TigerSHARK

A Hardware Accelerated Ray-tracing Engine

Greg Humphreys and
C. Scott Ananian

Independent Work
Presented to the
Department of Computer Science
at Princeton University

Advisors:
Andrew Wolfe and Doug Clark

May 14, 1996

This paper represents my own work in accordance with University regulations.

Contents

1 Introduction

The success of the Disney/Pixar film *Toy Story* has catapulted photorealistic computer graphics to the headlines, but the recent hype has, for the most part, obscured the technical details of the feat. Despite the hue and cry, the difference between *Toy Story* and previous entertainment-industry use of computer-generated effects is merely one of scale. The 10-year-old techniques used by Pixar sacrifice image realism and fidelity for the sake of speed. Even so, Pixar's farm of Sun workstations took a total of over 800,000 hours to perform the final render. The huge amounts of compute time necessary to produce realistic images have prevented more realistic techniques from being applied.

We propose a low-cost parallel architecture for ray-traced graphics to help eliminate the processing bottleneck. Ray-traced graphics produce much higher-quality output than scan-converting renderers (the technology used for *Toy Story*) but are currently too slow for wide-spread commercial use. Most existing hardware research has therefore concentrated on scan-converting renderers, but fundamental algorithm and hardware limitations affect the amount of parallelism obtainable. Ray-tracing, on the other hand, is amenable to massive parallelism, given the proper architecture. We propose a distributed DSP[1]-based architecture which should be linearly scalable up to hundreds of DSPs. Each DSP could process over half a million ray-triangle intersections per second, and a PCI card hosting 16 DSPs would be capable of 8 million ray-triangle intersections per second. A Cambridge-based company has begun work on custom silicon to attack the ray-tracing problem, but we believe our approach is superior due to a specialized architecture and the use of reprogrammable Digital Signal Processors.

2 Rendering Techniques

The diverse applications of computer graphics allow a large variety of different algorithms and techniques to be used. For example, the proliferation of 3D game engines has given rise to new classes of high-speed algorithms which sacrifice output resolution and quality for real-time display. Slightly more demanding are CAD/CAM applications, which don't necessarily require interactive speeds, but demand the ability to view and manipulate complex environments easily. Computer graphics' high end is filled by entertainment industry companies like

[1] **D**igital **S**ignal **P**rocessor

Pixar and Industrial Light and Magic, who require extremely realistic output, regardless of the cost in rendering time or computational power.

2.1 Ray Tracing

One of the first successful image synthesis methods was ray-tracing [28]. Just as lensmakers would plot the path of light rays through a lens, the computer was used to compute the paths of light rays entering the eye. The light rays were followed backwards from the eye until they hit an object in the scene: it was then known that the light ray must have emanated from the object hit. The color of the ray entering the eye could then be computed, based on the color of the object the ray had been emitted from.

Although such ray-tracing systems are conceptually very simple, they occupy the high end of computer graphics applications. The large number of rays which must be traced backwards into the scene creates a huge compute cost. Ray-tracing produces extremely realistic images because it is based on a physical model: reflections, shadows, motion blurs, and other effects are generated easily from the basic ray-tracing technique. However, ray-tracing is prohibitively slow for production work.

2.2 Scan Conversion

Scan converting renderers, on the other hand, can be made to run extremely quickly, at the cost of decreased realism. Instead of tracing light rays through the scene until they hit the model, scan converting renderers work directly with the model primitives. These primitives (often polygons or triangles) are transformed first into 2D space and then projected onto the viewing plane. Thus every primitive only has to be considered a maximum of once, rather than many times as necessary with iterative ray-tracing techniques.

Scan-converting renderers, although widely used for CAD/CAM applications and in the film industry, cannot produce scenes as visually appealing as can a ray-tracer. Most implementations of scan converting renderers deal only with triangles or simple polygons because of the difficulty of scan converting complex primitives. While it is possible to decompose an arbitrarily complex primitive into component triangles, this results in a huge increase in the number of primitives in the scene, and makes accurate representation of curved surfaces difficult. Because scan-conversion and ray-tracing are based on fundamentally different algorithms, it is much easier to add complex primitives to a ray-tracer.

More fundamentally, scan conversion systems have no inherent ability to represent reflection or refraction. Addition of such features generally amounts to embedding a ray-tracing system into the scan converter. This drawback is most obvious when trying to create accurate shadows, vital for realistic imagery; such 'simple' effects spring naturally from the ray-tracing algorithm, but are very difficult to synthesize in a scan converting renderer.

One notable exception is the REYES[2] system [5], which Pixar used as part of PhotoRealistic RenderMan to create *Toy Story*. REYES can synthesize realistic scenes, but it does so at the expense of speed. It is not unreasonable to expect a RenderMan scene to take days to render on a modern workstation.

3 Renderer Acceleration

The large amount of parallelism inherent in rendering tasks makes multi-processing a natural solution for graphics acceleration. However, the type of parallelism and architectures best suited to exploit it are very different for scan conversion systems and ray-tracers. Most extant work in the field has focused on scan converters [6, 12]. In scan conversion, parallelism is, in general, achieved by distributing primitives to different processors. Several different stages in the rendering pipeline have been proposed as the appropriate place to do the communication [7], but all suffer from very poor worst-case performance. In addition, most existing parallel rendering systems require a shared memory or message passing MIMD computer, which is an extremely expensive investment. This paper proposes parallel ray-tracing as a means to avoid the communications bottlenecks of scan conversion, and proposes a new architecture tailored for ray-tracing.

3.1 Task Description

The computational task of a scan converter differs substantially from that of a ray-tracer. The fundamental operation of a ray-tracer is computing ray-primitive intersections. The basic task is to find the first object that intersects each ray from the eyepoint. More intersections are computed to create reflection, refraction, shadow, and other effects in the image. Over 95% of the compute time of a ray-tracer is spent computing ray-primitive intersections [9, 28], therefore this is the task first parallelized. Amdahl's law states that the other 5%

[2] **R**enders **E**verything **Y**ou **E**ver **S**aw.

of the problem will become increasingly important as the amount of parallelism utilized increases; a hierarchical approach will be introduced later which allows us to deal with this increase.

The basic task, then, is to simply intersect all primitives in a scene with a set of rays, outputting the (small) set of intersecting primitives. For a given ray projected into a scene, we can expect only about 3–5 intersections to be found [24, 28]. From the nearest intersecting object, we will generate new rays to cast into the scene to compute shadows, reflections, refractions, and other natural effects.

3.2 Existing Approaches

There are a number of software acceleration techniques which often influence parallel implementations, primarily various partitioning methods. Partitioning allows us to reduce the number of ray-object intersection calculations by excluding various objects from consideration, sight unseen. Lin [17] mentions two widely used techniques: *hierarchical bounding*, which partitions a scene into a tree of enclosing volumes [11, 14], and *space subdivision*, which divides the objects among a tree of uniformly sized spatial cells [10]. Only objects within the cells or bounding volumes through which a ray passes are tested against that ray. Scherson and Caspary [24] analyze the performance of such algorithmic improvements in depth; the performance gains realized are significant.

Parallel ray tracing is done either asynchronously using MIMD[3] architectures or synchronously using SIMD[4] architectures. MIMD computers are more general, and there is evidence that they are more efficient for scan converting renderers (see, for example, [6, 12]), and so most research on parallel ray-tracing has focused on MIMD architectures. We contend, however, that the hardware resources of MIMD machines are not used efficiently for ray-tracing. SIMD architectures, such as the one we propose, perform the task as effectively with much less hardware cost and overhead.

[3] Multiple instruction stream, multiple data stream.
[4] Single instruction stream, multiple data stream.

4 Survey of Ray Tracing Architectures

4.1 MIMD Ray Tracing

Lin [17] describes two categories of MIMD parallel ray tracers: those that use *image-space partition algorithms* and those that use *object-space partition algorithms*. In image-space partitioning, the pixels in the output image are divided among the processors, with a single processor responsible for the entire ray-tracing task for those pixels. This obviously requires the entire image database to be accessible to all processors. Lin claims that shared memory is essential to this approach; however, multi-computers can implement this scheme equally well without any kind of shared memory protocol if the image primitives are simply broadcast to all nodes at initialization. The image components can be reassembled at completion. For complex images, such as those rendered for *Toy Story*, the communication penalty will be negligible.[5] Instead, the chief disadvantage of this method is the large hardware overhead. The distributed or shared memory used to store the image database, disk storage, operating system overhead, etc, are needlessly replicated. However, this solution can be implemented off-the-shelf (as Pixar apparently did for *Toy Story*) with general-purpose hardware, which is an advantage in commercial environments. Cost estimates for TigerSHARK, however, indicate that special purpose hardware of equivalent performance should cost a fraction of the cost of the general purpose multi-computer network. The hierarchy of parallel techniques utilized in TigerSHARK incorporates image-space partition as its top-most level.

Object-space partitioning derives from the software spatial subdivision technique described earlier. Each processor is reponsible for all ray-tracing tasks within a spatial cell, and the object database is distributed among the processors according to that spatial division. All rays entering the cell are tested against the objects residing in the cell. Rays leaving the cell are passed off to the processor responsible for the cell it will be entering. Although shared-memory and memory replication are not issues, unbalanced loads, ray propagation overhead, and object fragmentation caused by objects spanning more than one cell create problems for this approach.

[5] For real-time processing of simple scenes, however, the bandwidth required for this approach is excessive; a shared-memory system (with higher inter-node bandwidth) will have to be used.

4.2 SIMD Ray Tracing

The graphics community has at times doubted the feasibility of using SIMD processors for ray tracing [19], but it has been shown [17] that SIMD approaches can be as efficient as MIMD designs. Most published architectures, however, are unable to take advantage of the performance benefits of the various partitioning schemes, and instead use a 'brute-force' approach, blindly comparing every ray against every object [19, 23]. Lin [17] gives the speed-up of existing brute-force SIMD architectures using N processing elements as

$$N\frac{(1+k)nf}{m}$$

where n is the average number of objects tested against each ray by the optimal sequential algorithm; m is the number of objects in the scene; f, $f < 1$, is the degradation caused by SIMD processing, and k is the overhead due to space traversal, for those architectures which use it. Since n approaches m as m decreases, this approach works well for simple scenes, but fails for the complex scenes more common in practice, where $\frac{m}{n}$ can be greater than 1,000.

Clearly, a practical SIMD approach needs to be able to use space-partition methods to reduce the number of intersection calculations. This is more difficult with SIMD processors: one would think that rays could not be tested against different partitions of the object database without multiple instruction streams. In fact, several methods for accomplishing this are available. Lin describes a method using a partially data-driven architecture in [17]. We propose another method, using a hybrid architecture which takes advantage of spatial coherence.

5 The TigerSHARK Architecture

Our goal in designing a parallel ray-tracing accelerator is to tune the architecture to match the problem. SIMD designs for this task use hardware more efficiently, and existing supercomputer architectures [20] have demonstrated the performance benefits of using large numbers of very simple processors. We strive for very low cost-per-node, eliminating excess generality, to faciliate the use of large numbers of nodes.

5.1 Overview

Our proposed hybrid architecture consists of a collection of synchronous processing nodes connected to a single data memory, which they access in lock-step. The processing nodes are more complicated than the typical SIMD ALUs; in fact, they are full-featured digital signal processors, with a small amount of on-board memory and serial communications ports.

All memory accesses must be performed together; each processor reads the same memory location at the same time. Writes are disallowed for all but one processor, termed the master. Rays are distributed via the individual serial communications ports, daisy-chained together, and object intersections are collected the same way. The single memory holds the object database.

Note that this is not a pure SIMD architecture. The DSPs can copy their program information from the shared memory to their small internal memories, so that they do not require an external bus access for an instruction fetch. They can then branch and loop independently from the other processors, on the condition that the next external bus access must again be synchronized. A hardware primitive is provided to perform the resynchronization.

In practice, the first operation performed by the DSPs is to copy their programs to internal memory. They then use the daisy-chained serial bus to determine unique node ids, designating node 0 the write-priviledged master. A host processor loads the shared memory with the object database, and then sends rays down the serial bus. In the simple case, each processor gets a different ray, and the rays are tested in parallel against the entire object database. Objects are, of course, fetched synchronously.

We saw above that such brute-force approaches (testing every ray against every object) suffer very poor comparative performance on complex scenes. This is avoided by utilizing the spatial coherence of the input rays and a simple voting primitive. 'Spatial coherence' refers to the fact that consecutive rays tested are extremely likely to be close together, spatially. For example, consecutive rays might belong to an adjacent pair of pixels in the output image. This spatial similarity means that they will also have similiar intersection properties: if the first ray intersects a primitive or bounding volume, it is very likely the next ray will, as well. Obviously, some forethought is required to ensure that most of the spatial coherence present in the original task (adjacent pixels, etc) is preserved in the ray ordering as seen by the DSPs.

If we can guarantee this spatial coherence, however, we can also implement

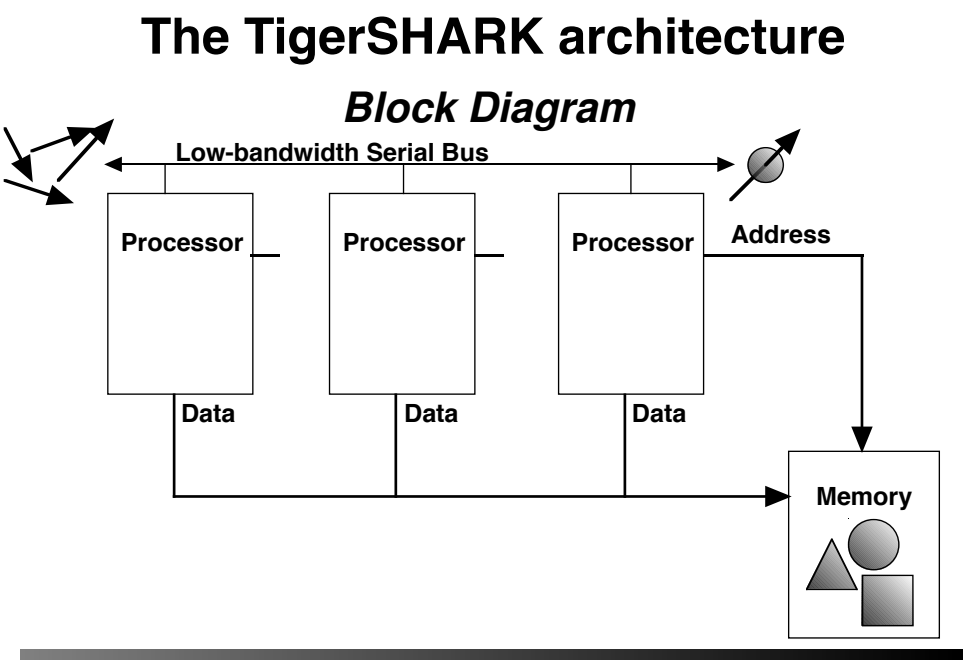

Figure 1: TIGERSHARK architecture block diagram

bounding-volume partitioning. Each ray is tested against a boundary volume, and then tells its neighbors whether or not it intersected. If none of the rays intersect the boundary volume then we need not examine the objects inside it. Otherwise, all the processors continue to examine the objects inside the bounding volume, since one of them may find an intersection. The degree of ray spatial coherence determines how probable a unanimous vote is.

This architecture requires very little hardware to implement a parallel system. Obviously, if the number of processors is greater than the amount of spatial coherence present, efficiency is poor. Anti-aliasing and stochastic sampling are two common techniques to improve image quality which utilize multiple rays per pixel. Large images (where the pixels are 'smaller' with respect to image variations) and multiple rays per pixel both provide large amounts of spatial coherence which can be utilized. Full implemention of the prototype system will provide a definitive measure of the amount of spatial coherence which can be exploited, but 16–32 processor systems should certainly be possible without

much performance degradation. We use a hierarchical organization to obtain further parallelism.

5.2 Hierarchical Parallelism

The prototype system is implemented as a PCI card hosted in a personal computer. The second level of parallelism is to add multiple processor 'cards' to the host. Each card contains a single object database memory, and a number of DSP processing elements. To improve latency times and reduce the amount of on-board memory required, we split up the object database among the PCI cards in the system. We then provide the same rays to each card (we needn't 'use up' any of our potential spatial coherency here), reading and merging the intersections found by the two cards. The limiting factor now is the host processor power. It was mentioned in section 3.1 that over 95% of the processing task is ray-primitive intersections; the remaining part of the task falls on the host processor. Obviously, as we continue to expand the system, we are going to want to provide multiple host processors, to avoid a bottleneck. The third level of parallelism is then to add multiple hosts, each with multiple PCI cards holding multiple DSPs. The task is divided up among the hosts using an image-space partition: each host gets a section of the final output image and the entire object database, renders independently of the other hosts, and assembles the final output image when the hosts have all completed. This image-space partitioning was discussed in section 4.1.

So the complete system contains three separate levels of parallelism: ray-vector parallelism at the lowest level, followed by object database distribution above it, and capped by an image-space partitioning scheme.

6 TigerSHARK System Hardware

6.1 Design Process

The first design parameter we set was the target performance level. We wanted to be able to render complex realistic scenes, on the scale of those created for *Toy Story*, at a better price-performance ratio than available with multi-computer arrangements. Complex scenes meant that we needed support for object database partitioning, and the recent announcement of Advanced Rendering Technologies' AR250-64 ray-tracing system [26] set our performance goal:

to be competitive, we should be able to compute close to 10^9 ray-triangle intersections per second. An initial estimate placed this as the compute power of a 512-DSP array; subsequent architecture design thus called for scalability to 512 processors. As will be discussed in section 7, actual system performance is not directly comparable on the ray-primitive level, since the AR250-64 system and TIGERSHARK use different sets of primitives; instead, scalability to 512 processors replaces raw compute speed as our performance target.

Digital signal processors seemed a good match to the low component-cost, high floating-point[6] performance required. General-purpose processors typically either required too much support hardware, cost too much, or provided too little floating-point performance to be viable options.

Various system bus options were then considered, starting with traditional shared-memory MIMD systems. The shared-memory hardware overhead and SIMD/MIMD issues did not generally justify the standard shared-memory architecture, but we examined a few variations on the architecture before discarding the idea. The Analog Devices SHARC DSP was considered as a processing node: it provides built in support for global-memory architectures, and includes substantial on-chip local memory (reducing necessary shared-memory bandwidth), but the high cost of this processor was a heavy disadvantage. Dual bus systems, like the Motorola DSP96002 processor, were evaluated as possible low-cost solutions to providing shared memory; however, the required support hardware and high processor cost prompted a search for other options.

Once the SIMD architecture was decided upon, it was fairly easy to select the Texas Instruments TMS320C32 60-MFLOP DSP on the basis of its extremely low cost (less than $10 each in quantity [13]) and excellent performance.

6.2 Hardware Description

An array of TMS320C32 DSPs are supplied with identical clock signals via a low-skew driver for synchronization. The model database is stored in zero-wait state SRAM, and the ray data input and intersection data output are via a high-speed serial port on the TMS320C32.

Only one DSP is connected to the address lines of the memory, since all DSPs are guaranteed to be driving the same address on any given cycle. Output from the SRAM is via a high-speed buffer to enable the SRAM to drive the data-bus

[6] Investigation was also done into the applicability of fixed-point processors: the results strongly favored floating-point; the reasons are outside the scope of this paper.

inputs of a large number of DSPs.

The DSP array is hosted on a PCI bus system; for the prototype, this is a Pentium PC. The PCI interface is via the AMCC S5933 'Matchmaker' chip; the host processor can directly load object database information into card SRAM, and a 32-word FIFO is used to buffer ray and intersection data coming to and from the device. The PCI interface is capable of bus mastering for added performance.

The TMS320C32 DSPs can execute 60 MFLOPS peak; estimated ray-tracing performance is on the order of half a million ray-triangle intersections per second (estimating worst-case 100 instructions per ray-primitive intersection).

A partial schematic of the system is shown in Figure 2.

6.3 Cost/Performance Issues

In order to keep the architecture inexpensive and uncomplicated we decided not to share the DSP bus with PCI. Originally, the model database was double-buffered to enable the PCI interface to update the model primitives without interrupting the DSPs. However, calculation showed that typical performance loss due to database updating remained under 1% for typical processor loadings,[7] so in the interest of reducing costs the double-buffering was eliminated. PCI access to the SRAM now requires the DSPs to yield the bus, but system costs are reduced almost 30%–40%.[8]

Our calculations also verified that the bandwidth available on the PCI bus was enough to sustain our target 512-processor system. Bus mastering and burst writes were employed to utilize the full bus bandwidth.

The size of the SRAM for model database storage was minimized subject to bandwidth and performance constraints. An overly small SRAM would create large amounts of paging traffic as parts of a bigger database are paged into local memory, but large amounts of fast SRAM are very expensive. It was found that 1Mb of SRAM was the smallest practical size for the target 512-processor implementation. The serial communication bandwidth turned out to be the limiting factor: a smaller SRAM would finish iterating rays through the object database before new rays could be transmitted to the DSPs.

[7] 100,000 rays iterated through object database between updates; 512-processor system

[8] RAM typically accounts for 60% of the cost of an average workstation [20]; the percentage is even higher for the TIGERSHARK board, since the processor cost is so low. Eliminating double-buffering allows us to use half as much RAM per board.

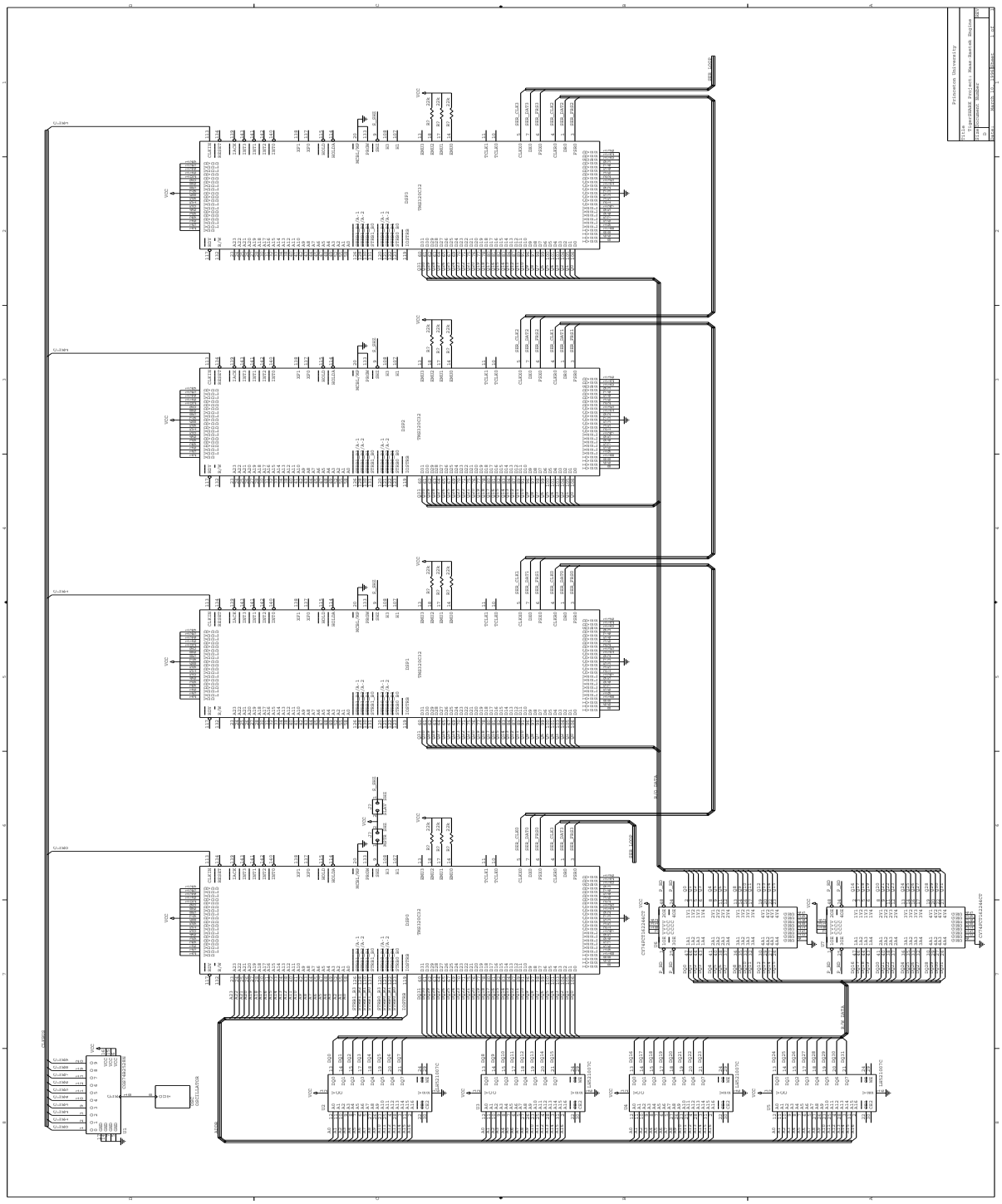

Figure 2: TigerSHARK Architecture

7 System Evaluation and Comparison

To evaluate the effectiveness of the TigerSHARK architecture, the system is compared against Advanced Rendering Technologies' AR250-64.[9] Advanced Rendering Technologies (ART) is a Cambridge-based company that announced on 2 October 1995 its plans to build a custom VLSI processor for ray-tracing applications. First samples are anticipated 4Q 1996. Although the goal of both TigerSHARK and ART is to accelerate ray-tracing, the relevant architectures are very different. It is unwise to make claims about the full 512-DSP TigerSHARK system until scalability can be empirically verified, so instead we will compare a single TigerSHARK PCI board to a single module of ART's AR250-64.

The AR250-64 is made up of 64 AR250 ray tracing engines. Each AR250 is claimed to be able to perform 80,000,000 ray intersection tests per second, "roughly 16 times the performace of the best graphics workstations." We compare this single AR250 with a TigerSHARK board containing 16 TMS320C32 DSPs.

ART's AR250 clearly holds the lead in raw speed. The AR250's custom silicon allows it to perform almost 4 GFLOPS, about 4 times more raw floating-point performance that TigerSHARK, and the AR250 can perform an order of magnitude more ray-trangle intersections per second than can TigerSHARK.

TigerSHARK makes up the deficit in flexibility, however. ART's decision to go for custom silicon entails sacrificing some complexity: in particular, the AR250 is only able to implement triangle primitives on-chip. All other primitives must be decomposed into triangles before they can be rendered. TigerSHARK's reprogrammable DSPs allow it to perform any type of ray-primitive intersection, which allows a lot of relative performance gain. For example, the AR250 must decompose a sphere in the object database into hundreds of component triangles, while TigerSHARK can process the sphere in one operation, as a single primitive. Moreover, for TigerSHARK ray-sphere intersections are actually easier to perform than ray-triangle intersections. TigerSHARK is therefore an order of magnitude *faster* than the AR250 on this task. On the assumption that most interesting scenes are fairly complex, with lots of non-flat surfaces that the AR250 will need to triangulate, a TigerSHARK board may very well equal or surpass the AR250's performance.[10]

[9] All data on Advanced Rendering Technologies' products is from [26].

[10] Furthermore, it is not clear that ART has implemented or can implement a partitioning

In addition, TigerSHARK has a much better price/performance ratio. ART is using a low-volume custom ASIC,[11] with the price penalty that that entails. TigerSHARK, on the other hand, uses high-volume TI DSPs to achieve a very low node cost.

Overall, we are optimistic that the TigerSHARK system will be able to match the performance of ART's custom silicon while maintaining a much lower price. Neither system has been fully prototyped yet; scalability is likely to be key to a comparison of operational systems. The results are certainly positive enough to justify a closer look at hybrid SIMD architectures for ray-tracing applications.

8 Conclusion

TigerSHARK was designed to achieve the three goals most prized by professional computer graphic artists: speed, cost, and quality. A high-end Silicon Graphics machine is fast, at the expense of cost and quality; a software ray-tracer can produce high quality output but renders extremely slowly; and hardware approaches to ray-tracing have thus far been extremely expensive due to their use of general-purpose hardware.

The TigerSHARK system provides extremely high quality output, scales well due to the task's inherent parallelism, and uses low-cost hardware extremely efficiently. We believe that a fully expanded TigerSHARK system would be able to rival the rendering speed of the fastest graphics systems available, at approximately three orders of magnitude less cost. Our hybrid architecture shows great promise for ray-tracing applications, challenging even custom silicon. In addition, the flexibility of the general purpose DSP chips enables us to add new primitives and even procedural shading techniques to achieve rich, complex, and realistic ray-traced scenes at speeds previously unattainable. We believe that TigerSHARK will set a new price/performance benchmark for graphics systems, and will push ray-tracing towards the realm of real-time realistic image synthesis.

Work is in progress on the construction of a 4-DSP TigerSHARK prototype to verify performance, feasibility, and scalability.

---

scheme in its hardware to avoid brute-forcing the object database.

[11] **A**pplication-**S**pecific **I**ntegrated **C**ircuit

A TigerSHARK Hardware Details

This appendix will describe in more detail the design of the 4-DSP prototype system.

A.1 Prototype Organization

The prototype system consists of four TMS320C32 DSPs, sharing a common system bus, connected to 128k words of 32-bit wide, zero wait-state memory. This common bus also interfaces with the AMCC S5933 PCI controller chip to provide simple interconnect with the host processor bus. A small PLD is required to implement some custom logic functions to tie the AMCC "Matchmaker" chip to the DSP bus.

A.1.1 PCI interface

A "PCI developer's kit," manufactured by AMCC and sold by Vector Electronics, is used to simplify the prototype's PCI interface. This kit has the AMCC S5933 PCI controller chip and a minimal interface pre-built, with headers to connect external devices to the S5933. This simplifies the PCI interface immensely, since the PCI specification[21] contains highly detailed notes on bus loading, signal path length, connector construction and other details of the interface that the pre-built board already satisfies. It also allows us to concentrate on /tigershark/ without having to worry as much about the PCI interface.

The S5933 chip contains an extensive errata list, however,[12] which limits straight-forward use of the device. The S5933 provides a FIFO interface to the PCI bus which would be ideal for our purposes, except for the fact that the interface as described in the data sheets [1] does not correspond to the interface implemented in silicon. For that reason, we must use a PLD to implement some custom logic which allows us to use the "Add-on Bus Interface," which, though still not bug-free, will actually do most of the things indicated in the data book.

A.1.2 Memory interface

A major limitation on the speed of the TigerSHARK processing elements is memory access time. Each primitive to be processed may consist of as many as 50 words of data, which must be fetched from the model database during

[12] The AMCC errata list is outrageous enough to have sparked some tentative attempts at humor; see for example `http://humper.student.princeton.edu:80/ humper/498/poem.html`.

the ray-primitive intersection routine. Fortunately, the TMS320C32 allows us to simulatanously fetch and process data due to its pipelined, multiple bus architecture. This helps with the memory bandwidth somewhat, but it is still desirable to implement a high speed primitive store. Thus, the RAM bank used operates with zero wait-states to avoid stalling the DSPs if possible.

No matter how large a model database store we implement on-board, film-quality animation will eventually exhaust and over run it. *Toy Story*, as a typical example, possessed a model database of tens of millions of primitives during certain scenes. Obviously, the information in the on-board primitive store will need to be 'paged' out at intervals to allow the DSPs to intersect rays with the entire model database.

Obviously, an ideal situation would double-buffer the model database, allowing the host processor to update the database without halting the DSP, and 'instantaneously' switch to the new database when the update is complete. However, the extra memory required for this scheme is very expensive. Typically, memory composes over 60% of the cost of a workstation [20]; TIGERSHARKś low-cost DSP exaggerate that percentage still more. Also, the extra hardware required to switch the system bus between two memory banks greatly increases our chip count and system complexity.

Single-buffering the model database will clearly incur an idle-time penalty. For a certain amount of time, the DSPs will need to be halted, and thus not doing useful work, while the host processor updates the model database. The performance hit caused by this idle time was calculated for typical scenes, and it was found that for a 512 DSP system requiring a refresh every 100,000 rays and transfering data at the maximum practical PCI bus speed,[13] the refresh would cause the processors to idle for less than 1% of their running time. This is a very reasonable trade-off, considering the savings in cost and complexity. The database memory was sized based on the above calculations and bandwidth considerations.

A.1.3 Bus Loading

Since TIGERSHARKś bus requires synchronized memory accesses, writes to memory must be strictly controlled to avoid line driver conflicts. Therefore, only one of the TMS320C32 processors on board has been given 'write permission.'

[13] Although the PCI bus has a 'never to exceed' bandwidth of 132 MB/s, typical figures cited in the literature [8] indicate a typical maximum bandwidth of 80 MB/s.

In addition, the capacitive loads placed on the bus by the parallel processors necessitate the use of buffers in the system bus[14]. Accordingly, dual high-speed 16-bit uni-directional buffers have been placed in the shared system bus between the 'master' DSP and the chain of slaves.

A.1.4 Synchronization

Tight clock synchronization is necessary in the system. A low-skew National Semiconductor CSG74B2528 clock driver chip is used to distribute a single system clock to all the DSPs in the system with less than 450 ps skew.

A low-complexity coarse synchronization primitive is also provided for system software via the !HOLD, !HOLDA, !XF0 and !XF1 signals of the TMS320C32.

A.2 Software Concepts

A number of interesting low-level software issues are related to the hardware design.

A.2.1 Node identification

If all of the DSPs execute the same code and access memory simultaneously, then one might wonder how we can tell which is which. The low-speed serial bus is an excellent means to do that: since the devices are daisy-chained together, it is no problem to place a token on the line at the host side which can be passed progressively down the chain to uniquely identify each processor. This also informs the 'master' processor that he (alone) can write the memory.

A.2.2 Bounding Box Optimizations

A very common ray-tracing optimization is the bounding box, briefly mentioned in sections 4.1 and 3.2. This is a method to avoid unnecessary ray-primitive intersections by constructing a series of enclosing primitives. If a ray does not intersect with the enclosing primitive, it is not necessary to test it against all the primitives inside the enclosure. Automated methods to construct these bounding hierarchies have been proposed [11].

The application of this optimization to the TIGERSHARK architecture was discussed in section 5.1. This optimization maps well onto the existing hard-

[14] In a four-processor system this is not strictly necessary, but it was considered good design practice to include it, as it would be required for larger systems.

ware, as well: when intersecting rays with a bounding box, each processor that computes a ray intersection pulls a shared, wire-OR, line low. Processors that compute no intersection flip the data direction register to 'input', and read the line status. If no processors find an intersection, then all processors will read the line pulled 'high' by a pull-up resistor and know that they can skip the data enclosed in the bounding box. If any processor finds an intersection, it pulls the shared line low, and the remainder of the processors will read the 'low' status and know that the bounding box can not be skipped. The non-reading processor knows already that his vote will ensure the the bounding box is not skipped. Thus will one read/write of a status line, and hardly any external hardware, we can implement the group-voting procedure necessary for the bounding box optimization.

B TigerSHARK software and simulation

Presented in this appendix is code for a software based simulator of our vectorized raytracing algorithm. This code maps quite directly onto the hardware we have proposed. The only modifications for the final version would be that the triangle intersection routine would be compiled to DSP assembly code, and the ray-queue operations would be handled in the device driver for the TIGERSHARK board. Otherwise, this is a simple raytracer that implements our algorithm.

B.1 shark.h

```
#ifndef SHARK_H
#define SHARK_H

#include "hvector.h"  /* Humper's own vector routines */
#include "humper.h"   /* Humper's random utilities */
#include "himage.h"   /* For the framebuffer */

typedef float Flt;

/************ Triangle Stuff ********/

typedef struct {
  Vec color;
  Flt kdiff;
  Flt kspec;
  Flt kamb;
  Flt ns;
  Flt ktrans;
  Flt reflect;
  Flt refindex;
} Material;

typedef struct {
  Vec v1, v2, v3;  /* Vertex array */
  Vec n1, n2, n3;  /* Normal array */
```

```
  int simple:1;    /* Are all the normals the same? */
  Flt plane[4];  /* Equation of plane defined by triangle */
  Material *m;     /* What's the material there? */
} Triangle;

typedef struct {
  Triangle *o;     /* Which object did we hit */
  Vec P;           /* Where did we hit? */
  Flt d;           /* How far along the ray? */
} Isect;

typedef struct {
  Vec Color;
  Vec Pos;
} Light;

/************ Ray stuff **********/

typedef struct Ray {
  Vec P;             /* Origin of ray */
  Vec D;             /* Direction of ray -- MUST BE NORMALIZED */
  struct Ray *parent;/* Our parent ray -- NULL if we're an eye ray */
  enum { COLOR, REFLECT, REFRACT, SHADOW } type; /* What kind of ray are we? */
  Vec Color;         /* The color associated with this ray */
  Flt x,y;           /* Screen space coordinates for this ray -- only if eye */
  Isect i;           /* best intersection seen so far */
  int depth;         /* Recursion depth */
  Flt dist;          /* How far is the light? */
  int light;         /* Which light is it, anyhow? */
} Ray;

/* Note -- we don't maintain child links because we *always* perform
   the child operations in the same order: shadow, reflect, and
   refract.  If there is no refraction (or no refraction and no
   reflection), the refraction routine will notice that and simply
   invoke the "done" method of the ray "class".  Otherwise, the
   refraction routine will simply invoke its parent's "done" when it has
```

```
   finished.  When done is invoked, either because all the children of that
   ray are finished, or we have hit the bottom of the ray-tree, or the ray
   has not hit an object, we invoke our parent's "done" method, until we get
   to the eye ray, whose done method will draw the color on the screen.
   Pretty straightforward. */

extern void RayDone(Ray *);

typedef struct {
  Vec Pos;     /* Position in 3-space */
  Vec Lookat;  /* Point we're looking at */
  Vec Up;      /* Up vector */
  Vec Direction;
  Vec Right;
} Camera;

extern humperImage *fb;  /* Our framebuffer */

/***** QUEUE OPERATIONS *******/

typedef struct Raynode {
  Ray *ray;
  struct Raynode *next;
} Raynode;

typedef struct {
  Raynode *head;
  Raynode *tail;
  int depth;
} Rayqueue;

extern Rayqueue rayQ;
extern void Raypush(Ray *);
extern Ray *Raypop(void);
extern int RayQdepth(void);

/********* Scene Stuff *********/
```

```
typedef struct {
  int width, height; /* Size of output image */
  Triangle *objects; /* Array of triangles */
  int num_objects;   /* Number of triangles in aforementioned array */
  Light *lights;     /* Array of lights */
  int num_lights;    /* Number of lights in aforementioned array */
  Camera theCamera;  /* The camera.  Duh. */
  Vec background;    /* Background color of the scene. */
} Scene;

extern Scene theScene;

extern void TransmissionDirection(Material *, Vec, Vec, Vec);
extern void SpecularDirection(Vec, Vec, Vec);
extern void TriIntersect(Ray *, Triangle *);
extern void SceneRead(char *);
extern void SceneIntersect(Ray *);
extern void ComputeEyeRays(void);
extern void ProcessRay(void);
extern void RayIsect(Ray *, Flt, Triangle *);
extern void Shade(Ray *);
extern void Contribute(Ray *);
extern void cross_product(Vec, Vec, Vec, Vec);
extern void boring_cross_product(Vec, Vec, Vec, Vec);

/********* Timer Stuff *********/

extern void ClearTimer(int timer);
extern void ClearAllTimers(void);
extern void StartTimer(int timer);
extern void StopTimer(int timer);
extern double ReadTimer(int timer);
extern void PrintTimers();

#define TOTAL_TIMER             0
#define NEWRAY_TIMER            1
```

```
#define SHADE_TIMER             2
#define INTERSECT_TIMER         3
#define DISK_TIMER              4

#define NUM_TIMERS 5



#endif /* SHARK_H */
```

B.2 shark.c

```
#include "shark.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static char out_file[100];

static void Usage(char *progname)
{
  humper_warning(3,"In Usage!");
  humper_error("Usage: %s FILENAME.shk\n",progname);
}

static void PrintBanner(void)
{
  humper_warning(3,"In PrintBanner!");
  printf ("+------------------------------------------------------------+\n");
  printf ("|                                                            |\n");
  printf ("|                          TigerSHARK                        |\n");
  printf ("|     Scalable Hardware Accelerated Ray-tracing Kernel       |\n");
  printf ("|                                                            |\n");
  printf ("|    Greg Humphreys, BSE Princeton Computer Science '97      |\n");
  printf ("|C. Scott Ananian, BSE Princeton Electrical Engineering '97  |\n");
  printf ("|         Doug Clark and Andrew Wolfe, advisers              |\n");
  printf ("|                                                            |\n");
```

```
  printf ("+------------------------------------------------------------+\n");
}

static void SHARKInit(int argc, char *argv[])
{
  humper_warning(3,"In SHARKInit!");
  PrintBanner();
  if (argc != 2)
    Usage(argv[0]);     /* Will never return */
  SceneRead(argv[1]);   /* Read the scene description from disk. */
  strcpy(out_file,argv[1]);
  strcpy(out_file+strlen(out_file)-4,".ppm");
}

static void SHARKShutDown(void)
{
  static int done_this = 0;
  humper_warning(3,"In SHARKShutDown!");
  if (!done_this)
    {
      done_this = 1;
      StartTimer(DISK_TIMER);
      humper_writeImage(out_file,fb);
      StopTimer(DISK_TIMER);
    }
  PrintTimers();
}

void main(int argc, char *argv[])
{
  humper_warning(3,"In Main!");
  StartTimer(TOTAL_TIMER);
  atexit(SHARKShutDown);
  SHARKInit(argc, argv);
  ComputeEyeRays();
  while(RayQdepth())
    {
```

```
      ProcessRay(); /* Should be done by the driver -- maybe this would just
                       be a busy loop? Maybe we should show results as they
                       come in (although they may come in in a totally random
                       order -- would be kind of neat to see */
    }
}
```

B.3 ray.c

```
#include "shark.h"
#include "humper.h"
#include "himage.h"

static void ComputeEyeRay(Flt i, Flt j)
{
  Ray *ray;
  double X_Scalar, Y_Scalar;
  Vec temp1, temp2;

  StartTimer(NEWRAY_TIMER);
  ray = humper_malloc(sizeof(Ray));

  X_Scalar = (i - (Flt)theScene.width/2.0)/(Flt)theScene.width;
  Y_Scalar = (j - (Flt)theScene.height/2.0)/(Flt)theScene.height;

  VecScale(Y_Scalar,theScene.theCamera.Up,temp1);
  VecScale(X_Scalar,theScene.theCamera.Right,temp2);
  VecAdd(temp1,temp2,ray->D);
  VecAdd(ray->D,theScene.theCamera.Direction,ray->D);
  VecUnit(ray->D,ray->D);
  VecCopy(theScene.theCamera.Pos,ray->P);
  ray->type = COLOR; /* We're certainly not a shadow ray! */
  ray->parent = NULL; /* We're an eye ray! */
  ray->x = i;
  ray->y = j;
  ray->depth = 1;
  Raypush(ray);
```

```
  StopTimer(NEWRAY_TIMER);
}


void ComputeEyeRays(void)
{
  int i,j;
  humper_warning(3,"In ComputeEyeRays!");
  for (i=0;i<theScene.width;i++)
    {
      for (j=0;j<theScene.height;j++)
        {
          ComputeEyeRay((Flt)i + .5, (Flt)j + .5);
        }
    }
}

void ProcessRay(void)
{
  Ray *ray;
  humper_warning(3,"In ProcessRay!");
  ray = Raypop();
  humper_warning(4,"Processing ray:");
  humper_warning(4,"\tPos: (%f,%f,%f)",ray->P[0],ray->P[1],ray->P[2]);
  humper_warning(4,"\tDir: (%f,%f,%f)",ray->D[0],ray->D[1],ray->D[2]);
  SceneIntersect(ray); /* this routine should call the RayDone() function! */
}

void RayDone(Ray *ray)
{
  humper_warning(3,"In RayDone!");

  if (ray->parent == NULL) /* Eye ray!  Write the color to the framebuffer! */
    {
      /* No oversampling -- should do something clever here */
      if ((int)(ray->x) == 100 && (int)(ray->y) == 100)
        {
```

```
          printf ("Hi!\n");
        }
      if (ray->Color[0] > 1) ray->Color[0] = 1;
      if (ray->Color[1] > 1) ray->Color[1] = 1;
      if (ray->Color[2] > 1) ray->Color[2] = 1;
      if (ray->Color[0] < 0) ray->Color[0] = 0;
      if (ray->Color[1] < 0) ray->Color[1] = 0;
      if (ray->Color[2] < 0) ray->Color[2] = 0;
      RED(fb,(int)(ray->x),(int)(ray->y)) = ray->Color[0];
      GREEN(fb,(int)(ray->x),(int)(ray->y)) = ray->Color[1];
      BLUE(fb,(int)(ray->x),(int)(ray->y)) = ray->Color[2];
    }
   else if (ray->type == REFLECT)
     {
       VecAddS(ray->parent->i.o->m->reflect,
               ray->Color,
               ray->parent->Color,
               ray->parent->Color);
     }
  else if (ray->type == SHADOW)
    {
      VecAdd(ray->Color,ray->parent->Color,ray->parent->Color);
      RayDone(ray->parent);
    }
  else if (ray->type == REFRACT)
    {
      VecAddS(ray->parent->i.o->m->ktrans,
              ray->Color,
              ray->parent->Color,
              ray->parent->Color);
      RayDone(ray->parent);
    }
  humper_free(ray);
}

void RayIsect(Ray *ray, Flt t, Triangle *tr)
{
```

```
  if (t < ray->i.d)
    {
      ray->i.d = t;
      VecAddS(t,ray->D,ray->P,ray->i.P);
      ray->i.o = tr;
    }
}
```

B.4 scene.c

```
#include <stdio.h>
#include <limits.h>
#include <ctype.h>
#include "shark.h"
#include "humper.h"
#include "himage.h"

Material *m = NULL;

typedef struct Tnode {
  Triangle *t;
  struct Tnode *next;
} Tnode;

Tnode *triangles = NULL;

typedef struct Lnode {
  Light *l;
  struct Lnode *next;
} Lnode;

Lnode *lights = NULL;

static void DefaultScene(void)
{
  theScene.width = 640;
  theScene.height = 480;
```

```
  theScene.objects = NULL;
  theScene.num_objects = 0;
  theScene.lights = NULL;
  theScene.num_lights = 0;
  VecSet(theScene.theCamera.Pos,1,0,0);
  VecSet(theScene.theCamera.Lookat,0,0,0);
  VecSet(theScene.theCamera.Up,0,1,0);
  VecSet(theScene.theCamera.Right,0,0,1);
  VecSet(theScene.theCamera.Direction,-1,0,0);
  VecSet(theScene.background,0,.5,1); /* Some color */
  m = humper_malloc(sizeof(Material));
  VecSet(m->color,1,1,1); /* White */
  m->kdiff = .8;
  m->kspec = .2;
  m->kamb = .2;
  m->ns = 5;
  m->ktrans = 0;
  m->reflect = 0;
  m->refindex = 1;
}

static void PrintSceneStats(void)
{
  int i;
  humper_warning(3,"In PrintSceneStats!");
  humper_warning(4,"Scene width: %d",theScene.width);
  humper_warning(4,"Scene height: %d",theScene.height);
  humper_warning(4,"Scene background color: (%f,%f,%f)",
                 theScene.background[0],
                 theScene.background[1],
                 theScene.background[2]);
  humper_warning(4,"Number of objects in scene: %d",theScene.num_objects);
  humper_warning(4,"Camera position: %f %f %f",
                 theScene.theCamera.Pos[0],
                 theScene.theCamera.Pos[1],
                 theScene.theCamera.Pos[2]);
  humper_warning(4,"Camera lookat: %f %f %f",
```

```
                theScene.theCamera.Lookat[0],
                theScene.theCamera.Lookat[1],
                theScene.theCamera.Lookat[2]);
  humper_warning(4,"Camera up: %f %f %f",
                theScene.theCamera.Up[0],
                theScene.theCamera.Up[1],
                theScene.theCamera.Up[2]);

  for (i=0;i<theScene.num_objects;i++)
    {
      humper_warning(4,"Triangle %d:",i);
      humper_warning(4,"\t(%.2f,%.2f,%.2f),(%.2f,%.2f,%.2f),(%.2f,%.2f,%.2f)",
                    theScene.objects[i].v1[0],
                    theScene.objects[i].v1[1],
                    theScene.objects[i].v1[2],
                    theScene.objects[i].v2[0],
                    theScene.objects[i].v2[1],
                    theScene.objects[i].v2[2],
                    theScene.objects[i].v3[0],
                    theScene.objects[i].v3[1],
                    theScene.objects[i].v3[2]);
      if (theScene.objects[i].simple)
        {
          humper_warning(4,"\tSimple triangle.");
          humper_warning(4,"\tNormal: (%.2f,%.2f,%.2f)",
                        theScene.objects[i].n1[0],
                        theScene.objects[i].n1[1],
                        theScene.objects[i].n1[2]);
        }
      else
        {
          humper_warning(4,"\tComplex triangle.");
          humper_warning(4,"\tNormals: (%.2f,%.2f,%.2f),(%.2f,%.2f,%.2f),(%.2f,%.2f,%.2f)",
                        theScene.objects[i].n1[0],
                        theScene.objects[i].n1[1],
                        theScene.objects[i].n1[2],
                        theScene.objects[i].n2[0],
```

```
                        theScene.objects[i].n2[1],
                        theScene.objects[i].n2[2],
                        theScene.objects[i].n3[0],
                        theScene.objects[i].n3[1],
                        theScene.objects[i].n3[2]);
        }
      humper_warning(4,"\tPlane equation: (%.2f,%.2f,%.2f,%.2f)",
                     theScene.objects[i].plane[0],
                     theScene.objects[i].plane[1],
                     theScene.objects[i].plane[2],
                     theScene.objects[i].plane[3]);
    }
}

void SceneRead(char *filename)
{
  FILE *fp;
  char buf[500];
  Tnode *new;
  Lnode *newl;
  int i;

  humper_warning (3,"In SceneRead!");
  if ((fp = fopen(filename,"r")) == NULL)
    humper_error("Can't open %s for reading!",filename);

  DefaultScene();

  StartTimer(DISK_TIMER);
  while(fgets(buf,500,fp))
    {
      switch(tolower(buf[0])) /* Not all that robust, but who cares. */
        {
        case 'm': /* Material */
          m = humper_malloc(sizeof(Material));
          sscanf(buf,"%*s %f %f %f %f %f %f %f %f %f %f",
                 &(m->color[0]),
```

```
            &(m->color[1]),
            &(m->color[2]),
            &(m->kdiff),
            &(m->kspec),
            &(m->kamb),
            &(m->ns),
            &(m->ktrans),
            &(m->reflect),
            &(m->refindex));
    break;
  case 'b': /* background */
    sscanf(buf, "%*s %f %f %f",
            &(theScene.background[0]),
            &(theScene.background[1]),
            &(theScene.background[2]));
    theScene.theCamera.Pos[3] = 1;
    break;
  case 'e': /* Eyepoint */
    sscanf(buf, "%*s %f %f %f",
            &(theScene.theCamera.Pos[0]),
            &(theScene.theCamera.Pos[1]),
            &(theScene.theCamera.Pos[2]));
    theScene.theCamera.Pos[3] = 1;
    break;
  case 'l': /* Lookat or light*/
    switch(tolower(buf[1]))
      {
      case 'o': /* Lookat */
        sscanf(buf, "%*s %f %f %f",
                &(theScene.theCamera.Lookat[0]),
                &(theScene.theCamera.Lookat[1]),
                &(theScene.theCamera.Lookat[2]));
        theScene.theCamera.Lookat[3] = 1;
        break;
      case 'i': /* Light */
        theScene.num_lights++;
        newl = humper_malloc(sizeof(Lnode));
```

```
        newl->next = lights;
        lights = newl;
        newl->l = humper_malloc(sizeof(Light));
        sscanf(buf,"%*s %f %f %f %f %f %f",
               &(newl->l->Pos[0]),&(newl->l->Pos[1]),&(newl->l->Pos[2]),
               &(newl->l->Color[0]),
               &(newl->l->Color[1]),
               &(newl->l->Color[2]));
        break;
      }
    break;
  case 'u': /* Eyepoint */
    sscanf(buf, "%*s %f %f %f",
           &(theScene.theCamera.Up[0]),
           &(theScene.theCamera.Up[1]),
           &(theScene.theCamera.Up[2]));
    theScene.theCamera.Up[3] = 1;
    break;
  case 't': /* Simple triangle */
    theScene.num_objects++;
    new = humper_malloc(sizeof(Tnode));
    new->next = triangles;
    triangles = new;
    new->t = humper_malloc(sizeof(Triangle));
    new->t->simple = 1;
    sscanf(buf,"%*s %f %f %f %f %f %f %f %f %f",
           &(new->t->v1[0]),&(new->t->v1[1]),&(new->t->v1[2]),
           &(new->t->v2[0]),&(new->t->v2[1]),&(new->t->v2[2]),
           &(new->t->v3[0]),&(new->t->v3[1]),&(new->t->v3[2]));

    cross_product(new->t->v1,new->t->v2,new->t->v3,new->t->n2);
    cross_product(new->t->v2,new->t->v3,new->t->v1,new->t->n3);
    cross_product(new->t->v3,new->t->v1,new->t->v2,new->t->n1);

    VecCopy(new->t->n1,new->t->n2);
    VecCopy(new->t->n1,new->t->n3);
    new->t->m = m; /* Assumes a default material */
```

```
  VecCopy(new->t->n1,new->t->plane);
  new->t->plane[3] = -(new->t->plane[0]*new->t->v1[0] +
                       new->t->plane[1]*new->t->v1[1] +
                       new->t->plane[2]*new->t->v1[2]);
  break;
case 'n': /* Normaled triangle */
  theScene.num_objects++;
  new = humper_malloc(sizeof(Tnode));
  new->next = triangles;
  triangles = new;
  new->t = humper_malloc(sizeof(Triangle));
  new->t->simple = 0;
  sscanf(buf,"%*s %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f",
         &(new->t->v1[0]),&(new->t->v1[1]),&(new->t->v1[2]),
         &(new->t->v2[0]),&(new->t->v2[1]),&(new->t->v2[2]),
         &(new->t->v3[0]),&(new->t->v3[1]),&(new->t->v3[2]),
         &(new->t->n1[0]),&(new->t->n1[1]),&(new->t->n1[2]),
         &(new->t->n2[0]),&(new->t->n2[1]),&(new->t->n2[2]),
         &(new->t->n3[0]),&(new->t->n3[1]),&(new->t->n3[2]));
  VecUnit(new->t->n1,new->t->n1);
  VecUnit(new->t->n2,new->t->n2);
  VecUnit(new->t->n3,new->t->n3);
  new->t->m = m; /* Assumes a default material */

  cross_product(new->t->v3,new->t->v1,new->t->v2,new->t->plane);
  new->t->plane[3] = -(new->t->plane[0]*new->t->v1[0] +
                       new->t->plane[1]*new->t->v1[1] +
                       new->t->plane[2]*new->t->v1[2]);
  break;
case 'w': /* Width */
  sscanf(buf,"%*s %d",&(theScene.width));
  break;
case 'h': /* Height */
  sscanf(buf,"%*s %d",&(theScene.height));
  break;
case '\n': /* Blank line */
case '#':  /* Comment */
```

```
          continue;
          break;
        default:
          humper_error("Duh, what's this: %s",buf);
          break;
        }
    }
  theScene.objects = humper_malloc(theScene.num_objects*sizeof(Triangle));
  for (new = triangles,i=0;i<theScene.num_objects;i++,new=new->next)
    theScene.objects[i] = *(new->t);
  theScene.lights = humper_malloc(theScene.num_lights*sizeof(Light));
  for (newl = lights,i=0;i<theScene.num_lights;i++,newl=newl->next)
    theScene.lights[i] = *(newl->l);

  /* Huge fucking memory leak!  Fix this! */

  VecSub(theScene.theCamera.Lookat,
         theScene.theCamera.Pos,
         theScene.theCamera.Direction);
  VecUnit(theScene.theCamera.Direction,theScene.theCamera.Direction);
  VecCross(theScene.theCamera.Direction,
           theScene.theCamera.Up,
           theScene.theCamera.Right);
  VecUnit(theScene.theCamera.Right,theScene.theCamera.Right);
  VecCross(theScene.theCamera.Right,
           theScene.theCamera.Direction,
           theScene.theCamera.Up);
  fb = humper_newImage(theScene.height,theScene.width);
  StopTimer(DISK_TIMER);
  PrintSceneStats();
}

void SceneIntersect(Ray *ray)
{
  int i;

  humper_warning(3,"In SceneIntersect!");
```

```
  StartTimer(INTERSECT_TIMER);

  ray->i.d = FLT_MAX;

  for (i=0;i<theScene.num_objects;i++)
    TriIntersect(ray,&(theScene.objects[i]));

  StopTimer(INTERSECT_TIMER);

  if (ray->type != SHADOW)
    {
      if (ray->i.d == FLT_MAX)
        {
          VecCopy(theScene.background,ray->Color);
          RayDone(ray);
        }
      else
        Shade(ray);
    }
  else
    {
      if (ray->i.d >= ray->dist) /* Yay!   Light contribution! */
        Contribute(ray);
    }
}
```

B.5 tri.c

```
#include "shark.h"
#include "humper.h"
#include <math.h>

void cross_product(Vec a, Vec b, Vec c, Vec normal)
{
  boring_cross_product(a,b,c,normal);
  VecUnit(normal,normal);
```

```
}

void boring_cross_product(Vec a, Vec b, Vec c, Vec normal)
{
  Vec t1, t2;
  t1[0] = b[0] - a[0];
  t1[1] = b[1] - a[1];
  t1[2] = b[2] - a[2];

  t2[0] = c[0] - b[0];
  t2[1] = c[1] - b[1];
  t2[2] = c[2] - b[2];

  normal[0] = t1[1]*t2[2] - t2[1]*t1[2];
  normal[1] = t1[2]*t2[0] - t2[2]*t1[0];
  normal[2] = t1[0]*t2[1] - t2[0]*t1[1];
}

#define THRESH 1e-2
#define THRESH3 1e-4

void TriIntersect(Ray *ray, Triangle *tr)
{
  Vec N, I;
  Flt vd, v0, t, rayeps;
  Vec NN1, NN2, NN3, N1, N2, N3;

  humper_warning(3,"In TriIntersect!");

  rayeps = 1E-1;

  N[0] = tr->plane[0];
  N[1] = tr->plane[1];
  N[2] = tr->plane[2];

  vd = VecDot(N,ray->D);
  if (!vd)
```

```
  return;

v0 = -(VecDot(ray->P,N) + tr->plane[3]);
t = v0/vd;

if (t <= rayeps)
  return;

VecAddS(t,ray->D,ray->P,I);

boring_cross_product(I,tr->v1,tr->v2,N1);
boring_cross_product(I,tr->v2,tr->v3,N2);
boring_cross_product(I,tr->v3,tr->v1,N3);

if (VecLen(N1) < THRESH3 || VecLen(N2) < THRESH3 || VecLen(N3) < THRESH3)
  {
    RayIsect(ray,t,tr);
    return;
  }

VecUnit(N1, NN1);
VecUnit(N2, NN2);
VecUnit(N3, NN3);

if((VecLen(N1)<THRESH3 && VecEqual(NN3,NN2,THRESH))
   || (VecLen(N2)<THRESH3 && VecEqual(NN3,NN1,THRESH))
   || (VecLen(N3)<THRESH3 && VecEqual(NN1,NN2,THRESH)))
  {
    RayIsect(ray,t,tr);
    return;
  }
if (VecEqual(NN1,NN2,rayeps) && VecEqual(NN2,NN3,rayeps))
  {
    RayIsect(ray,t,tr);
    return;
  }
return;
```

```
}
```

B.6 shade.c

B.7 rayq.c

```
#include "shark.h"
#include "humper.h"

void Raypush(Ray *ray)
{
  Raynode *new = humper_malloc(sizeof(Raynode));
  humper_warning(3,"In Raypush");
  new->ray = ray;
  new->next = NULL;
  if (rayQ.tail)
    {
      rayQ.tail->next = new;
      rayQ.tail = new;
    }
  else
    rayQ.head = rayQ.tail = new;
  rayQ.depth++;
}

Ray *Raypop(void)
{
  Ray *ret;
  Raynode *temp;
  humper_warning(3,"In Raypop");
  if (rayQ.depth == 0)
    humper_error("Ray queue underflow");
  ret = rayQ.head->ray;
  temp = rayQ.head;
   rayQ.head = rayQ.head->next;
  if (!rayQ.head) rayQ.tail = NULL;
```

```
  rayQ.depth--;
  humper_free(temp);
  return ret;
}

int RayQdepth(void)
{
  humper_warning(3,"In RayQdepth");
  return rayQ.depth; /* Suitable for use in a while loop :) */
}
```

B.8 humper.c

B.9 sharkglobals.c

```
#include "shark.h"
#include "himage.h"

Rayqueue rayQ = {NULL, NULL, 0};
Scene theScene;
humperImage *fb = NULL;
```

B.10 himage.c

```
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include "humper.h"
#include "himage.h"

humperImage *humper_image = NULL;

void humper_freeImage(humperImage *hi)
{
  humper_free(hi->data);
```

```
}

/****
 *
 * humper_setImage -- set the current image in memory
 *
 * This routine takes an image and replaces the current image with it.
 * If the two are the same pointer, this routine does nothing.
 * This allows the user defined filters to simply return their argument
 * if they want to do nothing.
 *
 * This is useful if you want to create a filter that is simply a combination of
 * two other filters -- just use the preprocess and the postprocess filters,
 * and the filter routine simply returns its argument.
 *
 ****/

void humper_setImage(humperImage *hi)
{
  if (hi == humper_image) return;
  humper_warning(3,"Setting image");
  if (humper_image)
    {
      humper_freeImage(humper_image);
      humper_free(humper_image);
    }
  humper_image = hi;

}

humperImage *humper_readImage(char *filename)
{
  humperImage *new;
  FILE *fp;
  int i,j;
  char buf[80];
  int need_magic = 1;
```

```
int need_dims = 1;
int need_max = 1;
int width, height;
int max;

if ((fp = fopen(filename,"r")) == NULL)
  humper_error("Can't open image file %s.",filename);

humper_warning(3,"Reading image from %s",filename);

i = 0 ; j = 0;
while (!feof(fp))
  {
    if (need_magic || need_dims || need_max)
      {
        fgets(buf,80,fp);
        if (buf[0] == '#') continue;
        if (need_magic)
          {
            need_magic = 0;
            if (strcmp(buf,"P6\n"))
              humper_error("%s is not a ppm file!",filename);
            continue;
          }
        if (need_dims)
          {
            need_dims = 0;
            sscanf(buf,"%d %d",&width,&height);
            new = humper_newImage(height,width);
            continue;
          }
        if (need_max)
          {
            need_max = 0;
            sscanf(buf,"%d",&max);
            continue;
          }
```

```
        }
      RED(new,i,j) = UB_TO_F((unsigned char) fgetc(fp));
      GREEN(new,i,j) = UB_TO_F((unsigned char) fgetc(fp));
      BLUE(new,i,j) = UB_TO_F((unsigned char) fgetc(fp));
      i++;
      if (i == width) { i = 0 ; j++; }
      if (j == height) break;
    }
  return new;
}

humperImage *humper_makeTexture(humperImage *hi)
{

  int x,y,i,j;
  humperImage *new;
  /* Texture dims must be powers of two. */

  for (x=1;x<hi->cols;x*=2) ;
  for (y=1;y<hi->rows;y*=2) ;

  new = humper_newImage(y,x);

  new->s = (float)hi->cols/(float)x;
  new->t = (float)hi->rows/(float)y;

  for (i=0;i<x;i++)
    {
      for (j=0;j<y;j++)
        {
          if (i < hi->cols && j < hi->rows)
            {
              RED(new,i,j) = RED(hi,i,j);
              GREEN(new,i,j) = GREEN(hi,i,j);
              BLUE(new,i,j) = BLUE(hi,i,j);
            }
        }
```

```
    }
  return new;
}


humperImage *humper_readTexture(char *filename)
{
  humperImage *temp, *new;

  temp = humper_readImage(filename);
  new = humper_makeTexture(temp);

  humper_freeImage(temp);
  humper_free(temp);
  return new;
}


void humper_writeImage(char *filename, humperImage *hi)
{
  FILE *fp;
  int i,j;

  if ((fp = fopen(filename,"wb"))==NULL)
    humper_error("Can't open image file %s for writing in binary mode.",filename);

  humper_warning(3,"Writing image data to %s.",filename);

  fprintf(fp,"P6\n%d %d\n255\n",hi->cols,hi->rows);
  for (i=hi->rows-1;i>=0;i--)
    {
      for (j=0;j<hi->cols;j++)
        {
          fputc(F_TO_UB(RED(hi,j,i)),fp);
          fputc(F_TO_UB(GREEN(hi,j,i)),fp);
          fputc(F_TO_UB(BLUE(hi,j,i)),fp);
        }
```

```
    }
  fclose(fp);
}

humperImage *humper_copyImage(humperImage *src)
{
  humperImage *new = humper_newImage(src->rows,src->cols);
  humper_warning(3,"copying Image");
  memcpy(new->data,src->data,new->rows*new->cols*3*sizeof(float));
  return new;
}

humperImage *humper_newImage(int rows, int cols)
{
  humperImage *new = humper_malloc(sizeof(humperImage));
  if (!new)
    humper_error("Out of memory in humper_newImage!");
  humper_warning(3,"making a new image (%d x %d)",rows,cols);
  if (rows < 0)
    humper_error("Invalid rows argument to humper_newImage");
  if (cols < 0)
    humper_error("Invalid cols argument to humper_newImage");
  new->rows = rows;
  new->cols = cols;
  new->data = humper_malloc(rows*cols*3*sizeof(float));
  return new;
}

void clamp(float *r, float *g, float *b)
{

 humper_warning(4,"Clamping: %f %f %f",*r,*g,*b);

  if (*r > 255) *r = 255;
  else if (*r < 0) *r = 0;

  if (*g > 255) *g = 255;
```

```
  else if (*g < 0) *g = 0;

  if (*b > 255) *b = 255;
  else if (*b < 0) *b = 0;
}

humperImage *humper_convolveImage(humperImage *hi, void *data)
{
  convolveStruct *c = (convolveStruct *) data;
  humperImage *temp = humper_newImage(hi->rows + 2*(c->size/2),hi->cols + 2*(c->size/2));
  humperImage *new = humper_newImage(hi->rows,hi->cols);
  int i,j,x,y;
  float r,g,b;
  int offset = c->size / 2;
  int offset2 = temp->cols * 3;
  int t1 = 3-c->size / 2;
  int t2 = 3+c->size / 2;
  int rows = new->rows;
  int cols = new->cols;
  int xcoord, ycoord;
  float *buf;

  humper_warning(3,"In the convolve kernel");
  humper_warning(3,"convolution size: %d",c->size);
  humper_warning(3,"convolution wrap: %s", c->wrap ? "TRUE" : "FALSE");
  humper_warning(3,"convolution bias: %d", c->bias);
  humper_warning(3,"kernel:");

#ifdef DEBUG_LEVEL
  if (DEBUG_LEVEL >= 3)
    {
      for (i = 3 - c->size/2 ; i <= 3 + c->size/2 ; i++)
        {
          for (j = 3 - c->size/2 ; j <= 3 + c->size/2 ; j++)
            {
              humper_warning(3, "c->data[%d][%d] = %f", i,j,c->data[i][j]);
            }
```

```
        }
    }
#endif

  /* Preprocess temp -- mirror image at borders */
  for (y = 0 ; y < temp->rows; y++)
    {
      for (x = 0 ; x < temp->cols ; x++)
        {
          if (x < c->size/2) xcoord = c->size - x;
          else if (x > temp->cols - c->size/2-1) xcoord = 2*(temp->cols-(c->size/2)-1) - x;
          else xcoord = x - c->size/2;

          if (y < c->size/2) ycoord = c->size - y;
          else if (y > temp->rows - c->size/2-1) ycoord = 2*(temp->rows-(c->size/2)-1) - y;
          else ycoord = y - c->size/2;

          RED(temp,x,y) = RED(hi,xcoord,ycoord);
          GREEN(temp,x,y) = GREEN(hi,xcoord,ycoord);
          BLUE(temp,x,y) = BLUE(hi,xcoord,ycoord);
        }
    }
  for (y=0;y<rows;y++)
    {
      for (x = 0 ; x < cols ; x++)
        {
          r = g = b = 0;
          for (i = t1 ; i <= t2 ; i++)
            {
              ycoord = (offset)+y+(i-3);
              for (j = t1 ; j <= t2 ; j++)
                {
                  xcoord = (offset)+x+(j-3);
                  buf = temp->data + ycoord*offset2 + xcoord*3;
                  r += c->data[i][j]*(*buf);
                  g += c->data[i][j]*(*(buf + 1));
                  b += c->data[i][j]*(*(buf + 2));
```

```
                }
            }
          if (c->wrap)
            {
              if (r < 0) r = -r;
              if (g < 0) g = -g;
              if (b < 0) b = -b;
            }
          if (c->bias)
            {
              r += c->bias;
              g += c->bias;
              b += c->bias;
            }
          clamp(&r,&g,&b);
          humper_warning(4,"Got: %d %d %d\n",r,g,b);
          RED(new,x,y) = r;
          GREEN(new,x,y) = g;
          BLUE(new,x,y) = b;
        }
    }
  humper_freeImage(temp);
  humper_free(temp);
  return new;
}


#define BOGUS -1

void humper_rgb2hsv(float r, float g, float b, float *h, float *s, float *v)
{
  float max,min;
  float delta;

  max = MAX(MAX(r,g),MAX(g,b));
  min = MIN(MIN(r,g),MIN(g,b));
```

```
  *v = max;
  if (max) *s = (max-min)/max;
  else *s = 0;
  if (*s == 0)
    *h = BOGUS;
  else
    {
      delta = max-min;
      if (r == max) *h = (g-b)/delta;
      else if (g == max) *h = 2 + (b-r)/delta;
      else *h = 4 + (r-g)/delta;
      *h *= 60;
      if (*h < 0) *h += 360;
    }
}

void humper_hsv2rgb(float h, float s, float v, float *r, float *g, float *b)
{
  int i;
  float f, p, q, t;
  if (s == 0 || h == BOGUS)
    {
      *r = v;
      *g = v;
      *b = v;
    }
  else
    {
      if (h == 360) h = 0;
      h /= 60.0;
      i = (int)h;
      f = h-i;
      p = v*(1-s);
      q = v*(1 - (s*f));
      t = v*(1-(s*(1-f)));
      switch(i)
        {
```

```
        case 0:
          *r = v ; *g = t ; *b = p ; break;
        case 1:
          *r = q ; *g = v ; *b = p ; break;
        case 2:
          *r = p ; *g = v ; *b = t ; break;
        case 3:
          *r = p ; *g = q ; *b = v ; break;
        case 4:
          *r = t ; *g = p ; *b = v ; break;
        case 5:
          *r = v ; *g = p ; *b = q ; break;
        default:
          *r = 0 ; *g = 0 ; *b = 0 ; break;
        }
    }
}
```

B.11 timers.c

```
#include <sys/time.h>
#include <stdio.h>
#include "shark.h"

static double start[NUM_TIMERS];
static double elapsed[NUM_TIMERS];

void ClearTimer(int timer)
{
  elapsed[timer] = 0.0;
}

void ClearAllTimers(void)
{
  int i;
  for (i=0; i<NUM_TIMERS; i++)
    ClearTimer(i);
```

```
}

void StartTimer(int timer)
{
  struct timeval t;

  gettimeofday(&t, NULL);
  start[timer] = (double) t.tv_sec + (double) t.tv_usec / 1000000.0;
}

void StopTimer(int timer)
{
  struct timeval t;
  double end;

  gettimeofday(&t, NULL);
  end = (double) t.tv_sec + (double) t.tv_usec / 1000000.0;

  elapsed[timer] += end - start[timer];
}

double ReadTimer(int timer)
{
  return elapsed[timer];
}

void PrintTimers()
{
  StopTimer(TOTAL_TIMER);
  printf ("Total time:\t\t\t%f\n",ReadTimer(TOTAL_TIMER));
  printf ("Ray computation time:\t\t%f\n",ReadTimer(NEWRAY_TIMER));
  printf ("Shade time:\t\t\t%f\n",ReadTimer(SHADE_TIMER));
  printf ("Primitive Intersection time:\t%f\n",ReadTimer(INTERSECT_TIMER));
  printf ("Disk I/O time:\t\t\t%f\n",ReadTimer(DISK_TIMER));
  StartTimer(TOTAL_TIMER);
}
```

C TMS320C3x Assembler

As part of the TigerSHARK project, an assembler for the TMS320C32 DSP was written, using the GNU bfd [2] and binutils [22] packages. The parser and lexer are presented here.

C.1 codelex.l

```
%{
/* TI TMS320C32 assembler.
 * C. Scott Ananian 3-4-96.
 * ELE398, TigerSHARK project.
 *
 * Lexer module.
 */

#include <assert.h>
#include <string.h>
#include "strsave.h"

int hexatoi(char *s);
unsigned parse_cond(char *cc);

%}

LSYM                    [a-zA-Z0-9_]
WS                      [ \t\n]
NEWLINE                 [\n]
HEXDIGIT                [0-9a-fA-F]
DIGIT                   [0-9]
POINT                   ["."]
HEXNUMBER               {DIGIT}{HEXDIGIT}*[Hh]
C_HEXNUMBER             (0x|0X){HEXDIGIT}+
DECNUMBER               \-?{DIGIT}+
FLOATNUMBER             (\-?{DIGIT}+{POINT}{DIGIT}*)|(\-?{DIGIT}*{POINT}{DIGIT}+)
REGISTER                [rR]{DECNUMBER}
LABEL                   [a-zA-Z_]{LSYM}*
```

```
STRING                  [\"][^\"]*[\"]

C_UNCONDITIONAL         (u)|(U)
C_UNSIGNED              (lo|ls|hi|hs|eq|ne)|(LO|LS|HI|HS|EQ|NE)
C_SIGNED                (lt|le|gt|ge|eq|ne)|(LT|LE|GT|GE|EQ|NE)
C_ZERO                  (z|nz|p|n|nn)|(Z|NZ|P|N|NN)
C_CC                    (nn|n|nz|z|nv|v|nuf|uf|nc|c|nlv|lv|nluf|luf|zuf)|(NN|N|NZ|Z|NV|V|NUF
CONDITION               ({C_UNCONDITIONAL}|{C_UNSIGNED}|{C_SIGNED}|{C_ZERO}|{C_CC})

OTHER                   .

%%

ABSF|absf               {       yylval.op.type = UNARY;
                                yylval.op.inst = 0000;
                                return GENERALOP;
                        }
ABSI|absi               {       yylval.op.type = UNARY;
                                yylval.op.inst = 0001;
                                return GENERALOP;
                        }
ADDC|addc               {       yylval.op.type = TWO_OP_3;
                                yylval.op.inst = 0002;
                                yylval.op.threeop = 0100;
                                return GENERALOP;
                        }
ADDF|addf               {       yylval.op.type = TWO_OP_3;
                                yylval.op.inst = 0003;
                                yylval.op.threeop = 0101;
                                return GENERALOP;
                        }
ADDI|addi               {       yylval.op.type = TWO_OP_3;
                                yylval.op.inst = 0004;
                                yylval.op.threeop = 0102;
                                return GENERALOP;
                        }
AND|and                 {       yylval.op.type = TWO_OP_3;
```

```
                        yylval.op.inst = 0005;
                        yylval.op.threeop = 0103;
                        return GENERALOP;
                }
ANDN|andn       {       yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0006;
                        yylval.op.threeop = 0104;
                        return GENERALOP;
                }
ASH|ash         {       yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0007;
                        yylval.op.threeop = 0105;
                        return GENERALOP;
                }
CMPF|cmpf       {       yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0010;
                        yylval.op.threeop = 0106;
                        return COMPAREOP;
                }
CMPI|cmpi       {       yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0011;
                        yylval.op.threeop = 0107;
                        return COMPAREOP;
                }
FIX|fix         {       yylval.op.type = UNARY;
                        yylval.op.inst = 0012;
                        return GENERALOP;
                }
FLOAT|float     {       yylval.op.type = UNARY;
                        yylval.op.inst = 0013;
                        return GENERALOP;
                }
IDLE|idle       {       yylval.op.type = CONTROL;
                        yylval.op.inst = 0014;
                        return NO_OPD_OP;
                }
LDE|lde         {       yylval.op.type = LOAD_STORE;
```

```
                        yylval.op.inst = 0015;
                        return GENERALOP;
                    }
LDF|ldf             {   yylval.op.type = LOAD_STORE;
                        yylval.op.inst = 0016;
                        return GENERALOP;
                    }
LDFI|ldfi           {   yylval.op.type = INTERLOCK;
                        yylval.op.inst = 0017;
                        return GENERALOP;
                    }
LDI|ldi             {   yylval.op.type = LOAD_STORE;
                        yylval.op.inst = 0020;
                        return GENERALOP;
                    }
LDII|ldii           {   yylval.op.type = INTERLOCK;
                        yylval.op.inst = 0021;
                        return GENERALOP;
                    }
LDM|ldm             {   yylval.op.type = LOAD_STORE;
                        yylval.op.inst = 0022;
                        return GENERALOP;
                    }
LSH|lsh             {   yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0023;
                        yylval.op.threeop = 0110;
                        return GENERALOP;
                    }
MPYF|mpyf           {   yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0024;
                        yylval.op.threeop = 0111;
                        return GENERALOP;
                    }
MPYI|mpyi           {   yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0025;
                        yylval.op.threeop = 0112;
                        return GENERALOP;
```

```
                    }
NEGB|negb           {       yylval.op.type = UNARY;
                            yylval.op.inst = 0026;
                            return GENERALOP;
                    }
NEGF|negf           {       yylval.op.type = UNARY;
                            yylval.op.inst = 0027;
                            return GENERALOP;
                    }
NEGI|negi           {       yylval.op.type = UNARY;
                            yylval.op.inst = 0030;
                            return GENERALOP;
                    }
NOP|nop             {       yylval.op.type = CONTROL;
                            yylval.op.inst = 0031;
                            return NOP_OP;
                    }
NORM|norm           {       yylval.op.type = UNARY;
                            yylval.op.inst = 0032;
                            return GENERALOP;
                    }
NOT|not             {       yylval.op.type = UNARY;
                            yylval.op.inst = 0033;
                            return GENERALOP;
                    }
POP|pop             {       yylval.op.type = LOAD_STORE;
                            yylval.op.inst = 0034;
                            return UNARYGENERALOP;
                    }
POPF|popf           {       yylval.op.type = LOAD_STORE;
                            yylval.op.inst = 0035;
                            return UNARYGENERALOP;
                    }
PUSH|push           {       yylval.op.type = LOAD_STORE;
                            yylval.op.inst = 0036;
                            return UNARYGENERALOP;
                    }
```

```
PUSHF|pushf             {       yylval.op.type = LOAD_STORE;
                                yylval.op.inst = 0037;
                                return UNARYGENERALOP;
                        }
OR|or                   {       yylval.op.type = TWO_OP_3;
                                yylval.op.inst = 0040;
                                yylval.op.threeop = 0113;
                                return GENERALOP;
                        } /* operand 0041 skipped! */

RND|rnd                 {       yylval.op.type = UNARY;
                                yylval.op.inst = 0042;
                                return GENERALOP;
                        }
ROL|rol                 {       yylval.op.type = TWO_OP;
                                yylval.op.inst = 0043;
                                return GENERALOP;
                        }
ROLC|rolc               {       yylval.op.type = TWO_OP;
                                yylval.op.inst = 0044;
                                return GENERALOP;
                        }
ROR|ror                 {       yylval.op.type = TWO_OP;
                                yylval.op.inst = 0045;
                                return GENERALOP;
                        }
RORC|rorc               {       yylval.op.type = TWO_OP;
                                yylval.op.inst = 0046;
                                return GENERALOP;
                        }
RPTS|rpts               {       yylval.op.type = CONTROL;
                                yylval.op.inst = 0047;
                                return RPTS_OP;
                        }
STF|stf                 {       yylval.op.type = LOAD_STORE;
                                yylval.op.inst = 0050;
                                return STOREOP;
```

```
                        }
STFI|stfi               {       yylval.op.type = INTERLOCK;
                                yylval.op.inst = 0051;
                                return STOREOP;
                        }
STI|sti                 {       yylval.op.type = LOAD_STORE;
                                yylval.op.inst = 0052;
                                return STOREOP;
                        }
STII|stii               {       yylval.op.type = INTERLOCK;
                                yylval.op.inst = 0053;
                                return STOREOP;
                        }
SIGI|sigi               {       yylval.op.type = INTERLOCK;
                                yylval.op.inst = 0054;
                                return NO_OPD_OP;
                        }
SUBB|subb               {       yylval.op.type = TWO_OP_3;
                                yylval.op.inst = 0055;
                                yylval.op.threeop = 0114;
                                return GENERALOP;
                        }
SUBC|subc               {       yylval.op.type = TWO_OP;
                                yylval.op.inst = 0056;
                                return GENERALOP;
                        }
SUBF|subf               {       yylval.op.type = TWO_OP_3;
                                yylval.op.inst = 0057;
                                yylval.op.threeop = 0115;
                                return GENERALOP;
                        }
SUBI|subi               {       yylval.op.type = TWO_OP_3;
                                yylval.op.inst = 0060;
                                yylval.op.threeop = 0116;
                                return GENERALOP;
                        }
SUBRB|subrb             {       yylval.op.type = TWO_OP;
```

```
                        yylval.op.inst = 0061;
                        return GENERALOP;
                    }
SUBRF|subrf         {   yylval.op.type = TWO_OP;
                        yylval.op.inst = 0062;
                        return GENERALOP;
                    }
SUBRI|subri         {   yylval.op.type = TWO_OP;
                        yylval.op.inst = 0063;
                        return GENERALOP;
                    }
TSTB|tstb           {   yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0064;
                        yylval.op.threeop = 0117;
                        return COMPAREOP;
                    }
XOR|xor             {   yylval.op.type = TWO_OP_3;
                        yylval.op.inst = 0065;
                        yylval.op.threeop = 0120;
                        return GENERALOP;
                    }
IACK|iack           {   yylval.op.type = CONTROL;
                        yylval.op.inst = 0066;
                        return IACK_OP;
                    }
ADDC3|addc3         {   yylval.op.type = THREE_OP;
                        yylval.op.inst = 0100;
                        return THREEOP;
                    }
ADDF3|addf3         {   yylval.op.type = THREE_OP;
                        yylval.op.inst = 0101;
                        return THREEOP;
                    }
ADDI3|addi3         {   yylval.op.type = THREE_OP;
                        yylval.op.inst = 0102;
                        return THREEOP;
                    }
```

```
AND3|and3               {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0103;
                                return THREEOP;
                        }
ANDN3|andn3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0104;
                                return THREEOP;
                        }
ASH3|ash3               {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0105;
                                return THREEOP;
                        }
CMPF3|cmpf3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0106;
                                return COMPAREOP;
                        }
CMPI3|cmpi3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0107;
                                return COMPAREOP;
                        }
LSH3|lsh3               {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0110;
                                return THREEOP;
                        }
MPYF3|mpyf3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0111;
                                return THREEOP;
                        }
MPYI3|mpyi3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0112;
                                return THREEOP;
                        }
OR3|or3                 {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0113;
                                return THREEOP;
                        }
SUBB3|subb3             {       yylval.op.type = THREE_OP;
```

```
                                yylval.op.inst = 0114;
                                return THREEOP;
                        }
SUBF3|subf3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0115;
                                return THREEOP;
                        }
SUBI3|subi3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0116;
                                return THREEOP;
                        }
TSTB3|tstb3             {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0117;
                                return COMPAREOP;
                        }
XOR3|xor3               {       yylval.op.type = THREE_OP;
                                yylval.op.inst = 0120;
                                return THREEOP;
                        } /* we skip a lot of opcodes here */

LDF{CONDITION}|ldf{CONDITION}   {
                                yylval.op.type = LOAD_STORE;
                                yylval.op.cond = parse_cond(yytext+3);
                                yylval.op.inst = 0200|yylval.op.cond;
                                return GENERALOP;
                        }
LDI{CONDITION}|ldi{CONDITION}   {
                                yylval.op.type = LOAD_STORE;
                                yylval.op.cond = parse_cond(yytext+3);
                                yylval.op.inst = 0240|yylval.op.cond;
                                return GENERALOP;
                        }
BR|br                   {
                                yylval.op.type = CONTROL;
                                yylval.op.inst = 0300;
                                return LONGIMMOP;
                        }
```

```
BRD|brd                 {
                                yylval.op.type = CONTROL;
                                yylval.op.inst = 0302;
                                return LONGIMMOP;
                        }
CALL|call               {
                                yylval.op.type = CONTROL;
                                yylval.op.inst = 0304;
                                return LONGIMMOP;
                        }
RPTB|rptb               {
                                yylval.op.type = CONTROL;
                                yylval.op.inst = 0310;
                                return LONGIMMOP;
                        }
SWI|swi                 {
                                yylval.op.type = CONTROL;
                                yylval.op.inst = 0314;
                                return NO_OPD_OP;
                        }
{WS}(B|b){WS}           {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = 0; /* unconditional */
                                yylval.op.inst = 0320;
                                return CONDOP;
                        }
B{CONDITION}|b{CONDITION} {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+1);
                                yylval.op.inst = 0320;
                                return CONDOP;
                        }
B{CONDITION}D|b{CONDITION}d {   /* trap this specially */
                                yytext[yyleng-1]=0; /* kill the 'd' */
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+1);
                                yylval.op.inst = 0321; /* <- delay bit misplaced*/
```

```
                                return CONDOP;
                        }
DB{CONDITION}|db{CONDITION} {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+2);
                                yylval.op.inst = 0330;
                                return DECCONDOP;
                        }
DB{CONDITION}D|db{CONDITION}d {   /* trap this specially */
                                yytext[yyleng-1]=0; /* kill the 'd' */
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+2);
                                yylval.op.inst = 0331; /* <- delay bit misplaced*/
                                return DECCONDOP;
                        }
CALL{CONDITION}|call{CONDITION} {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+4);
                                yylval.op.inst = 0340;
                                return CONDOP;
                        }
TRAP{CONDITION}|trap{CONDITION} {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+4);
                                yylval.op.inst = 0350;
                                return TRAPOP;
                        }
RETI{CONDITION}|reti{CONDITION} {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+4);
                                yylval.op.inst = 0360;
                                return RETOP;
                        }
RETS|rets               {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = 0000;
                                yylval.op.inst = 0361;
```

```
                                return RETOP;
                        }
RETS{CONDITION}|rets{CONDITION} {
                                yylval.op.type = CONTROL;
                                yylval.op.cond = parse_cond(yytext+4);
                                yylval.op.inst = 0361;
                                return RETOP;
                        } /* parallel opcodes ignored.  we'll catch them in parse. */

LDP|ldp         { yylval.op.inst = 0020; return LDPOP; }

".global"|".GLOBAL"|".globl"|".GLOBL"   { return  DOT_GLOBAL; }
".ref"|".REF"           { return DOT_REF; }

".equ"|".EQU"|".set"|".SET"|"equ"|"EQU" { return  DOT_EQU;  }
".org"|".ORG"           { return DOT_ORG; }

".float"|".FLOAT"       { return DOT_FLOAT;}
".word"|".WORD"|".int"|".INT" { return DOT_WORD; }
".long"|".LONG"         { return DOT_WORD; }

".title"|".TITLE"       { return DOT_TITLE; }
".length"|".LENGTH"     { return DOT_LENGTH; }
".width"|".WIDTH"       { return DOT_WIDTH; }
".page"|".PAGE"         { return DOT_PAGE; }

".sect"|".SECT"|".section"|".SECTION" { return DOT_SECT; }
".data"|".DATA"         { return DOT_DATA; }
".text"|".TEXT"         { return DOT_TEXT; }
".bss"|".BSS"           { return DOT_BSS; }

".end"|".END"           { return  DOT_END; }

[rR]{DECNUMBER}         {
                          yylval.u = (unsigned) atoi(yytext+1);
                          assert((yylval.u>=0)&&(yylval.u<=7));
                          return REG;
```

```
                        }
[aA][rR]{DECNUMBER}     { yylval.u = (unsigned) atoi(yytext+2);
                          assert((yylval.u>=0)&&(yylval.u<=7));
                          yylval.u+=0x08;
                          return REG;
                        }
DP|dp                   { yylval.u = 0x10; /* data page pointer */
                          return REG;
                        }
IR0|ir0                 { yylval.u = 0x11; /* index register 0 */
                          return REG;
                        }
IR1|ir1                 { yylval.u = 0x12; /* index register 1 */
                          return REG;
                        }
BK|bk                   { yylval.u = 0x13; /* block-size register */
                          return REG;
                        }
SP|sp                   { yylval.u = 0x14; /* active stack pointer */
                          return REG;
                        }
ST|st                   { yylval.u = 0x15; /* status register */
                          return REG;
                        }
IE|ie                   { yylval.u = 0x16; /* CPU/DMA interrupt enable */
                          return REG;
                        }
IF|if                   { yylval.u = 0x17; /* CPU interrupt flags */
                          return REG;
                        }
IOF|iof                 { yylval.u = 0x18; /* I/O flags */
                          return REG;
                        }
RS|rs                   { yylval.u = 0x19; /* Repeat start address */
                          return REG;
                        }
RE|re                   { yylval.u = 0x1a; /* Repeat end address */
```

```
                        return REG;
                      }
RC|rc                 { yylval.u = 0x1b; /* Repeat counter */
                        return REG;
                      }

"@"                   {
                        return AT;
                      }

","                   {
                        return COMMA;
                      }

"||"                  {
                        return PARALLEL;
                      }

"++"                  {
                        return PP;
                      }
"--"                  {
                        return MM;
                      }
">="                  { return GE; }
"<="                  { return LE; }
"=="                  { return EQ; }
"!="                  { return NE; }

{FLOATNUMBER}         {
                        yylval.val.type = FLOAT;
                        yylval.val.val.f = (double) atof(yytext);
                        return FLOATK;
                      }

{DECNUMBER}           {
                        yylval.val.type = INT;
```

```
                        yylval.val.val.i = atoi(yytext);
                        return DECK;
                      }

{HEXNUMBER}           {
                        yylval.val.type = INT;
                        yytext[yyleng-1]=0; /* kill trailing 'h' */
                        yylval.val.val.i = hexatoi(yytext);
                        return HEXK;
                      }
{C_HEXNUMBER}         {
                        yylval.val.type = INT;
                        yylval.val.val.i = hexatoi(yytext+2);
                        return HEXK;
                      }

{STRING}              { yytext[yyleng-1] = 0; /* no trailing quote */
                        yylval.label = strsave(yytext+1); /*no leading " */
                        return STRING;
                      }

^"*"                  { int c;
                        while (((c=input()) != '\n')&&(c!=EOF)) ;
                        unput(c);
                      }

[";"]                 { int c;
                        while (((c=input()) != '\n')&&(c!=EOF)) ;
                        unput(c);
                      }

{LABEL}:?             {
                        if (strcasecmp(yytext, "B")==0) return 'B';
                        if (yytext[yyleng-1]==':')
                          yytext[yyleng-1] = 0; /* no colon */
                        yylval.label = strsave(yytext);
                        return LABEL;
```

```
                        }

{WS}                    {
                        /*
                          int i;
                          for (i = 0; i < yyleng; i++) {
                            if (yytext[i] == '\n')
                              Line_Number++;
                          }
                        */
                        }


"/*"                    {       int t;
                                loop:
                                while ((t = input()) != '*') {
                        /*        if (t == '\n')
                                    Line_Number++;
                        */      }
                                switch (input()) {
                                  case '/': break;
                                  case '*': unput('*');
                                  default:  goto loop;
                                }
                        }

{OTHER}                 { return yytext[0]; }

%%

int     hexatoi(s)
  char  *s;
{
  int a = 0;
  while (*s) {
    a <<= 4;
    switch (*s) {
```

```
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
        a += (int)(*s - '0');
        break;

      case 'a':
      case 'b':
      case 'c':
      case 'd':
      case 'e':
      case 'f':
        a += (int)(*s - 'a'+10);
        break;

      case 'A':
      case 'B':
      case 'C':
      case 'D':
      case 'E':
      case 'F':
        a += (int)(*s - 'A'+10);
        break;

      default:
        break;
    }
    s++;
  }
  return a;
```

```
}

int     yywrap()
{
  return 1;
}

unsigned parse_cond(char *cc)
{
        struct tablestruct {
                char *condition;
                unsigned value;
        };

        struct tablestruct lookup[] = {
                /* unconditional compares */
                { "U",    000 },
                /* unsigned compares */
                { "LO",   001 },
                { "LS",   002 },
                { "HI",   003 },
                { "HS",   004 },
                { "EQ",   005 },
                { "NE",   006 },
                /* signed compares */
                { "LT",   007 },
                { "LE",   010 },
                { "GT",   011 },
                { "GE",   012 },
                { "EQ",   005 },
                { "NE",   006 },
                /* compare to zero */
                { "Z",    005 },
                { "NZ",   006 },
                { "P",    011 },
                { "N",    007 },
                { "NN",   012 },
```

```
                /* compare to condition flags */
                { "NN",   012 },
                { "N",    007 },
                { "NZ",   006 },
                { "Z",    005 },
                { "NV",   014 },
                { "V",    015 },
                { "NUF",  016 },
                { "UF",   017 },
                { "NC",   004 },
                { "C",    001 },
                { "NLV",  020 },
                { "LV",   021 },
                { "NLUF", 022 },
                { "LUF",  023 },
                { "ZUF",  024 },
                { NULL,   000 } };
        struct tablestruct *tblptr;

        for (tblptr = lookup; tblptr->condition!=NULL; tblptr++)
                if (strcasecmp(tblptr->condition, cc)==0)
                        return tblptr->value;
        assert(0);
}
```

C.2 codeparse.y

```
/* TI TMS320C32 assembler.
 * C. Scott Ananian 3-4-96.
 * ELE398, TigerSHARK project.
 *
 * Parser module.
 */
%{

#include <assert.h>
#include <string.h>
```

```
#include "strsave.h"
#include "mem.h"
#include "types.h"
#include "table.h"
#include "vector.h"
#include "bfd.h"
#include "fixup.h"
#include "yyerror.h"

struct {
        bfd *bfd;
} objout;

char *output_name;
char *output_target;

typedef struct {
        char *name;
        asection *bfd_sect;
        vector_t *data;
        UINT32 start_pc, end_pc;
        arelent **rel;
} Section_T;

Table_T sect_table;
Section_T *current_section;

typedef struct {
        UINT32 mode;
        UINT32 reg;
        UINT32 disp;
} indirect_operand;

UINT32 pc=0;

UINT32 short_indirect(indirect_operand opd);
UINT32 general_indirect(indirect_operand opd);
```

```
typedef struct {
        UINT32 top, bottom, parallel;
} parallel_table;

UINT32 lookup_parallel(UINT32 top, UINT32 bottom, parallel_table *table);

#define SYM_DEFINED 0x1
#define SYM_GLOBAL  0x2
#define SYM_EQUATE  0x4

typedef struct {
        unsigned flags;
        typed_value value;
        Section_T *section;
} Symbol_T;

struct {
  Table_T priv;
  Table_T bfd;
} symtable;

Symbol_T Symbol_new(char *name, typed_value tv, Section_T *s, unsigned flags)
{       Symbol_T sym, *symp;
        symp = Table_get(symtable.priv, name);
        sym.value = tv; sym.section = s; sym.flags = flags;
        if (symp!=NULL) {
                if ((sym.flags&SYM_DEFINED)&&(symp->flags&SYM_DEFINED))
                   yywarning("Symbol Redefinition: %s", name);
                if ( (!(symp->flags&SYM_DEFINED)) &&
                     (!(symp->flags&SYM_GLOBAL ))   ) {
                   if (sym.flags&SYM_GLOBAL)
                    yyerror("Symbol %s previously referenced as local.",name);
                   else if (sym.flags&SYM_DEFINED) {
                        if (sym.section != symp->section)
                           yyerror("Illegal reference of local label %s "
                                   "from outside its section.", name);
```

```
                    }
                }
                if (!(sym.flags&SYM_DEFINED)) {
                    sym.value = symp->value;
                    sym.section = symp->section;
                }
                sym.flags|=symp->flags;
        }
        return sym;
}

int dereference_label(char *label, typed_value *tv)
{ Symbol_T *sym = Table_get(symtable.priv, label);
  if ((sym==NULL)||(!(sym->flags&SYM_DEFINED))) return 0;
  else { *tv = sym->value; return 1; }
}

void setup_sections(void);
void make_symbol_table(void);
void write_word(UINT32 word, UINT32 this_pc);
void change_section(char *name);
void write_bfd(void);
void free_sections(void);

%}

%union {
        struct {
          enum { TWO_OP, TWO_OP_3, THREE_OP, UNARY,
                 CONTROL, LOAD_STORE, INTERLOCK } type;
          UINT32 inst, threeop;
          UINT32 cond;
          enum { FLOAT_OP, INT_OP } fi;
        } op;

        UINT32 u;
        UINT32 instr;
```

```
        typed_value val;
        indirect_operand indirect;
        char * label;
        char * * sym_list;
        }

/* Terminal Symbols */

%token <op> GENERALOP THREEOP LONGIMMOP NOP_OP NO_OPD_OP RPTS_OP IACK_OP
%token <op> CONDOP DECCONDOP TRAPOP RETOP UNARYGENERALOP LDPOP COMPAREOP
%token <op> STOREOP
%token OPCODE REG COMMA PARALLEL
%token <val> FLOATK DECK HEXK
%token <label> LABEL STRING
%token '(' ')' '+' '-' '*' '/' '%' '>' '<' '~'
%token GE LE EQ NE /* >= <= == != */
%token '&' '^' '|'
%token PP MM /* ++ -- */
%token AT
%token DOT_GLOBAL DOT_REF
%token DOT_WORD DOT_FLOAT
%token DOT_EQU DOT_ORG
%token DOT_TITLE DOT_LENGTH DOT_WIDTH DOT_PAGE
%token DOT_SECT DOT_DATA DOT_TEXT DOT_BSS
%token DOT_END
%type <val> expression
%type <instr> instruction generalinstr threeopinstr condbranchinstr
%type <instr> longimmedinstr parallelinstr compareinstr miscinstr
%type <u> register direct short_immediate optional_disp
%type <u> relative
%type <indirect> indirect
%type <u> REG
%type <u> integer ldp_addr
%type <sym_list> symbol_list

/* Precedence table */
%left '~'
```

```
%left '|'
%left '^'
%left '&'
%left EQ NE
%left '<' '>' GE LE
%left '+' '-'
%left '*' '/' '%'

%%

program
        : {     /* Initialize! */
                fixup_init();
                sect_table  =   Table_new(sizeof(Section_T));
                symtable.priv = Table_new(sizeof(Symbol_T));
                symtable.bfd  = Table_new(sizeof(asymbol**));
                bfd_init();
                objout.bfd = bfd_openw(output_name, output_target);
                if(!objout.bfd) bfd_perror("Output file open"), exit(1);
                bfd_set_format(objout.bfd, bfd_object);
                bfd_set_arch_mach(objout.bfd, bfd_arch_i386, 0);
                change_section(".text");
          } lines end {
                setup_sections();
                make_symbol_table();
                fixup_sections();
                write_bfd();
                bfd_close(objout.bfd);
                Table_destroy(symtable.priv);
                Table_destroy(symtable.bfd);
                fixup_done();
                free_sections();
          }

end     :
        | DOT_END
        | DOT_END integer
```

```
          { Symbol_T sym;
            bfd_set_start_address(objout.bfd, $2);
            /* Um, this isn't great, 'cause we don't really know the */
            /* correct section for this symbol.  We'll deal.         */
            sym = Symbol_new("_start", $2, current_section,
                             SYM_DEFINED|SYM_GLOBAL);
            Table_put(symtable.priv, "_start", &sym);
          }

lines
        : line
        | lines line

line
        : LABEL
          { Symbol_T sym;
            sym = Symbol_new($1, type_int(pc), current_section, SYM_DEFINED);
            Table_put(symtable.priv, $1, &sym);
          } instr_field
        | instr_field
        | pseudoop
        | LABEL DOT_EQU expression
          { Symbol_T sym = Symbol_new($1, $3, NULL, SYM_EQUATE|SYM_DEFINED);
            Table_put(symtable.priv,  $1, &sym); }
        | DOT_ORG integer

allocspace
        : DOT_WORD expression
          { write_word(fixup_integer($2,32,0,0), pc++); }
        | DOT_FLOAT expression
          { write_word(fixup_float  ($2,32,0),   pc++); }

pseudoop
        : DOT_GLOBAL symbol_list
          { int i; Symbol_T sym;
                for (i=0; i<vector_length($2); i++) {
                   sym = Symbol_new(($2)[i], type_int(0), NULL, SYM_GLOBAL);
```

```
                Table_put(symtable.priv, ($2)[i], &sym);
             }
             vector_destroy($2);
          }
        | DOT_REF symbol_list
          { int i; Symbol_T sym;
             for (i=0; i<vector_length($2); i++) {
                sym = Symbol_new(($2)[i], type_int(0), NULL, SYM_GLOBAL);
                Table_put(symtable.priv, ($2)[i], &sym);
             }
             vector_destroy($2);
          }
        | DOT_SECT STRING
          { change_section($2); }
        | DOT_TEXT
          { change_section(".text"); }
        | DOT_DATA
          { change_section(".data"); }
        | DOT_BSS
          { change_section(".bss"); }
        | DOT_TITLE STRING
        | DOT_LENGTH integer
        | DOT_WIDTH integer
        | DOT_PAGE

symbol_list
        : LABEL
          { $$ = vector_create(sizeof(char *), 19, 0);
            $$ = vector_extend($$, &($1));
          }
        | symbol_list COMMA LABEL
          { $$ = vector_extend($1, &($3)); }

instr_field
        : instruction
          { write_word($1, pc++); }
        | allocspace
```

```
instruction
        : generalinstr
        | threeopinstr
        | parallelinstr
        | longimmedinstr
        | condbranchinstr
        | compareinstr
        | miscinstr

parallelinstr
        :          GENERALOP indirect COMMA register
          PARALLEL STOREOP register COMMA indirect
           { parallel_table tbl[] = {
                { 0000, 0050, 0620 }, /* ABSF || STF */
                { 0001, 0052, 0624 }, /* ABSI || STI */
                { 0012, 0052, 0650 }, /* FIX  || STI */
                { 0013, 0050, 0654 }, /* FLOAT|| STF */
                { 0016, 0050, 0660 }, /* LDF  || STF */
                { 0020, 0052, 0664 }, /* LDI  || STI */
                { 0027, 0050, 0704 }, /* NEGF || STF */
                { 0030, 0052, 0710 }, /* NEGI || STI */
                { 0033, 0052, 0714 }, /* NOT  || STI */
                {0,0,0} };
             UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
             $$ = (inst<<23)|($4<<22)|($7<<16)|
                  (short_indirect($9)<<8)|(short_indirect($2));
           }
        |          STOREOP register COMMA indirect
          PARALLEL GENERALOP indirect COMMA register
           { /* reverse orientation */
             parallel_table tbl[] = {
                { 0050, 0000, 0620 }, /* STF || ABSF (reverse) */
                { 0052, 0001, 0624 }, /* STI || ABSI (reverse) */
                { 0052, 0012, 0650 }, /* STI || FIX  (reverse) */
                { 0050, 0013, 0654 }, /* STF || FLOAT(reverse) */
                { 0050, 0016, 0660 }, /* STF || LDF  (reverse) */
```

```
        { 0052, 0020, 0664 }, /* STI || LDI  (reverse) */
        { 0050, 0027, 0704 }, /* STF || NEGF (reverse) */
        { 0052, 0030, 0710 }, /* STI || NEGI (reverse) */
        { 0052, 0033, 0714 }, /* STI || NOT  (reverse) */
        {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
      $$ = (inst<<23)|($9<<22)|($2<<16)|
           (short_indirect($4)<<8)|(short_indirect($7));
    }
|         THREEOP indirect COMMA register COMMA register
  PARALLEL STOREOP register COMMA indirect
    { parallel_table tbl[] = {
        { 0101, 0050, 0630 }, /* ADDF3 || STF */
        { 0102, 0052, 0634 }, /* ADDI3 || STI */
        { 0103, 0052, 0640 }, /* AND3  || STI */
        { 0111, 0050, 0674 }, /* MPYF3 || STF */
        { 0112, 0052, 0700 }, /* MPYI3 || STI */
        { 0113, 0052, 0720 }, /* OR3   || STI */
        { 0120, 0052, 0734 }, /* XOR3  || STI */
        {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
      $$ = (inst<<23)|($6<<22)|($4<<19)|($9<<16)|
           (short_indirect($11)<<8)|(short_indirect($2));
    }
|         STOREOP register COMMA indirect
  PARALLEL THREEOP indirect COMMA register COMMA register
    { parallel_table tbl[] = {
        { 0050, 0101, 0630 }, /* STF || ADDF3 (Reverse) */
        { 0052, 0102, 0634 }, /* STI || ADDI3 (Reverse) */
        { 0052, 0103, 0640 }, /* STI || AND3  (Reverse) */
        { 0050, 0111, 0674 }, /* STF || MPYF3 (Reverse) */
        { 0052, 0112, 0700 }, /* STI || MPYI3 (Reverse) */
        { 0052, 0113, 0720 }, /* STI || OR3   (Reverse) */
        { 0052, 0120, 0734 }, /* STI || XOR3  (Reverse) */
        {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
      $$ = (inst<<23)|($11<<22)|($9<<19)|($2<<16)|
```

```
            (short_indirect($4)<<8)|(short_indirect($7));
    }
|           GENERALOP indirect COMMA register COMMA register
    PARALLEL STOREOP   register COMMA indirect
    { parallel_table tbl[] = {
         { 0003, 0050, 0630 }, /* ADDF || STF */
         { 0004, 0052, 0634 }, /* ADDI || STI */
         { 0005, 0052, 0640 }, /* AND  || STI */
         { 0024, 0050, 0674 }, /* MPYF || STF */
         { 0025, 0052, 0700 }, /* MPYI || STI */
         { 0040, 0052, 0720 }, /* OR   || STI */
         { 0065, 0052, 0734 }, /* XOR  || STI */
         {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
      $$ = (inst<<23)|($6<<22)|($4<<19)|($9<<16)|
           (short_indirect($11)<<8)|(short_indirect($2));
    }
|           STOREOP   register COMMA indirect
    PARALLEL GENERALOP indirect COMMA register COMMA register
    { parallel_table tbl[] = {
         { 0050, 0003, 0630 }, /* STF || ADDF (Reverse) */
         { 0052, 0004, 0634 }, /* STI || ADDI (Reverse) */
         { 0052, 0005, 0640 }, /* STI || AND  (Reverse) */
         { 0050, 0024, 0674 }, /* STF || MPYF (Reverse) */
         { 0052, 0025, 0700 }, /* STI || MPYI (Reverse) */
         { 0052, 0040, 0720 }, /* STI || OR   (Reverse) */
         { 0052, 0065, 0734 }, /* STI || XOR  (Reverse) */
         {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
      $$ = (inst<<23)|($11<<22)|($9<<19)|($2<<16)|
           (short_indirect($4)<<8)|(short_indirect($7));
    }
|           THREEOP register COMMA indirect COMMA register
    PARALLEL STOREOP register COMMA indirect
    { parallel_table tbl[] = {
         { 0105, 0052, 0644 }, /* ASH3  || STI */
         { 0110, 0052, 0670 }, /* LSH3  || STI */
```

```
        { 0115, 0050, 0724 }, /* SUBF3 || STF */
        { 0116, 0052, 0730 }, /* SUBI3 || STI */

        { 0101, 0050, 0630 }, /* ADDF3 || STF */
        { 0102, 0052, 0634 }, /* ADDI3 || STI */
        { 0103, 0052, 0640 }, /* AND3  || STI */
        { 0111, 0050, 0674 }, /* MPYF3 || STF */
        { 0112, 0052, 0700 }, /* MPYI3 || STI */
        { 0113, 0052, 0720 }, /* OR3   || STI */
        { 0120, 0052, 0734 }, /* XOR3  || STI */
        {0,0,0} };
    UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
    $$ = (inst<<23)|($6<<22)|($2<<19)|($9<<16)|
         (short_indirect($11)<<8)|(short_indirect($4));
  }
|         STOREOP register COMMA indirect
  PARALLEL THREEOP register COMMA indirect COMMA register
  { parallel_table tbl[] = {
        { 0052, 0105, 0644 }, /* STI || ASH3  (Reverse) */
        { 0052, 0110, 0670 }, /* STI || LSH3  (Reverse) */
        { 0050, 0115, 0724 }, /* STF || SUBF3 (Reverse) */
        { 0052, 0116, 0730 }, /* STI || SUBI3 (Reverse) */

        { 0050, 0101, 0630 }, /* STF || ADDF3 (Reverse) */
        { 0052, 0102, 0634 }, /* STI || ADDI3 (Reverse) */
        { 0052, 0103, 0640 }, /* STI || AND3  (Reverse) */
        { 0050, 0111, 0674 }, /* STF || MPYF3 (Reverse) */
        { 0052, 0112, 0700 }, /* STI || MPYI3 (Reverse) */
        { 0052, 0113, 0720 }, /* STI || OR3   (Reverse) */
        { 0052, 0120, 0734 }, /* STI || XOR3  (Reverse) */
        {0,0,0} };
    UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
    $$ = (inst<<23)|($11<<22)|($7<<19)|($2<<16)|
         (short_indirect($4)<<8)|(short_indirect($9));
  }
|         GENERALOP register COMMA indirect COMMA register
  PARALLEL STOREOP   register COMMA indirect
```

```
    { parallel_table tbl[] = {
        { 0007, 0052, 0644 }, /* ASH  || STI */
        { 0023, 0052, 0670 }, /* LSH  || STI */
        { 0057, 0050, 0724 }, /* SUBF || STF */
        { 0060, 0052, 0730 }, /* SUBI || STI */

        { 0003, 0050, 0630 }, /* ADDF || STF */
        { 0004, 0052, 0634 }, /* ADDI || STI */
        { 0005, 0052, 0640 }, /* AND  || STI */
        { 0024, 0050, 0674 }, /* MPYF || STF */
        { 0025, 0052, 0700 }, /* MPYI || STI */
        { 0040, 0052, 0720 }, /* OR   || STI */
        { 0065, 0052, 0734 }, /* XOR  || STI */

        {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
      $$ = (inst<<23)|($6<<22)|($2<<19)|($9<<16)|
           (short_indirect($11)<<8)|(short_indirect($4));
    }
|          STOREOP   register COMMA indirect
  PARALLEL GENERALOP register COMMA indirect COMMA register
    { parallel_table tbl[] = {
        { 0052, 0007, 0644 }, /* STI || ASH  (Reverse) */
        { 0052, 0023, 0670 }, /* STI || LSH  (Reverse) */
        { 0050, 0057, 0724 }, /* STF || SUBF (Reverse) */
        { 0052, 0060, 0730 }, /* STI || SUBI (Reverse) */

        { 0050, 0003, 0630 }, /* STF || ADDF (Reverse) */
        { 0052, 0004, 0634 }, /* STI || ADDI (Reverse) */
        { 0052, 0005, 0640 }, /* STI || AND  (Reverse) */
        { 0050, 0024, 0674 }, /* STF || MPYF (Reverse) */
        { 0052, 0025, 0700 }, /* STI || MPYI (Reverse) */
        { 0052, 0040, 0720 }, /* STI || OR   (Reverse) */
        { 0052, 0065, 0734 }, /* STI || XOR  (Reverse) */
        {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
      $$ = (inst<<23)|($11<<22)|($7<<19)|($2<<16)|
```

```
            (short_indirect($4)<<8)|(short_indirect($9));
    }
|          STOREOP register COMMA indirect
  PARALLEL STOREOP register COMMA indirect
    { parallel_table tbl[] = {
         { 0050, 0050, 0600 }, /* STF || STF             */
         { 0052, 0052, 0604 }, /* STI || STI             */
         {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
      $$ = (inst<<23)|($2<<22)|($7<<16)|
            (short_indirect($9)<<8)|(short_indirect($4));
    }
|          GENERALOP indirect COMMA register
  PARALLEL GENERALOP indirect COMMA register
    { parallel_table tbl[] = {
         { 0016, 0016, 0610 }, /* LDF || LDF             */
         { 0020, 0020, 0614 }, /* LDI || LDI             */
         {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $6.inst, tbl);
      $$ = (inst<<23)|($4<<22)|($9<<16)|
            (short_indirect($7)<<8)|(short_indirect($2));
    }
|          THREEOP indirect COMMA indirect COMMA register
  PARALLEL THREEOP register COMMA register COMMA register
    { parallel_table tbl[] = {
         { 0111, 0101, 0400 }, /* MPYF3 || ADDF3         */
         { 0111, 0115, 0410 }, /* MPYF3 || SUBF3         */
         { 0112, 0102, 0420 }, /* MPYI3 || ADDI3         */
         { 0112, 0116, 0430 }, /* MPYI3 || SUBI3         */

         { 0101, 0111, 0404 }, /* ADDF3 || MPYF3         */
         { 0115, 0111, 0414 }, /* SUBF3 || MPYF3         */
         { 0102, 0112, 0424 }, /* ADDI3 || MPYI3         */
         { 0116, 0112, 0434 }, /* SUBI3 || MPYI3         */
         {0,0,0} };
      UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
      if ((inst&7)==0) {
```

```
            if (($6!=0)&&($6!=1))
               yyerror("Dest register for MPYF3 must be R0 or R1.");
            if (($13!=2)&&($13!=3))
               yyerror("Dest register for ADDF3 must be R2 or R3.");
            $$ = (inst<<23)|($6<<23)|(($13-2)<<22)|
               ($11<<19)|($9<<16)|
               (short_indirect($4)<<8)|(short_indirect($2));
         } else {
            if (($13!=0)&&($13!=1))
               yyerror("Dest register for MPYF3 must be R0 or R1.");
            if (($6!=2)&&($6!=3))
               yyerror("Dest register for ADDF3 must be R2 or R3.");
            $$ = (inst<<23)|($13<<23)|(($6-2)<<22)|
                 ($11<<19)|($9<<16)|
                 (short_indirect($4)<<8)|(short_indirect($2));
         }
       }
    |          THREEOP indirect COMMA register COMMA register
      PARALLEL THREEOP indirect COMMA register COMMA register
       { parallel_table tbl[] = {
            { 0111, 0101, 0406 }, /* MPYF3 || ADDF3         */
            { 0111, 0115, 0416 }, /* MPYF3 || SUBF3         */
            { 0112, 0102, 0426 }, /* MPYI3 || ADDI3         */
            { 0112, 0116, 0436 }, /* MPYI3 || SUBI3         */

            { 0101, 0111, 0407 }, /* ADDF3 || MPYF3         */
            { 0115, 0111, 0417 }, /* SUBF3 || MPYF3         */
            { 0102, 0112, 0427 }, /* ADDI3 || MPYI3         */
            { 0116, 0112, 0437 }, /* SUBI3 || MPYI3         */
            {0,0,0} };
         UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
         if (!(inst&1)) {
            if (($6!=0)&&($6!=1))
               yyerror("Dest register for MPYF3 must be R0 or R1.");
            if (($13!=2)&&($13!=3))
               yyerror("Dest register for ADDF3 must be R2 or R3.");
            $$ = (inst<<23)|($6<<23)|(($13-2)<<22)|
```

```
                ($4<<19)|($11<<16)|
                (short_indirect($2)<<8)|(short_indirect($9));
          } else {
             inst&=0776;
             if (($13!=0)&&($13!=1))
                yyerror("Dest register for MPYF3 must be R0 or R1.");
             if (($6!=2)&&($6!=3))
                yyerror("Dest register for ADDF3 must be R2 or R3.");
             $$ = (inst<<23)|($13<<23)|(($6-2)<<22)|
                ($11<<19)|($4<<16)|
                (short_indirect($9)<<8)|(short_indirect($2));
          }
        }
      |         THREEOP register COMMA register COMMA register
        PARALLEL THREEOP indirect COMMA indirect COMMA register
        { parallel_table tbl[] = {
             { 0111, 0101, 0404 }, /* MPYF3 || ADDF3         */
             { 0111, 0115, 0414 }, /* MPYF3 || SUBF3         */
             { 0112, 0102, 0424 }, /* MPYI3 || ADDI3         */
             { 0112, 0116, 0434 }, /* MPYI3 || SUBI3         */

             { 0101, 0111, 0400 }, /* ADDF3 || MPYF3         */
             { 0115, 0111, 0410 }, /* SUBF3 || MPYF3         */
             { 0102, 0112, 0420 }, /* ADDI3 || MPYI3         */
             { 0116, 0112, 0430 }, /* SUBI3 || MPYI3         */
             {0,0,0} };
          UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
          if ((inst&7)==4) {
             if (($6!=0)&&($6!=1))
                yyerror("Dest register for MPYF3 must be R0 or R1.");
             if (($13!=2)&&($13!=3))
                yyerror("Dest register for ADDF3 must be R2 or R3.");
             $$ = (inst<<23)|($6<<23)|(($13-2)<<22)|
                ($4<<19)|($2<<16)|
                (short_indirect($11)<<8)|(short_indirect($9));
          } else {
             if (($13!=0)&&($13!=1))
```

```
            yyerror("Dest register for MPYF3 must be R0 or R1.");
         if (($6!=2)&&($6!=3))
            yyerror("Dest register for ADDF3 must be R2 or R3.");
         $$ = (inst<<23)|($13<<23)|(($6-2)<<22)|
              ($4<<19)|($2<<16)|
              (short_indirect($11)<<8)|(short_indirect($9));
      }
     }
|          GENERALOP indirect COMMA indirect COMMA register
  PARALLEL GENERALOP register COMMA register COMMA register
     { parallel_table tbl[] = {
         { 0024, 0003, 0400 }, /* MPYF || ADDF         */
         { 0024, 0057, 0410 }, /* MPYF || SUBF         */
         { 0025, 0004, 0420 }, /* MPYI || ADDI         */
         { 0025, 0060, 0430 }, /* MPYI || SUBI         */

         { 0003, 0024, 0404 }, /* ADDF || MPYF         */
         { 0057, 0024, 0414 }, /* SUBF || MPYF         */
         { 0004, 0025, 0424 }, /* ADDI || MPYI         */
         { 0060, 0025, 0434 }, /* SUBI || MPYI         */

         {0,0,0} };
       UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
       if ((inst&7)==0) {
         if (($6!=0)&&($6!=1))
            yyerror("Dest register for MPYF3 must be R0 or R1.");
         if (($13!=2)&&($13!=3))
            yyerror("Dest register for ADDF3 must be R2 or R3.");
         $$ = (inst<<23)|($6<<23)|(($13-2)<<22)|
            ($11<<19)|($9<<16)|
            (short_indirect($4)<<8)|(short_indirect($2));
       } else {
         if (($13!=0)&&($13!=1))
            yyerror("Dest register for MPYF3 must be R0 or R1.");
         if (($6!=2)&&($6!=3))
            yyerror("Dest register for ADDF3 must be R2 or R3.");
         $$ = (inst<<23)|($13<<23)|(($6-2)<<22)|
```

```
                ($11<<19)|($9<<16)|
                (short_indirect($4)<<8)|(short_indirect($2));
        }
      }
    |          GENERALOP indirect COMMA register COMMA register
      PARALLEL GENERALOP indirect COMMA register COMMA register
       { parallel_table tbl[] = {
            { 0024, 0003, 0406 }, /* MPYF || ADDF         */
            { 0024, 0057, 0416 }, /* MPYF || SUBF         */
            { 0025, 0004, 0426 }, /* MPYI || ADDI         */
            { 0025, 0060, 0436 }, /* MPYI || SUBI         */

            { 0003, 0024, 0407 }, /* ADDF || MPYF         */
            { 0057, 0024, 0417 }, /* SUBF || MPYF         */
            { 0004, 0025, 0427 }, /* ADDI || MPYI         */
            { 0060, 0025, 0437 }, /* SUBI || MPYI         */

            {0,0,0} };
         UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
         if (!(inst&1)) {
            if (($6!=0)&&($6!=1))
               yyerror("Dest register for MPYF3 must be R0 or R1.");
            if (($13!=2)&&($13!=3))
               yyerror("Dest register for ADDF3 must be R2 or R3.");
            $$ = (inst<<23)|($6<<23)|(($13-2)<<22)|
                 ($4<<19)|($11<<16)|
                 (short_indirect($2)<<8)|(short_indirect($9));
         } else {
            inst&=0776;
            if (($13!=0)&&($13!=1))
               yyerror("Dest register for MPYF3 must be R0 or R1.");
            if (($6!=2)&&($6!=3))
               yyerror("Dest register for ADDF3 must be R2 or R3.");
            $$ = (inst<<23)|($13<<23)|(($6-2)<<22)|
                 ($11<<19)|($4<<16)|
                 (short_indirect($9)<<8)|(short_indirect($2));
         }
```

```
          }
        |         GENERALOP register COMMA register COMMA register
          PARALLEL GENERALOP indirect COMMA indirect COMMA register
          { parallel_table tbl[] = {
               { 0024, 0003, 0404 }, /* MPYF || ADDF         */
               { 0024, 0057, 0414 }, /* MPYF || SUBF         */
               { 0025, 0004, 0424 }, /* MPYI || ADDI         */
               { 0025, 0060, 0434 }, /* MPYI || SUBI         */

               { 0003, 0024, 0400 }, /* ADDF || MPYF         */
               { 0057, 0024, 0410 }, /* SUBF || MPYF         */
               { 0004, 0025, 0420 }, /* ADDI || MPYI         */
               { 0060, 0025, 0430 }, /* SUBI || MPYI         */

               {0,0,0} };
            UINT32 inst = lookup_parallel($1.inst, $8.inst, tbl);
            if ((inst&7)==4) {
               if (($6!=0)&&($6!=1))
                  yyerror("Dest register for MPYF3 must be R0 or R1.");
               if (($13!=2)&&($13!=3))
                  yyerror("Dest register for ADDF3 must be R2 or R3.");
               $$ = (inst<<23)|($6<<23)|(($13-2)<<22)|
                    ($4<<19)|($2<<16)|
                    (short_indirect($11)<<8)|(short_indirect($9));
            } else {
               if (($13!=0)&&($13!=1))
                  yyerror("Dest register for MPYF3 must be R0 or R1.");
               if (($6!=2)&&($6!=3))
                  yyerror("Dest register for ADDF3 must be R2 or R3.");
               $$ = (inst<<23)|($13<<23)|(($6-2)<<22)|
                    ($4<<19)|($2<<16)|
                    (short_indirect($11)<<8)|(short_indirect($9));
            }
          }

condbranchinstr
        : DECCONDOP register COMMA register
```

```
          { assert(($1.inst>>6)==03);
            if(($2>>3)!=01) /* check register type */
              yyerror("Invalid register for branch (AR0-AR7 only)");
            $$ = (($1.inst&0770)<<23)|(00<<25)|(($2&07)<<22)|
                 (($1.inst&1)<<21)|($1.cond<<16)|($4);
          }
        | DECCONDOP register COMMA relative
          { assert(($1.inst>>6)==03);
            if(($2>>3)!=01) /* check register type */
              yyerror("Invalid register for branch (AR0-AR7 only)");
            $$ = (($1.inst&0770)<<23)|(01<<25)|(($2&07)<<22)|
                 (($1.inst&1)<<21)|($1.cond<<16)|($4);
          }
        | CONDOP register
          { assert(($1.inst>>6)==03);
            $$ = (($1.inst&0770)<<23)|(00<<25)|
                 (($1.inst&1)<<21)|($1.cond<<16)|($2);
          }
        | CONDOP relative
          { assert(($1.inst>>6)==03);
            $$ = (($1.inst&0770)<<23)|(01<<25)|
                 (($1.inst&1)<<21)|($1.cond<<16)|($2);
          }

relative
        : expression
          { $$ = fixup_relative($1, 16, 0, pc+3); }

miscinstr
        : NO_OPD_OP
          { $$ = ($1.inst<<23); }
        | NOP_OP
          { $$ = ($1.inst<<23); }
        | NOP_OP indirect
          { $$ = ($1.inst<<23)|(02<<21)|(general_indirect($2)); }
        | RPTS_OP register
          { $$ = ($1.inst<<23)|(00<<21)|(033<<16)|($2); }
```

```
        | RPTS_OP direct
          { $$ = ($1.inst<<23)|(01<<21)|(033<<16)|($2); }
        | RPTS_OP indirect
          { $$ = ($1.inst<<23)|(02<<21)|(033<<16)|(general_indirect($2)); }
        | RPTS_OP short_immediate
          { $$ = ($1.inst<<23)|(03<<21)|(033<<16)|($2); }
        | IACK_OP direct
          { $$ = ($1.inst<<23)|(01<<21)|($2); }
        | IACK_OP indirect
          { $$ = ($1.inst<<23)|(02<<21)|(general_indirect($2)); }
        | TRAPOP expression
          { $$ = ($1.inst<<23)|($1.cond<<16)|fixup_integer($2, 5, 0, 0); }
        | RETOP
          { $$ = ($1.inst<<23)|($1.cond<<16); }
        | LDPOP ldp_addr
          { $$ = ($1.inst<<23)|(03<<21)|(020<<16)|($2); }
        | LDPOP ldp_addr COMMA register
          { $$ = ($1.inst<<23)|(03<<21)|($4<<16)|($2); }

ldp_addr
        : expression
          { $$ = fixup_integer($1, 24, -16, 0);   }
        | AT expression
          { $$ = fixup_integer($2, 24, -16, 0);   }

integer
        : expression
          { typed_value tvi = type_int(0);
            type_promote(&$1, &tvi);
            if ($1.type != INT)
                  yyerror("Non-integer operand (requires int).");
            $$ = $1.val.i;
          }

longimmedinstr
        : LONGIMMOP expression
          { assert((($1.inst>>3)|1)==031); /* long immediate addr mode */
```

```
            $$ = ($1.inst<<23)|fixup_integer($2, 24, 0, 0);
          }
        | LONGIMMOP AT expression
          { assert((($1.inst>>3)|1)==031); /* long immediate addr mode */
            $$ = ($1.inst<<23)|fixup_integer($3, 24, 0, 0);
          }

threeopinstr
        : THREEOP register COMMA register COMMA register
          { assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(00<<21)|($6<<16)|($4<<8)|($2);
          }
        | THREEOP indirect COMMA register COMMA register
          { assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(01<<21)|($6<<16)|($4<<8)|(short_indirect($2));
          }
        | THREEOP register COMMA indirect COMMA register
          { assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(02<<21)|($6<<16)|(short_indirect($4)<<8)|($2);
          }
        | THREEOP indirect COMMA indirect COMMA register
          { assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(03<<21)|($6<<16)|(short_indirect($4)<<8)|(short_indirect($2)
          }
        | GENERALOP register COMMA register COMMA register
          { if ($1.type != TWO_OP_3) yyerror("Not a 3-operand instr.");
            $1.inst = $1.threeop;
            assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(00<<21)|($6<<16)|($4<<8)|($2);
          }
        | GENERALOP indirect COMMA register COMMA register
          { if ($1.type != TWO_OP_3) yyerror("Not a 3-operand instr.");
            $1.inst = $1.threeop;
            assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(01<<21)|($6<<16)|($4<<8)|(short_indirect($2));
          }
        | GENERALOP register COMMA indirect COMMA register
```

```
          { if ($1.type != TWO_OP_3) yyerror("Not a 3-operand instr.");
            $1.inst = $1.threeop;
            assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(02<<21)|($6<<16)|(short_indirect($4)<<8)|($2);
          }
        | GENERALOP indirect COMMA indirect COMMA register
          { if ($1.type != TWO_OP_3) yyerror("Not a 3-operand instr.");
            $1.inst = $1.threeop;
            assert(($1.inst>>6)==001); /* three operand addressing mode */
            $$ = ($1.inst<<23)|(03<<21)|($6<<16)|(short_indirect($4)<<8)|(short_indirect($2)
          }

generalinstr
        : GENERALOP register COMMA register
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(00<<21)|($4<<16)|($2);
          }
        | GENERALOP register
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            if ($1.type!=UNARY) yyerror("Not a unary opcode.");
            $$ = ($1.inst<<23)|(00<<21)|($2<<16)|($2);
          }
        | GENERALOP direct COMMA register
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(01<<21)|($4<<16)|($2);
          }
        | GENERALOP indirect COMMA register
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(02<<21)|($4<<16)|(general_indirect($2));
          }
        | GENERALOP short_immediate COMMA register
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(03<<21)|($4<<16)|($2);
          }
        | UNARYGENERALOP register
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(01<<21)|($2<<16);
```

```
          }
        | STOREOP register COMMA register
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(00<<21)|($2<<16)|($4);
          }
        | STOREOP register COMMA direct
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(01<<21)|($2<<16)|($4);
          }
        | STOREOP register COMMA indirect
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(02<<21)|($2<<16)|(general_indirect($4));
          }
        | STOREOP register COMMA short_immediate
          { assert((($1.inst>>6)|2)==002); /* general addressing mode */
            $$ = ($1.inst<<23)|(03<<21)|($2<<16)|($4);
          }

compareinstr
        : COMPAREOP register COMMA register
          { if ($1.type == THREE_OP)
               $$ = ($1.inst<<23)|(00<<21)|($4<<8)|($2);
            else
               $$ = ($1.inst<<23)|(00<<21)|($4<<16)|($2);
          }
        | COMPAREOP direct COMMA register
          { if ($1.type == THREE_OP)
                yyerror("Direct addressing illegal for 3-operand mode.");
            $$ = ($1.inst<<23)|(01<<21)|($4<<16)|($2);
          }
        | COMPAREOP short_immediate COMMA register
          { if ($1.type == THREE_OP)
                yyerror("Immediate addressing illegal for 3-operand mode.");
            $$ = ($1.inst<<23)|(03<<21)|($4<<16)|($2);
          }
        | COMPAREOP indirect COMMA register
          { if ($1.type == THREE_OP)
```

```
            $$ = ($1.inst<<23)|(01<<21)|($4<<8)|(short_indirect($2));
          else
            $$ = ($1.inst<<23)|(02<<21)|($4<<16)|(general_indirect($2));
        }
      | COMPAREOP register COMMA indirect
        { if ($1.type != THREE_OP)
            $1.inst = $1.threeop;
          $$ = ($1.inst<<23)|(02<<21)|(short_indirect($4)<<8)|($2);
        }
      | COMPAREOP indirect COMMA indirect
        { if ($1.type != THREE_OP)
            $1.inst = $1.threeop;
          $$ = ($1.inst<<23)|(03<<21)|(short_indirect($4)<<8)|
               (short_indirect($2));
        }

register
      : REG

direct
      : AT expression
        { $$ = fixup_address($2, 16, 0, 0);  }

indirect
      : '*' '+' REG optional_disp
        { $$.mode = 000; $$.reg = $3; $$.disp = $4; }
      | '*' '-' REG optional_disp
        { $$.mode = 001; $$.reg = $3; $$.disp = $4; }
      | '*' PP REG optional_disp
        { $$.mode = 002; $$.reg = $3; $$.disp = $4; }
      | '*' MM REG optional_disp
        { $$.mode = 003; $$.reg = $3; $$.disp = $4; }
      | '*' REG PP optional_disp
        { $$.mode = 004; $$.reg = $2; $$.disp = $4; }
      | '*' REG MM optional_disp
        { $$.mode = 005; $$.reg = $2; $$.disp = $4; }
      | '*' REG PP optional_disp '%'
```

```
  { $$.mode = 006; $$.reg = $2; $$.disp = $4; }
| '*' REG MM optional_disp '%'
  { $$.mode = 007; $$.reg = $2; $$.disp = $4; }
| '*' '+' REG '(' REG ')'
  { if      ($5 == 0x11) $$.mode = 010;
    else if ($5 == 0x12) $$.mode = 020;
    else yyerror("Invalid Index Register r%d.",$5);
    $$.reg = $3; $$.disp = 0;
  }
| '*' '-' REG '(' REG ')'
  { if      ($5 == 0x11) $$.mode = 011;
    else if ($5 == 0x12) $$.mode = 021;
    else yyerror("Invalid Index Register r%d.",$5);
    $$.reg = $3; $$.disp = 0;
  }
| '*' PP  REG '(' REG ')'
  { if      ($5 == 0x11) $$.mode = 012;
    else if ($5 == 0x12) $$.mode = 022;
    else yyerror("Invalid Index Register r%d.",$5);
    $$.reg = $3; $$.disp = 0;
  }
| '*' MM  REG '(' REG ')'
  { if      ($5 == 0x11) $$.mode = 013;
    else if ($5 == 0x12) $$.mode = 023;
    else yyerror("Invalid Index Register r%d.",$5);
    $$.reg = $3; $$.disp = 0;
  }
| '*' REG PP  '(' REG ')'
  { if      ($5 == 0x11) $$.mode = 014;
    else if ($5 == 0x12) $$.mode = 024;
    else yyerror("Invalid Index Register r%d.",$5);
    $$.reg = $2; $$.disp = 0;
  }
| '*' REG MM  '(' REG ')'
  { if      ($5 == 0x11) $$.mode = 015;
    else if ($5 == 0x12) $$.mode = 025;
    else yyerror("Invalid Index Register r%d.",$5);
```

```
            $$.reg = $2; $$.disp = 0;
          }
        | '*' REG PP  '(' REG ')' '%'
          { if      ($5 == 0x11) $$.mode = 016;
            else if ($5 == 0x12) $$.mode = 026;
            else yyerror("Invalid Index Register r%d.",$5);
            $$.reg = $2; $$.disp = 0;
          }
        | '*' REG MM  '(' REG ')' '%'
          { if      ($5 == 0x11) $$.mode = 017;
            else if ($5 == 0x12) $$.mode = 027;
            else yyerror("Invalid Index Register r%d.",$5);
            $$.reg = $2; $$.disp = 0;
          }
        | '*' REG
          { $$.mode = 030; $$.reg = $2; $$.disp = 0; }
        | '*' REG PP '(' REG ')' 'B'
          { if ($5==0x11) $$.mode = 031;
            else yyerror("Invalid Index Register for Bit reversed modify.");
            $$.reg = $2; $$.disp = 0;
          }

optional_disp
        :
          { $$ = 1; } /* null displacement */
        | '(' expression ')'
          { if ($2.type != INT) /* check type of constant */
              yyerror("Displacement not an integer.");
            $$ = $2.val.i;
          }

short_immediate
        : expression
           { if ($1.type == FLOAT) $$ = fixup_float  ($1, 16, 0);
             else                  $$ = fixup_integer($1, 16, 0, 2);
             /* Note that the format (signed/unsigned) *changes* from */
             /* instruction to instruction.  We don't check for that..yet */
```

```
            /* instead we sidestep by passing the 'magic' value of */
            /* 2 to is_signed in fixup_integer */
            assert(($$&(~0xFFFF))==0);
          }

expression
        : LABEL
          {    Symbol_T *sym;
               sym = Table_get(symtable.priv, $1);
               if ((sym!=NULL)&&
                   ( (sym->flags&SYM_GLOBAL) ||
                     (sym->section!=current_section) ) ) {
                 $$.type = GLOBALREF;
                 $$.val.reloc.label = strsave($1);  /* def: "label+0" */
                 $$.val.reloc.disp = mem_alloc(sizeof(typed_value));
                 *($$.val.reloc.disp) = type_int(0);
               } else {
                 if (sym==NULL) {
                   /* record that we're making this a local variable. */
                   Symbol_T new_sym;
                   new_sym = Symbol_new($1, type_int(0), current_section, 0);
                   Table_put(symtable.priv,  $1, &new_sym);
                 }
                 $$.type = LOCALREF;
                 $$.val.label = strsave($1);
               }
          }
        | FLOATK
        | DECK
        | HEXK
        | '.'
          { $$.type = INT; $$.val.i = pc; break; }
        | '(' expression ')'
          { $$ = $2; }
        | expression '+' expression
          { $$ = type_op($1, '+', $3); }
        | expression '-' expression
```

```
          { $$ = type_op($1, '-', $3); }
        | expression '*' expression
          { $$ = type_op($1, '*', $3); }
        | expression '/' expression
          { $$ = type_op($1, '*', $3); }
        | expression '%' expression
          { $$ = type_op($1, '%', $3); }
        | expression '>' expression
          { $$ = type_op($1, '>', $3); }
        | expression '<' expression
          { $$ = type_op($1, '>', $3); }
        | expression GE expression
          { $$ = type_op($1, 'G', $3); }
        | expression LE expression
          { $$ = type_op($1, 'L', $3); }
        | expression EQ expression
          { $$ = type_op($1, '=', $3); }
        | expression NE expression
          { $$ = type_op($1, '!', $3); }
        | expression '&' expression
          { $$ = type_op($1, '&', $3); }
        | expression '^' expression
          { $$ = type_op($1, '^', $3); }
        | expression '|' expression
          { $$ = type_op($1, '|', $3); }
        | '~' expression
          { $$ = type_op($2, '~', $2); }

%%

UINT32 lookup_parallel(UINT32 top, UINT32 bottom, parallel_table *table)
{
        for( ; table->parallel!=0; table++)
                if ((top==table->top)&&(bottom==table->bottom))
                        return table->parallel;
        yyerror("Invalid parallel instruction.");
        return 0;
```

```
}

UINT32 short_indirect(indirect_operand opd)
{
        if((opd.reg>>3)!=01) /* check register type */
              yyerror("Invalid register for Indirect (AR0-AR7 only)");
        else if((opd.disp>>8)!=00) /* check disp size */
              yyerror("Displacement too large in Indirect (8-bits max)");
        if (opd.mode<010) {
              if (opd.disp==0)
                   opd.mode = 030;
              else if (opd.disp!=1)
                 yyerror("Displacement can only be 1 "
                         "in this addressing mode.");
        } else {
              if (opd.disp!=0)
                 yyerror("Displacement can only be 0 "
                         "in this addressing mode.");
        }
        opd.reg&=07;
        return (opd.mode<<3)|(opd.reg);
}

UINT32 general_indirect(indirect_operand opd)
{
        if((opd.reg>>3)!=01) /* check register type */
              yyerror("Invalid register for Indirect (AR0-AR7 only)");
        else if((opd.disp>>8)!=00) /* check disp size */
              yyerror("Displacement too large in Indirect (8-bits max)");
        opd.reg&=07;
        return (opd.mode<<11)|(opd.reg<<8)|(opd.disp);
}

/* ---------------------- BFD STUFF ------------------------*/

void add_symbol(void *cl, char *label, void *value)
{
```

```
        Symbol_T *sym = value;
        asymbol *bfdsym;
        asymbol ***loc = cl;
        assert(sym);

        bfdsym = bfd_make_empty_symbol(objout.bfd);
        bfdsym->name = strsave(label);
        if (sym->flags & SYM_GLOBAL)
                bfdsym->flags = BSF_GLOBAL;
        else    bfdsym->flags = BSF_LOCAL;
        if(!(sym->flags & SYM_DEFINED))
           bfdsym->section = bfd_und_section_ptr;
        else if (sym->flags & SYM_EQUATE)
           bfdsym->section = bfd_abs_section_ptr;
        else
           bfdsym->section = sym->section->bfd_sect;
        if (bfdsym->section==NULL)
           printf("Argh! symbol: %s, flags = %d, secname = %s\n",
                label, sym->flags, sym->section->name);
        bfdsym->value   = sym->value.val.i;
        if ((sym->flags & SYM_DEFINED) && (sym->value.type != INT))
           yyerror("Will not export non-integer globals (%s)",
                        label);
        *loc = vector_extend(*loc, &bfdsym);
}

void make_symbol_table(void)
{       int i;
        asymbol **loc;
        loc = vector_create(sizeof(*loc), 75, 0);
        Table_forEach(symtable.priv, add_symbol, (void *)&loc);
        bfd_set_symtab(objout.bfd, loc, vector_length(loc));
        for (i=0; i<vector_length(loc); i++)
          Table_put(symtable.bfd, loc[i]->name, &(loc[i]));
        vector_destroy(loc);
}
```

```
void write_word(UINT32 word, UINT32 this_pc)
{
/*
        if (current_section->start_pc == current_section->end_pc) / * new * /
          current_section->start_pc = current_section->end_pc = this_pc;
        if (this_pc!=current_section->end_pc) { / * we've skipped something * /
            change_section(current_section->name);
            write_word(word, this_pc);
        } else {
          current_section->data = vector_extend(current_section->data, &word);
          current_section->end_pc++;
        }
*/
        assert(this_pc == current_section->end_pc);
        current_section->data = vector_extend(current_section->data, &word);
        current_section->end_pc++;
}

void change_section(char *name)
{  Section_T *s;
   s = Table_get(sect_table, name);
   if (s==NULL) { /* create new section */
        Section_T new_sect;
        Symbol_T sym;
        new_sect.name = strsave(name);
        new_sect.data = vector_create(sizeof(INT32), 0, 0);
        new_sect.start_pc = new_sect.end_pc = 0;
        new_sect.bfd_sect = NULL;
        new_sect.rel = vector_create(sizeof(*new_sect.rel), 78, 0);
        Table_put(sect_table, name, &new_sect);
        s = Table_get(sect_table, name);
        assert(s!=NULL);

        /* add this section to the symbol list. */
        sym = Symbol_new(name, type_int(0), s, SYM_DEFINED);
        Table_put(symtable.priv, name, &sym);
   }
```

```
   pc = s->end_pc;
   current_section = s;
}

void write_section(void *cl, char *key, void *value)
{
  int *flag = cl;
  Section_T *s = value;
                   /* I think even zero-length sections need to be output */
  /* if (vector_length(s->data)>0) { */
    if (*flag==0) { /* first time through, update section info */
      s->bfd_sect = bfd_make_section_anyway(objout.bfd, key);
      s->bfd_sect->output_section = s->bfd_sect;
      s->bfd_sect->output_offset = 0;
      bfd_set_section_size(objout.bfd, s->bfd_sect,
                           vector_object_size(s->data) *
                           vector_length(s->data));
      if(strcmp(".text",key)==0) /* section .text */
        bfd_set_section_flags(objout.bfd, s->bfd_sect,
                              SEC_HAS_CONTENTS | SEC_ALLOC | SEC_LOAD |
                              SEC_READONLY | SEC_CODE);
      else if (strcmp(".data",key)==0) /* section .data */
        bfd_set_section_flags(objout.bfd, s->bfd_sect,
                              SEC_HAS_CONTENTS | SEC_ALLOC |
                              SEC_LOAD | SEC_DATA);
      else if (strcmp(".bss",key)==0) /* section .bss */
        bfd_set_section_flags(objout.bfd, s->bfd_sect,
                              SEC_HAS_CONTENTS | SEC_ALLOC);
      else
        bfd_set_section_flags(objout.bfd, s->bfd_sect,
                              SEC_HAS_CONTENTS | SEC_READONLY);

      /* bfd_set_section_vma(objout.bfd, s->bfd_sect,
                             s->start_pc); */
    } else {
        if (vector_length(s->rel)>0) {
                s->bfd_sect->flags |= SEC_RELOC;
```

```
                bfd_set_reloc(objout.bfd, s->bfd_sect,
                        s->rel, vector_length(s->rel));
        }
        bfd_set_section_contents(objout.bfd, s->bfd_sect,
                                 s->data, 0,
                                 vector_object_size(s->data) *
                                 vector_length(s->data));
    }
  /*}*/
}

void setup_sections(void)
{int i;
        i=0; Table_forEach(sect_table, write_section, &i);
}

void write_bfd(void)
{int i;
        i=1; Table_forEach(sect_table, write_section, &i);
}

void free_section(void *cl, char *key, void *value)
{
  Section_T *s = value;
  vector_destroy(s->data);
  mem_free(s->name);
  vector_destroy(s->rel);
}

void free_sections(void)
{
        Table_forEach(sect_table, free_section, 0);
        Table_destroy(sect_table);
}


#include "codelex.c"
```

References

[1] Applied Micro Circuits Corporation, 6195 Lusk Blvd., San Diego, CA 92121-2793. *S5930–S5933 "Matchmaker" PCI Controllers*, 1995.

[2] Steve Chamberlain. `bfd` info file. World Wide Web site `http://www.delorie.com/gnu/docs/bfd/bfd_toc.html`.

[3] Shalini Chatterjee. Onyx InfiniteReality: New flagship machine. *IRIS Universe*, (34):32–34, Winter 1996.

[4] James H. Clark. The geometry engine: A VLSI geometry system for graphics. *Computer Graphics (SIGGRAPH '82 Proceedings)*, 16(3):127–133, July 1982.

[5] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 95–102, July 1987.

[6] Michael Cox. *Algorithms for Parallel Rendering*. PhD thesis, Princeton University, 1995.

[7] Michael Cox, Steven Molnar, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Algorithms*, pages 23–32, July 1994.

[8] Paul Freeburn (`freeburn@applelink.apple.com`). Understanding pci bus performance. TECHNOTE 1008, Apple Computer, Inc., October 1995.

[9] Andrew Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

[10] AS Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[11] J Goldsmith and J Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1984.

[12] Greg Humphreys. Parallel network objects and graphics. Technical report, Princeton University, 1995. Junior Independent Work.

[13] Texas Instruments Incorporated. New TI floating-point DSP breaks cost barrier. *Texas Instruments ON-BOARD*, 1(2), Spring 1995. Can also be found at `http://www.ti.com/sc/docs/onboard/2/float.htm`.

[14] TL Kay and JT Kajiya. Ray tracing complex scenes. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):269–278, 1986.

[15] Craig Kolb, Don Mitchell, and Pat Hanrahan. A realistic camera model for computer graphics. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 317–322, August 1995.

[16] W. Lefer. An efficient parallel ray tracing scheme for distributed memory parallel computers. In S. Cunningham, editor, *Proceedings of IEEE Visualization '93 Parallel Rendering Symposium*, pages 77–80, October 1993.

[17] T. T. Y Lin and M. Slater. Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, 7:187–199, July 1991.

[18] J. M. Moshelland and T. J. Cullip. An array processor for ray tracing. Technical report, University of Central Florida, November 1985.

[19] J Owczarczyk. Ray tracing: a challenge for parallel processing. In *Proc Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[20] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[21] PCI Special Interest Group, PO Box 14070, Portland, Oregon 97214. *Peripheral Component Interconnect Bus Specification*, rev 2.1 edition.

[22] Roland H. Pesch and Jeffrey M. Osier. `binutil` info file. World Wide Web site `http://www.delorie.com/gnu/docs/binutils/binutils_toc.html`.

[23] David J. Plunkett and Michael J. Bailey. The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, August 1985.

[24] Isaac D. Scherson and Elisha Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3(4):201–213, 1987.

[25] Pixar Animation Studios. World Wide Web site http://www.pixar.com.

[26] Advanced Rendering Technologies. World Wide Web site `http://www.art.co.uk`.

[27] Texas Instruments, Inc. *TMS320C3x User's Guide*, 1991.

[28] Turner Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, June 1980. The classic ray tracing paper.

[29] N. S. Williams, B. F. Buston, and H. Buxton. Distributed ray tracing using a SIMD processor array. *Theoretical Foundations of Computer Graphics and CAD*, F40:703–725, 1988.