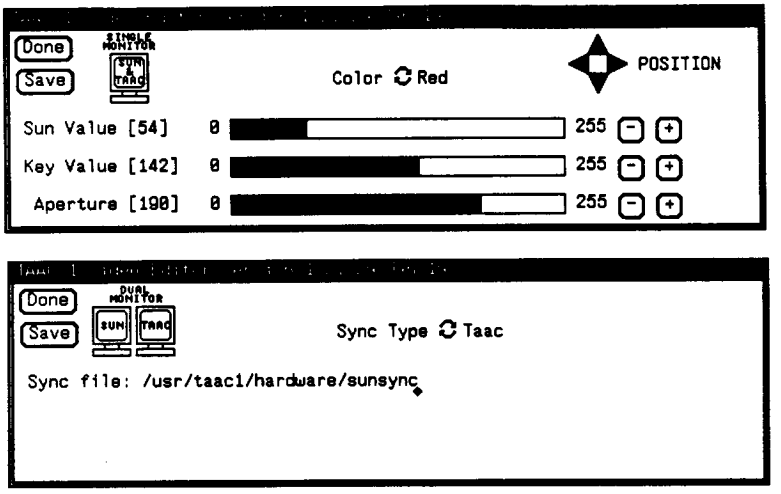
To call up the Video Editor, execute `tatool`, position the cursor within the `tatool` window display area, and press the <Escape> (Esc) key. The windows shown in the following figure will appear on screen:

Figure 9-4 `tatool` *Window and the Video Editor*



The Video Editor window looks different in single- and dual-monitor configurations, as shown in the next figure.

Figure 9-5 *Video Editor Windows - Single- and Dual-Monitor Versions*



Summary of Video Editor Controls

The left side of Video Editor contains three button controls. To activate these controls, position the cursor over them and click with the left mouse button. The button controls are:

Done	Removes the Video Editor. Any newly set keying parameters remain in effect while the window is open; the parameters are not saved to the configuration file.
Save	Saves the Video Editor parameters to the configuration file located in `$TAAC1/hardware/taconfig.<hostname>`.
Single/Dual	Switches between single- and dual-monitor operation and between the two Video Editor formats. This button is actually represented with one or two monitor icons.

In the single-monitor Video Editor window, you will also find a four-button compass used to center TAAC-1 video in the host window. To set keying values, use the Color cycle symbol and the three sliders (Sun Value, Key Value, Aperture) with associated increment/decrement buttons.

In the dual-monitor Video Editor window, the Sync Type cycle symbol is used to select internal or external (genlock) sync. A Sync File prompt provides a text field for entering the full file specification of the

sync file or you can select a sync file from all the sync files (Sun, NTSC, hires, etc.) found in the directory `$TAAC1/hardware`, using a pull-down menu and the right mouse button.

Seeing TAAC-1 Video in a Single-Monitor Configuration

There are two ways to see TAAC-1 video in a single-monitor configuration:

- Display TAAC-1 video only, by switching video control. This approach has the distinct disadvantage of forcing you to work blind in the Sun Unix environment, which can be impossible if your cursor gets moved off the switch or out of the active window that caused the transition to TAAC-1 video. If this happens, press the TAAC-1 hardware reset button located on the TAAC-1 backpanel to return to Sun video control.
- Create a host window within which TAAC-1 video is inserted using video keying parameters set up with the Video Editor. In general, this is the only reasonable approach to single-monitor operation with the TAAC-1.

While you will probably want to create your own windows using SunView software or other alternative windowing libraries, `tatool` provides an easy means of creating a host window with inserted TAAC-1 video. The `tatool` window can be positioned and sized like any Sun window. To quit the window and the process, position the cursor at the top of the window, press and hold the right mouse button, pull down the standard Sun menu, and select Done.

Keying Setup: Single-Monitor Operation

Video keying on the TAAC-1 is an analog process determined by an RGB keying color in a host window and three definable windowed thresholds on the TAAC-1. Ideally one would like the TAAC-1 to switch on a single unique host color (e.g., R = 100, G = 31, B = 88), but this is not possible in the analog world. We therefore must accept a set of host video levels around the nominal color which will cause the insertion of TAAC-1 video.

The entire keying setup process is driven by the goal of picking a unique class of color in the Sun windowing environment upon which the TAAC-1 can key and making this set of colors as small as possible so as to not unduly reduce the colors available to the remainder of the host

display. The selection and adjustment of the keying parameters is an iterative process of adjusting the RGB key values against a host canvas color, tightening the aperture to reduce key windowing tolerance, and repeating the process until acceptable keying is achieved.

`tatool` provides two standard test patterns (color bars and a gray scale wedge) for use in the video setup process. To call up a test pattern, position the cursor within the `tatool` window outside the Video Editor panel and press the right mouse button. A pop-up menu appears that allows you to select either a 512 x 512 color bar pattern or 512 x 512 gray scale pattern. When selected, these patterns are written to TAAC-1 image memory (0,0 to 511,511), overwriting any existing data.

Video Keying Controls

The Video Editor has three large slide bar controls. The top slide bar (Sun Value) controls the canvas color of the host window. The lower two slide bars (Key Value, Aperture) control the TAAC-1 keying color and the size of the TAAC-1 key color window or aperture.

The color slide bars (Sun Value and Key Value) can be used to adjust the red, green and blue color components independently or collectively and equally. The Color cycle symbol, located above the sliders, is used to select the color component(s) affected by the slide bars. To select a color component for adjustment, position the cursor over the round cycle symbol and click the left mouse button. Choices include red, green, blue and gray.

The gray color adjustment allows simultaneous and equal adjustment of red, green, and blue component values, allowing only grays for the host canvas and TAAC-1 key values. This mode is required to initially set up keying, since it is infinitely easier to adjust one value for a keying match than to simultaneously find three independent values.

To adjust slide bar values, position the cursor over the bar, press and hold the left mouse button, and move the slide bar to a new position. Releasing the mouse button "locks in" the adjustment. The "+" and "-" button controls to the right of each slide bar permit incremental adjustments of the slide bar values. To use these button controls, position the cursor over the button and click the left mouse button.

Adjusting Video Parameters

It is best to go through the video keying process with a known image in

TAAC-1 memory so you can actually judge the success and quality of the keying adjustment. Begin by bringing up the Video Editor with the <Escape> (Esc) key. Then use the right mouse button, with cursor still positioned over the main `tatool` window, to select the color bar image from the pop-up menu. If the color bar image does not appear, the TAAC-1 video is not properly keying and the setup procedure must be followed. Even if the color bar image does appear, it is generally best to proceed with the adjustment to optimize or tailor the keying parameters to the user environment.

Gray Adjustment

When the TAAC-1 image is not visible in the `tatool` window or when you wish to begin the keying adjustment from scratch, it is important to start the interactive process in the gray mode with a large aperture.

1. Set the Aperture slide bar to approximately 200.
2. Set the Color cycle to gray and set the Sun Value to a value near 128.
3. Move the Key Value slide bar until the TAAC-1 image appears in the host window. The key value will in general be larger than the Sun Value. If keying of TAAC-1 video into the host window does not occur, increase the Aperture value and try again.
4. Once keying begins, note that a range of key values exists over which TAAC-1 video is visible. This is the analog nature of the process. Without trying to be precise, center the Key Value in this "good" key range as well as is possible using either the slide bar or the associated increment/decrement buttons.

RGB Keying

Keying is now operational using a fairly wide aperture. Since many users will be using monochrome Suntool windows, it is generally best to pick a final host window color other than gray to achieve good keying. A distinct separation of RGB components, such as R = 128, G = 64 and B = 192, is recommended. The exact color must be chosen by the user to avoid keying conflict with colors used in other host window processes.

The final adjustment phase dials-in your selected color, *one component at a time.*

5. Change the Color cycle symbol to red and set your selected host red value. The video keying will likely be interrupted until you

readjust the red Key Value, again centering the slide bar in the keying range.

6. Change the Color cycle symbol to green to set up the user-selected Sun green value and repeat the above procedure for the green Key Value.
7. Repeat the procedure for the blue value.
8. To minimize the number of host colors which are keyable, reduce the aperture and repeat the Key Value slide bar adjustment for the red, green, and blue TAAC-1 key values. Repeat this procedure two or three times or until you are satisfied with the video keying. A final aperture setting of 130 to 180 is typical.
9. Once the keying parameters are adjusted to your satisfaction, click over the Save button. This selection writes the parameters to the configuration file:

   `$TAAC1/hardware/taconfig.<hostname>`

   The next time `tatool` or other TAAC-1 programs are started, this file will be read and the keying/video parameters will be initialized.

   One way to test your keying settings is to move the cursor into the `tatool` display window to judge the quality of the key around the small arrow symbol. If a "ringing" or noisy trail exists, continue the iterative adjustment process to eliminate or minimize its effect. Once you are satisfied with keying, check the video centering using the next procedure.

Video Centering

To center the video:

10. Display one of the 512 x 512 test patterns generated by `tatool`. The test pattern should precisely fit the display window.
11. Position the cursor over the left/right or up/down buttons, and click the left mouse button to center the image in the display window.
12. Once the centering parameters are adjusted to your satisfaction, click over the Save button. This selection writes the parameters to the configuration file:

    `$TAAC1/hardware/taconfig.<hostname>`

    The next time `tatool` or other TAAC-1 programs are started, this

file will be read and the centering will be initialized.

Windowing Library

`tatool` uses a small library of windowing routines included on the distribution tape. Refer to the description of the unsupported window operations in the demo chapter for information on the windowing routines. The *SunView Programmer's Guide* is also helpful.

Seeing TAAC-1 Video in a Dual-Monitor Configuration

While the dual-monitor configuration is the most direct approach to viewing TAAC-1 video, it is necessary to modify the TAAC-1 configuration file to identify the default mode of operation (dual monitor), the video format (Sun, NTSC, etc.) and sync source (internal or external). The `tatool` Video Editor can be used in this capacity at initial installation or to re-configure the site defaults.

If the Video Editor window appears in the single-monitor format, position the cursor over the monitor icon and push the left mouse button to switch to dual-monitor format. In dual-monitor format of the Video Editor window, there is a Sync Type cycle symbol used to select internal or external (genlock) sync. A Sync File prompt provides a text field for entering the full file specification of the sync file, or you can select a sync file form all the sync files (Sun, NTSC, hires, etc.) in the directory `$TAAC1/hardware`, using a pull-down menu and the right mouse button. Current video formats include:

`/sunsync`	Normal sun video, 1152 x 900, 66 Hz non-interlaced
`/ntscsync`	RS-170, 648 x 486 active, 30 Hz interlaced
`/hiressync`	High Resolution, 1024 x 1024, 60 Hz non-interlaced

Others video formats may be supplied for unique monitors. Please check the directory $TAAC1/hardware and the comments within individual files for specifications.

In summary, to setup dual-monitor operation from the Video Editor:

1. Select dual-monitor operation by clicking on the single-monitor icon button.
2. Select Sync File from the pull down menu of available video formats which matches your dedicated TAAC-1 monitor.
3. Select Sync Type to be internally generated TAAC-1 sync or

external applied sync in which case the TAAC-1 genlocks to incoming sync (see hardware installation for cabling details).

4. Save the results to the file:

   `$TAAC1/hardware/taconfig.<hostname>`

   by pressing the Save button.

9.15. `tatxt2o`, Convert Text File to Sun Object File

The utility `tatxt2o` converts a text file to a host object file so that it can be linked into a host program. It creates a null-terminated string with the same result as if you had declared:

```
char ta_helptxt[] = "ENTIRE CONTENTS OF FILE";
```

This utility was designed for use in conjunction with the unsupported demo window library routine `ta_help()`. `ta_help()` displays the contents of the global variable `ta_helptxt[]` in a pop-up text window.

The object file (`.o`) created by `tatxt2o` should be added to the object file list in the link command. Only one text file can be compiled and linked into a host program.

For more information, see `ta_help()` in the demo description chapter.

Command Syntax

To invoke `tatxt2o`, type:

```
% tatxt2o input_file output_file
```

Where:

- `input_file` specifies the input text file.
- `output_file` specifies the output Sun object file.

TAAC-1 User Guide: Revision History

This page helps you keep track of changes to this manual. If you do not know if your manual is current, use this table to confirm that all updates have been added:

Revision History Table

Revision Date	*Pages Removed*	*Pages Inserted*
15 September 1989		Complete revision
15 April 1989	ix, x	ix, x
	3-9, 3-10	3-9, 3-10
	3-17, 3-18	3-17, 3-18
	3-19, 3-20	3-19, 3-20
	3-27, 3-28	3-27, 3-28
	3-53, 3-54	3-53, 3-54
	4-5, 4-6	4-5, 4-6
	4-7, 4-8	4-7, 4-8
		4-7.1, 4-7.2, 4-8
	4-9, 4-10	4-9, 4-10
	6-3, 6-4	6-3, 6-4
	6-5, 6-6	6-5, 6-6
	8-27, 8-28	8-27, 8-28

TAAC-1 User Guide: Change Pages

TAAC-1 software release 2.3 includes a complete revision of the *TAAC-1 Software Reference* and a set of change pages for the *TAAC-1 User Guide*. These instructions show you how to insert the new pages into the *User Guide*. The update table describes the nature of the changes.

A revision history page is also provided, for addition to the back of the *User Guide*.

TAAC-1 User Guide Update Summary		
Remove These Pages	*Add These Pages*	*Description of Change*
ix, x	ix, x	new utilities chapter 9 contents
3-9, 3-10	3-9, 3-10	new restrictions on use of ta_map and ta_use_map
3-17, 3-18 3-19, 3-20	3-17, 3-18 3-19, 3-20	addition of usleep() to host program
3-27, 3-28	3-27, 3-28	addition of usleep() to host program example
3-53, 3-54	3-53, 3-54	change to fscanf() arguments in poly.c example
4-5, 4-6 4-7, 4-8 4-9, 4-10	4-5, 4-6 4-7.1, 4-7.2 4-8, 4-9, 4-10	new description of -fsingle functions returning structures now supported
6-3, 6-4 6-5, 6-6	6-3 – 6-6	new -g option to talink
8-27, 8-28	8-27, 8-28	added missing line in double precision math example
(continued)		

TAAC-1 User Guide Update Summary, continued		
Remove These Pages	*Add These Pages*	*Description of Change*
Chap. 9, all	Chapter 9, all	new Image File Format (IFF) utilities changes to taload, taread to support IFF format tadeb: how to access local variables taabs2o: multiple .abs files