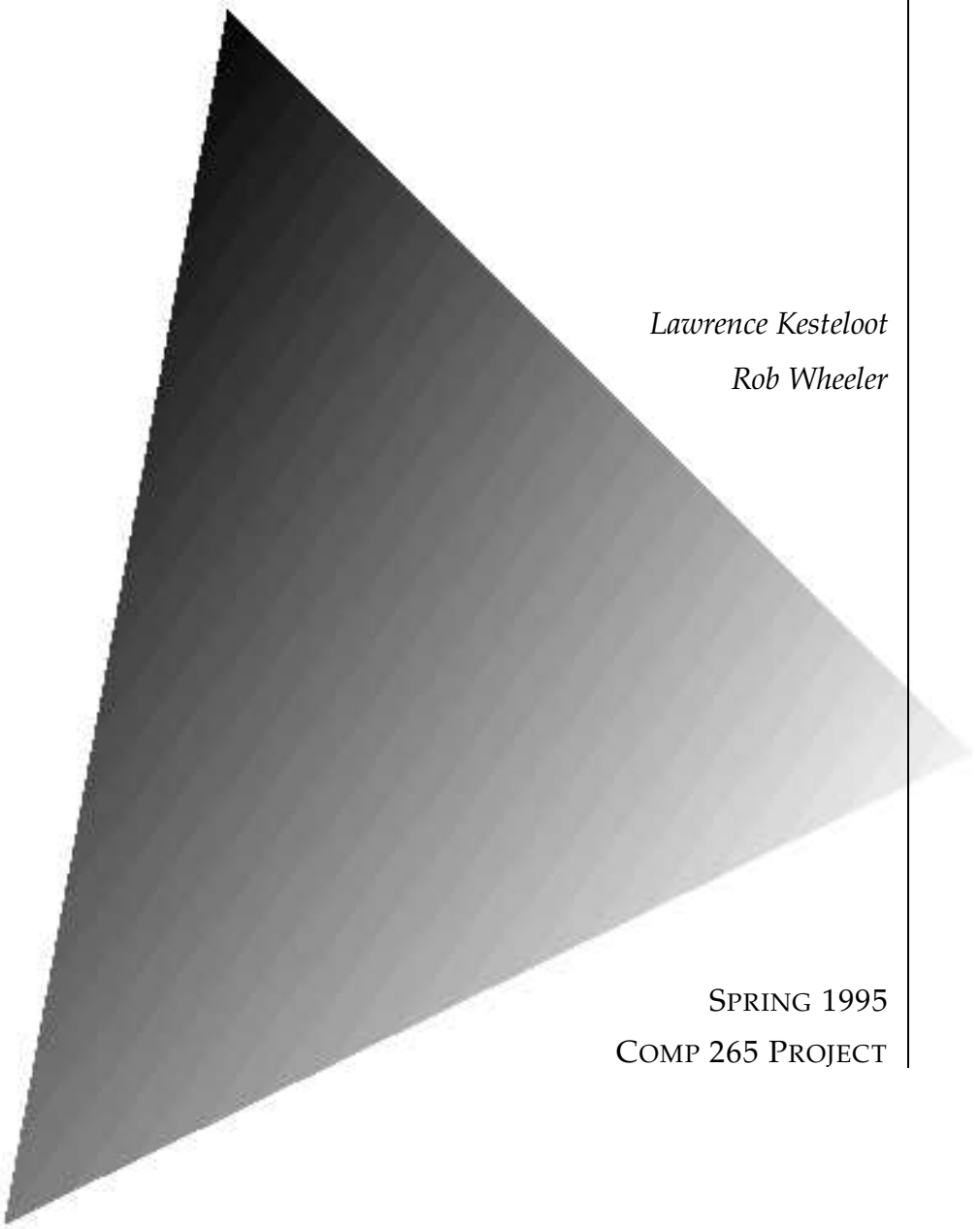
GRAPHICS ENGINE



Lawrence Kesteloot

Rob Wheeler

SPRING 1995

COMP 265 PROJECT

The cover triangle was rasterized by the Graphics Engine's simulator and the sample application provided in the appendix.

Contents

List of Figures

List of Tables

1 System Overview

1.1 Introduction

This document outlines the design of the Graphics Engine (GE), which has an optimized instruction set for transforming and rasterizing 3-D graphics primitives for low-end machines. The GE typically is included in home or arcade video game machines. A general-purpose CPU and the GE share memory for communication: The CPU calculates the game dynamics and sets up a graphics data structure (a display list of 3-D primitives) that the GE traverses and rasterizes in a pipeline arrangement. The GE has a RISC core with a back-end specialized for rasterizing triangles.

A typical application is a game that uses 3-D graphics along with sound and 2-D sprites (the latter two being supplied by other hardware). The application uses the GE to transform, light, and shade the generated geometry. A typical game might require up to 3,000 triangles per scene to achieve rendering detail comparable to current 2-D games. Thus, at 30 frames a second, roughly 100,000 triangles per second must be transformed and half of these must be lit and rasterized. (Over half of the triangles will be removed by backface and frustum culling.)

The GE instruction set is optimized for the following tasks:

1. traversing a hierarchical display list;
2. transforming coordinates;
3. calculating plane equations; and
4. rasterizing polygons.

Only minimal 3-D features are supported—clipping, diffuse lighting, gouraud shading, screen-door transparency, and Z-buffering. The GE has no anti-aliasing, texturing, or blending support.

This is a very specialized CPU, not only because it is optimized for graphics, but because it is designed for low-end, inexpensive graphics. It does not provide the flexibility of high-end graphics workstations, and the game programmer is expected to abide by restrictions, such as keeping coordinates within the

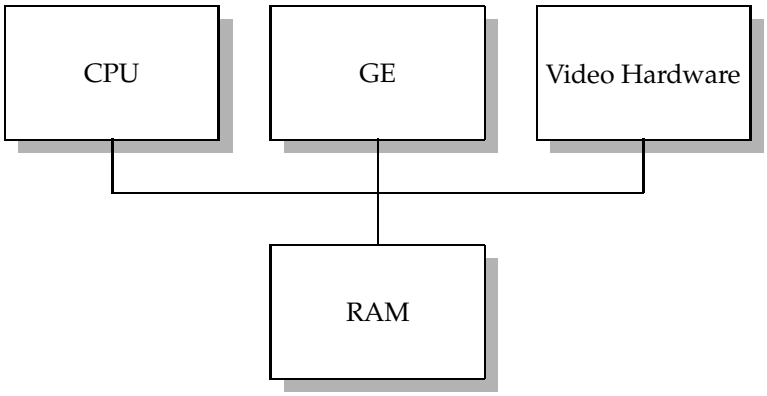


Figure 1.1: Typical machine configuration

range of fixed-point values. When necessary, we simplified the hardware at the expense of a heavier burden on the programmer. Most game programmers are used to programming in assembler and having no programming support whatsoever, so these are not unusual nor unexpected demands.

A typical machine configuration is shown in Figure 1.1. The CPU and the GE will communication via display lists stored in shared memory. The CPU will setup the display list for the next frame while the GE will traverse and rasterize the display list for the current frame. Semaphores in memory will keep the producer-consumer pipeline synchronized.

A note about this architecture document: In a very low-end high-performance architecture, the application, the processor architecture, and the implementation are more interdependent than a pure, transparent architecture will allow. On the GE in particular, we not only know what *kind* of application will run on it, but we know the actual application! (Although game writers are free to rewrite the program running on the GE, most are likely to use the one supplied in the Appendix of this document.) As a result, this document describes the application, the architecture, and touches on the implementation. In a sense, the application can be seen as part of the architecture, with the display list being the architecture's machine language and the on-chip program being the microcode. In any case, the close synergy between the three parts of the Graphics Engine allow for a design that optimizes for both performance and economy.

1.2 Application Background

In a typical graphics pipeline, geometry (usually triangles) is generated in 3-D space. The vertices are transformed by a model-view matrix which converts the primitives from object space to world space and from world space to camera

space in one step. The primitives are then lit in world space, transformed by a projection matrix, clipped to the viewing frustum, projected onto a 2-D screen, and rasterized (drawn). See [2] for a more complete description of the pipeline.

As is often done in interactive graphics, our pipeline makes some approximations to reduce the computational burnen on the system. For example, if we assume that the viewer and the light sources are infinitely far away, then we can keep a combined model-view-projection matrix, saving one matrix multiplication. The normals do not need to be transformed by the inverse transpose of the model-view matrix (as would normally be correct); instead, we can transform the light source and view vectors by the inverse of the model-view matrix, yielding the same results and saving a matrix multiplication per vertex.

The rasterization entails finding the area on the screen which is covered by the primitive and interpolating the information at the vertices (in our case, color and Z). We use the *plane-equation* method of rasterization. The idea is that each edge of a triangle can be described by a equation of the form $Ax + By + C = 0$. Substituting the coordinates of each pixel in the bounding box of the triangle into all three edge equations will determine if the pixel is inside or outside the triangle. Those that are inside can interpolate parameters using more plane equations of the form $p = Ax + By + C$, where p is the parameter to be interpolated. This method is easier to implement than traditional edge-walkers, is more easily parallelized, and is often faster for small triangles[1].

In order for primitives to occlude other primitives that are behind them, we keep track of the distance to the closest primitive at each pixel. This *Z-buffer* is initialized to infinity and for each pixel about to be drawn the Z value in the Z-buffer is compared to the primitive's Z value at that pixel, and the closest pixel (the one with the lowest Z value) is placed in the color and Z buffers. The Graphics Engine supports a Z-buffer with 16 bits per pixel, which is certainly enough for game applications and probably enough for most applications as well[2].

Since we use hierarchical display lists, a model-view matrix stack is kept that allows sub-lists' transformations to remain localized. Although in our implementation only diffuse lighting is implemented, the architecture of the Graphics Engine would support specular (Phong) lighting quite well.

Screen-door transparency is a method to simulate transparent surfaces by only drawing a subset of the primitive's pixels, thus allowing the background to show through. The GE supports basic screen-door transparency by allowing a bitmask for each row of 16-pixel inside the primitive.

1.3 System Requirements

[1] Although the inside loops of edge-walkers are tighter than those of plane-equation algorithms, the per-primitive and per-scanline overhead is not sufficiently amortized on small triangles.

[2] Although most systems use from 24 to 32 bits of Z per pixel, much of that resolution (probably around 8 bits) is lost because of the exponential distribution of Z values as the yon plane is approached.

1.3.1 Transformation/Rasterization Rate

If our goal is to draw 100,000 meshed triangled per second, and if we assume that a simple RISC architecture can easily be run at a 100 MHz clock rate, then we must transform, light, clip, and rasterize a triangle in 1000 clock cycles (or almost equivalently, instructions). If we consider an "average" triangle be cover 50 pixels, then a general-purpose RISC architecture could spend its 1000 instructions on the rasterization alone! Clearly, special-purpose instructions and hardware are needed to achieve our goal.

A more detailed outline of the operations performed per triangle will show which instructions need attention:

1. Copy vertex information into buffer.
2. Transform vertex by model-view-projection matrix.
3. Light the vertex.
4. Clip to hither and yon planes.
5. Do perspective divide.
6. Compute screen-space bounding box.
7. Compute edge equations (3).
8. Compute parameter equations (4).
9. Loop over bounding box and rasterize.

Transforming a vertex is an expensive operations, with 16 multiplications and 12 additions. A special-purpose dot-product operation is provided to reduce the time to the equivalent of 4 multiplications and 8 additions. (Equally significant is that the dot-product operation works directly in data memory, avoiding the 16 loads that would be required to load the matrix.) Lighting also requires a few dot products, and the same dot-product operation can be used to speed up this part of the pipeline.

Clipping is straightforward and inexpensive (since we only have two planes to clip to), and normal arithmetic operations can be used. The perspective divide requires a reciprocal, and since a reciprocal can be found almost as fast in software as in hardware, a reciprocal operation is not provided.

Computing the edge and parameter equations requires heavy use of fixed-point data types, and a fixed-point multiply is provided (in addition to the integer multiply) to avoid the considerable extra work that would be required to simulate such an operation in software.

All of the gains of the above operations, however, could easily be shadowed by the expensive rasterization loop. This is where special-purpose instructions and hardware are most necessary. We use a linear-expression tree to allow 16 instances of the linear equations to be evaluated in 4 clock cycles. This allows

16 pixels to be tested against the edge equations and have their parameters interpolated in about 13 clock cycles. An average triangle (50 pixels) can be rasterized in about 400 clock cycles this way, allowing us to reach our goal.

1.3.2 Shared Memory Size

Video memory. Video memory is *double-buffered*. Since the output device of this machine is likely to be an NTSC television or a monitor of comparable quality (in video arcades), a frame resolution of at most 640×480 is required. Such a low resolution will increase the effective rasterization speed since triangles will be smaller. A colormap system is not acceptable since we need to interpolate color for lighting[3], so an RGB scheme with 5 bits for red and blue and 6 bits for green is used[4]. A *Z-buffer* is used for hidden surface removal, with 16 bits per pixel. The total video memory requirement is then 3 buffers $\times 640$ X-resolution $\times$ 480 Y-resolution $\times$ 2 bytes/pixel $= 1,843,200$ bytes.

Display list memory. Each vertex needs to store three coordinates (x, y, z) of 16 bits each (6 bytes), a vertex number, and the vertex op-code. We expect to rasterize about 3,000 triangles per frame, with many of those in mesh format (one vertex per triangle). This will take roughly 6000×8 bytes plus other information, such as transformation matrices (which are few). The total display list memory requirement is then at most 100,000 bytes per display list, times two since display lists are double-buffered[5]. The GE will access all memory in big-endian order, so it would be most efficient if the CPU's endianess were the same.

Program memory. The program running on the GE will be on-chip. Programs running on the CPU will probably be at most be 100,000 bytes, plus 200,000 bytes for data.

Total. The total external memory requirements are then $1.8 + 0.2 + 0.3 = 2.3$ megabytes. This probably will be rounded up to 4 megabytes, requiring at least 22 bits in address fields.

[3] Internally, all colors are kept in RGB format. The architecture does not restrict the implementation from dithering the final RGB value and using a colormap system in the frame buffer.

[4] The extra bit is given to green because the eye is most sensitive to the green part of the spectrum. An extra bit will allow more subtle shades of intensity.

[5] See Section 8.1.1 for a discussion of how to single-buffer most of the display list.

2 Graphics Engine Architecture

The Graphics Engine is a Harvard architecture chip with 1024 32-bit words of on-chip program memory, 1024 words of on-chip data memory, access to external RAM (through DMA, or *Direct Memory Access*), 32 word-length registers, one program counter, one status word, and a RISC instruction set specialized for graphics processing. It has no interrupts, exceptions, no memory management unit, and no cache. See Figure 2.1 for the basic programming model.

Operation:

```
typedef unsigned long Word;
```

2.1 Registers

The GE has 32 general-purpose registers that are word-length (32 bits). Reading register 0 always returns 0 and writing to it has no effect.

Operation:

```
Word reg[32];
```

2.2 Address Space

Memory locations 0 through 1023 address words in the on-chip data memory. External RAM is read and written through special DMA instructions that copy blocks of memory in and out of internal data memory. Video memory is accessed through special rasterization instructions. There is no direct access to the program counter or the status word.

Operation:

```
Word iMem[1024], dMem[1024], eMem[1 << 22];
```

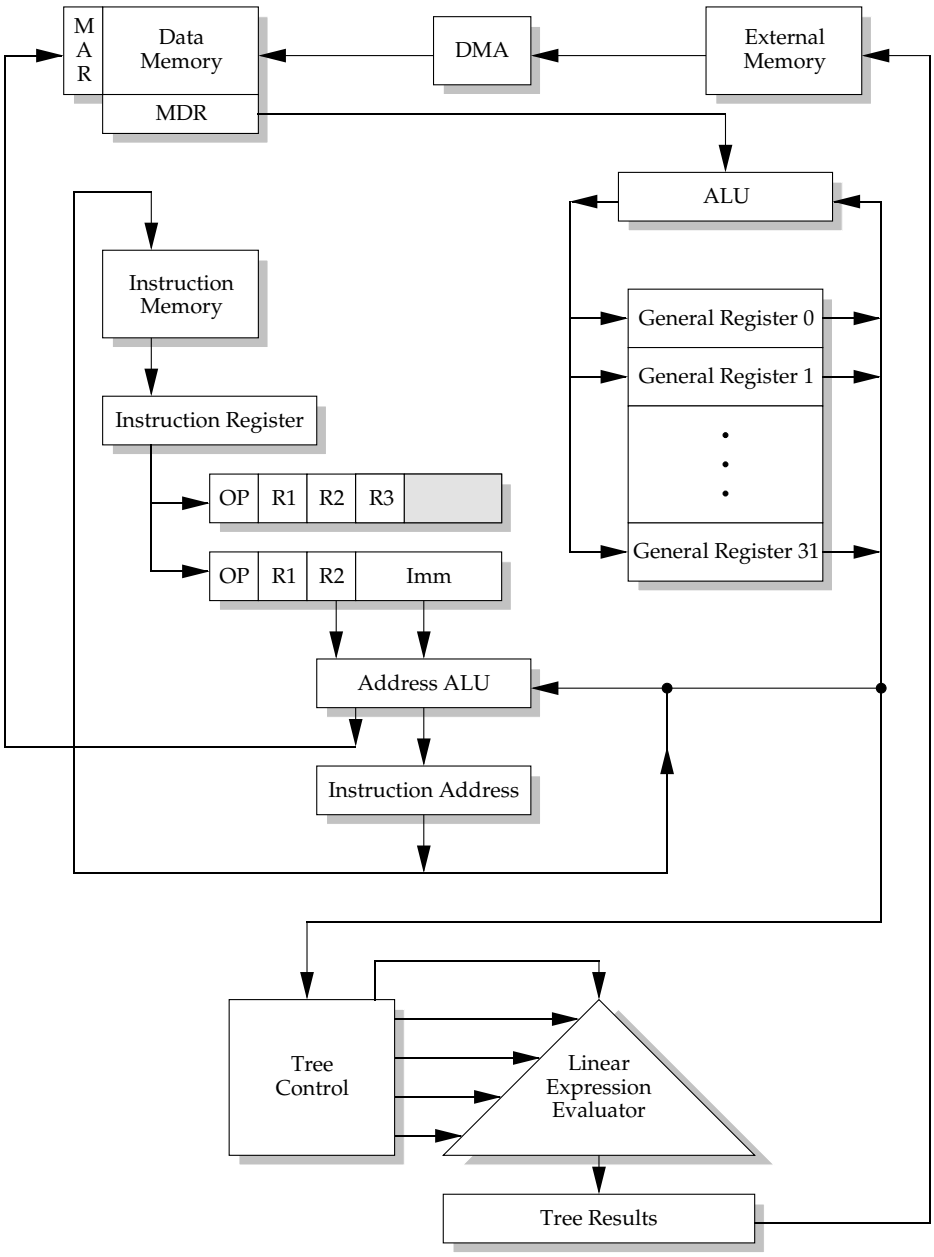


Figure 2.1: Basic Programming Model

2.3 Status Word

The status word is not bit addressable, but its contents can be tested with various branching instructions.

BIT 0 — ZERO (Z)

If 0 the last arithmetic or logical scalar or vector instruction returned a non-zero number.

If 1 otherwise.

BIT 1 — NEGATIVE (N)

Set to the most significant bit of the result of the last arithmetic or logical scalar or vector operation.

The "Operation" sections of the instruction descriptions in Chapter 3 use the following function to set the status word after arithmetic and logical operations:

Operation:

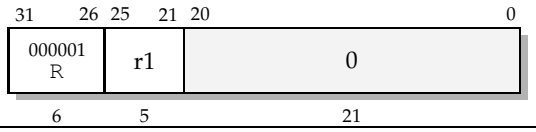
```
#define Z 1
#define N 2
Word sw;
SetStatusWord (Word val) {
    nbit = (val >> 31);
    zbit = (val == 0);
    sw = zbit | (nbit << 1);
}
```

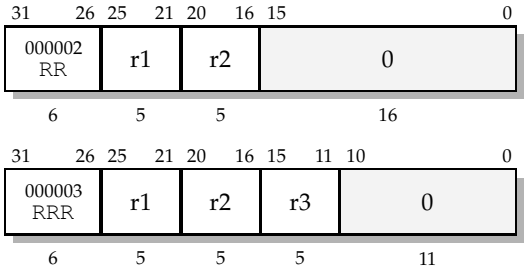
2.4 Instruction Format

There are only two basic instruction formats: RRR and RRI. The four others are subsets of one of these. Six bits are reserved for the op-code, with the rest reserved for register operands and immediate values.

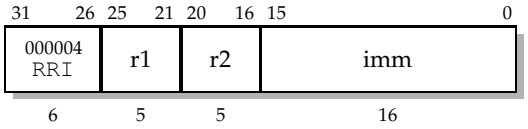
The OP format only has the op-code. It is useful for tree instructions that need no operands.

The R, RR, and RRR instructions deal with registers directly. They are used for arithmetic and logical operations, as well as tree operations. The R format is not used in the current instruction set.

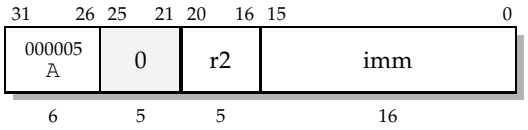




The RRI format is used both for the immediate logical instructions as well as for loads and stores. For the former, the immediate value determines the number of bits to shift, and for the latter, the second register and the immediate determine the address to access.



Finally, the A format is used for branches and jumps. Only the second register and the immediate are used to determine the branch destination. The immediate, being 16 bits, has enough range to cover the internal instruction space, but the register can optionally be used for jump tables. Notice that the assembly language format for the `LD` and `ST` instructions are reversed: The destination is always first, even if in the encoding the destination is second (as in the `ST` instruction).



Operation:

```
Word pc, inst, op, r1, r2, r3, imm;
inst = iMem[pc++];
op = inst >> 26;
r1 = (inst >> 21) & 0x31;
r2 = (inst >> 16) & 0x31;
r3 = (inst >> 11) & 0x31;
/* Sign-extend immediate: */
imm = (signed long)(signed short)(inst & 0xFFFF);
```

2.5 Addressing Modes

Only the load, store, branch, and jump instructions need to reference memory. The first two address internal data memory and the last two address instruction memory. All four instruction types add the second register specified

in the instruction word with the immediate value to determine the address. Register `R0`, which is always zero, can be used if the immediate value directly specifies the address.

Notice that the `LI` (Load Immediate) instruction has the same format as the above four instruction types, but the sum of the second register and the immediate is placed directly into the first register without referencing memory, so it is not strictly an addressing mode since the resulting value is not checked against the memory bounds.

All addresses reference words. There are no addressing modes for half-words or bytes.

Operation:

```
Word addr;
addr = reg[r2] + imm;
```

2.6 Rasterization Registers

The rasterization back-end has several registers that can be set with the `SCRSET` and `TREESET` instructions.

Operation:

```
Word width, height;     /* Size of the screen */
Word fbAddr, zAddr;     /* Location in bytes of buffers */
Word xTree, yTree;      /* Position of left-most pixel of tree */
```

2.7 Instruction Sequencing

The instruction set has one unconditional jump and conditional branches for the following conditions: equal to, not equal to, less than, less than or equal to, greater than, and greater than or equal to. The implementation can assume that these conditional branches are not likely to be taken. A "likely" version of each branch instruction is available that allows the implementation to fill the instruction pipeline from the destination address. All jump and branch instructions have their destination address specified by the sum of a register-immediate pair. The register can be `R0` (which is always zero) for absolute addresses.

There is a single branch delay slot which is always executed, whether or not the branch is taken. This slot may not be filled with another branch, jump, or call instruction, although this is not policed and the results of doing so are undefined.

The processor has no mechanism for halting or termination. The application is a loop that continuously reads data from the main CPU and rasterizes it.

2.8 External Synchronization

The two DMA instructions `READ` and `WRITE` can read or write 16 words to external memory. Since the main CPU and the graphics engine are set up in a typical producer-consumer relationship, two semaphores (one each way) need to be kept. The two DMA instructions can be used for this purpose since there are no race conditions between the processors.

The `READ` instruction is primarily used to read the display lists. The `WRITE` instruction can be used to modify the frame buffer or to return debugging and performance information to the CPU.

2.9 Policing

Policing is useful both to facilitate compatibility with future architectures and to help find bugs. It is difficult to police an architecture that has no interrupts or exceptions, so this architecture tries to at least achieve the first advantage of policing: A violation halts the processor. This encourages programmers to use valid opcodes and to keep memory references in range. An exception would have made debugging easier at the cost of considerable architecture and implementation complications. (Consider what happens in the instruction pipeline when an instruction needs to be restarted after an exception.) In addition, programmers can use the simulator to find bugs.

The following violations halt the processor: Invalid op-code; invalid program counter; invalid data memory reference; invalid tree setup operands; and DMA access that reads or writes outside of data memory.

3 Instruction Set

Instructions are in the general format `OP dest,src1,src2`. The operands are almost always registers, and the destination register is allowed to be the same as one of the source registers. Almost all operations set the status bits based on the result of the operation. Note that the bits are set before the result is placed in the register. This means that using `R0` (which is always zero) as a destination still sets the status bits according to the result of the operation. Table 3.1 on page 38 summarizes the instruction set.

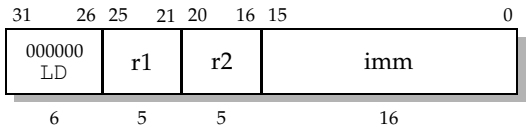
3.1 Register Loading and Storing

LD

LOAD REGISTER

Format:

```
LD r1, r2(imm)
```



Description:

The word in data memory at location `r2(imm)` is loaded into register `r1`.

Operation:

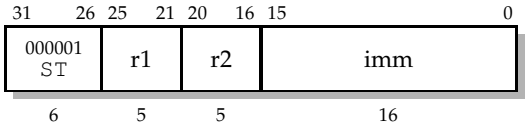
```
reg[r1] = dMem[imm + reg[r2]];
SetStatusWord (reg[r1]);
```

ST

SAVE REGISTER

Format:

```
ST r2(imm), r1
```



Description:

The word in register `r1` is stored in data memory at location `r2(imm)`. Note that although the assembly operands are the reverse of the `LD` instruction, the instruction encoding does not change.

Operation:

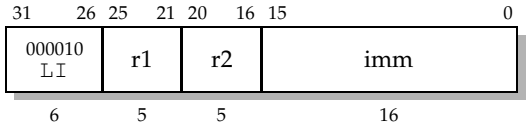
```
dMem[imm + reg[r2]] = reg[r1];
```

LI

LOAD IMMEDIATE

Format:

```
LI r1, r2(imm)
```



Description:

The 16-bit value `r2(imm)` is loaded into `r1`. The upper 16 bits of `imm` are sign-extended. (I.e., all upper 16 bits are set to bit 15.)

Operation:

```
reg[r1] = (signed long)(signed short)(imm + reg[r2]);
SetStatusWord (reg[r1]);
```

3.2 General Purpose Arithmetic and Logic

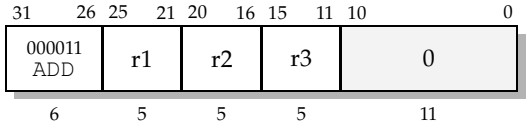
3.2.1 Arithmetic

ADD

FIXED-POINT ADD

Format:

```
ADD r1, r2, r3
```



Description:

Registers `r2` and `r3` are added together and the sum is placed in `r1`.

Operation:

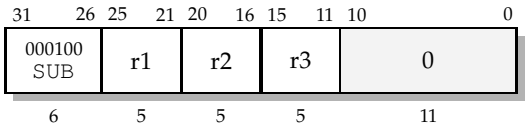
```
reg[r1] = reg[r2] + reg[r3];
SetStatusWord (reg[r1]);
```

SUB

FIXED-POINT SUBTRACT

Format:

```
SUB r1, r2, r3
```



Description:

Register `r3` is subtracted from register `r3` and the difference is placed in `r1`. If `r0` is used as the destination, then this instruction is effectively a `CMP` (compare) instruction. If `r0` is used for `r2`, then this instruction is effectively a `NEG` (negate) instruction.

Operation:

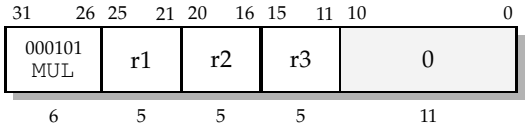
```
reg[r1] = reg[r2] - reg[r3];
SetStatusWord (reg[r1]);
```

MUL

INTEGER SIGNED MULTIPLICATION

Format:

```
MUL r1, r2, r3
```



Description:

Registers `r2` and `r3` are multiplied together and the product is placed in `r1`. The multiplication assumes that both operands are signed in two's-complement notation, and the result is also signed in two's-complement notation. Overflow is ignored.

Operation:

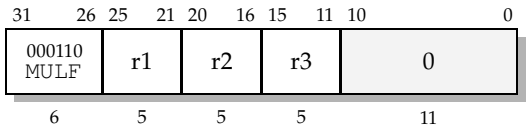
```
reg[r1] = (signed long)reg[r2] * (signed long)reg[r3];
SetStatusWord (reg[r1]);
```

MULF

FIXED-POINT SIGNED MULTIPLICATION

Format:

```
MULF r1, r2, r3
```



Description:

Registers `r2` and `r3` are multiplied together and the product is placed in `r1`. The multiplication assumes that both operands are signed in two's complement notation, and the result is also signed in two's complement notation. Operands and product are in `s15.16` fixed-point format. (Actually, it really means that one operand is in `s15.16` and the other operand is anything. The result is in the same format as the other operand.) Overflow is ignored.

Operation:

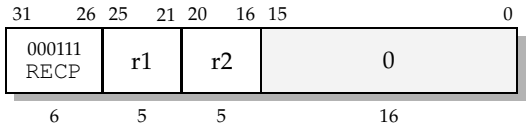
```
reg[r1] = ((signed long)reg[r2] * (signed long)reg[r3]) >> 16;
SetStatusWord (reg[r1]);
```

RECP

FIXED-POINT SIGNED RECIPROCAL

Format:

```
RECP r1, r2
```



Description:

The reciprocal of register `r2` is placed in `r1`. Both operands and result are in two's complement notation. The input is assumed to be an integer (in `s31.0` format) and the result is in `s.31` fixed-point format.

Operation:

```
reg[r1] = 0x7FFFFFFF / (signed long)reg[r2];
SetStatusWord (reg[r1]);
```

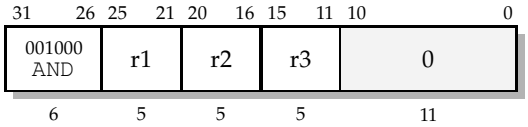
3.2.2 Logic and Bitwise

AND

BIT-WISE AND

Format:

```
AND r1, r2, r3
```



Description:

Registers `r2` and `r3` and bit-wise ANDed and the result is placed in `r1`.

Operation:

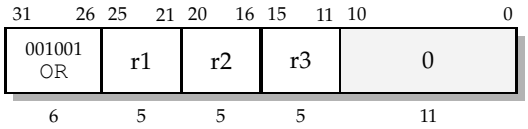
```
reg[r1] = reg[r2] & reg[r3];
SetStatusWord (reg[r1]);
```

OR

BIT-WISE OR

Format:

```
OR r1, r2, r3
```



Description:

Registers `r2` and `r3` and bit-wise ORed and the result is placed in `r1`.

Operation:

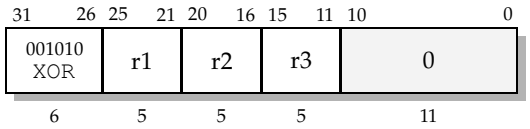
```
reg[r1] = reg[r2] | reg[r3];
SetStatusWord (reg[r1]);
```

XOR

BIT-WISE XOR

Format:

```
XOR r1, r2, r3
```



Description:

Registers `r2` and `r3` and bit-wise XORed and the result is placed in `r1`.

Operation:

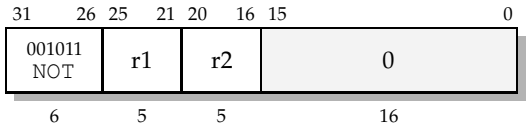
```
reg[r1] = reg[r2] ^ reg[r3];
SetStatusWord (reg[r1]);
```

NOT

BIT-WISE NOT

Format:

```
NOT r1, r2
```



Description:

The bit-wise inversion of register `r2` is placed in `r1`.

Operation:

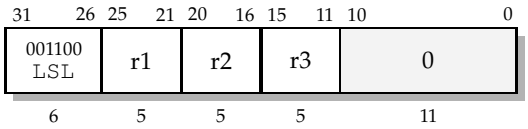
```
reg[r1] = ~reg[r2];
SetStatusWord (reg[r1]);
```

LSL

LOGICAL SHIFT LEFT

Format:

```
LSL r1, r2, r3
```



Description:

Register `r2` is shifted left `r3` bits and the result is placed in `r1`.

Operation:

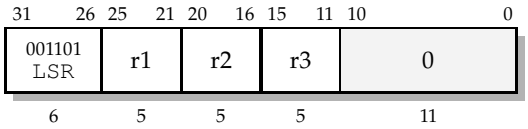
```
reg[r1] = reg[r2] << reg[r3];
SetStatusWord (reg[r1]);
```

LSR

LOGICAL SHIFT RIGHT

Format:

```
LSR r1, r2, r3
```



Description:

Register `r2` is shifted right `r3` bits and the result is placed in `r1`. The bits shifted in from the left are zeros.

Operation:

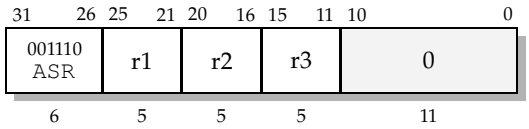
```
reg[r1] = reg[r2] >> reg[r3];
SetStatusWord (reg[r1]);
```

ASR

ARITHMETIC SHIFT RIGHT

Format:

```
ASR r1, r2, r3
```



Description:

Register `r2` is shifted right `r3` bits and the result is placed in `r1`. The bits shifted in from the left are the same as the sign bit.

Operation:

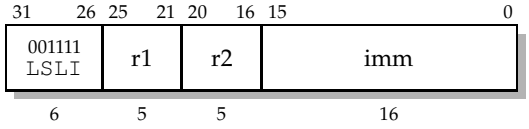
```
reg[r1] = (signed)reg[r2] >> reg[r3];
SetStatusWord (reg[r1]);
```

LSLI

LOGICAL SHIFT LEFT IMMEDIATE

Format:

```
LSLI r1, r2, imm
```



Description:

Register `r2` is shifted left `imm` bits and the result is placed in `r1`.

Operation:

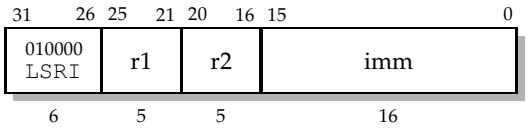
```
reg[r1] = reg[r2] << imm;
SetStatusWord (reg[r1]);
```

LSRI

LOGICAL SHIFT RIGHT IMMEDIATE

Format:

```
LSRI r1, r2, imm
```



Description:

Register `r2` is shifted right `imm` bits and the result is placed in `r1`. The bits shifted in from the left are zeros.

Operation:

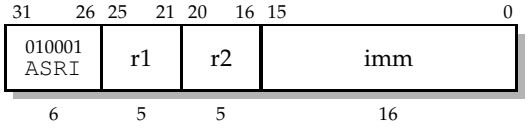
```
reg[r1] = reg[r2] >> imm;
SetStatusWord (reg[r1]);
```

ASRI

ARITHMETIC SHIFT RIGHT IMMEDIATE

Format:

```
ASRI r1, r2, imm
```



Description:

Register `r2` is shifted right `r3` bits and the result is placed in `r1`. The bits shifted in from the left are the same as the sign bit.

Operation:

```
reg[r1] = (signed)reg[r2] >> imm;
SetStatusWord (reg[r1]);
```

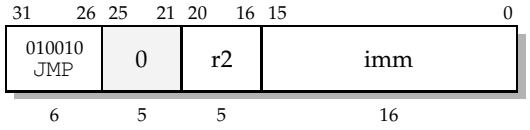
3.3 Control Flow

JMP

UNCONDITIONAL JUMP

Format:

```
JMP r2(imm)
```



Description:

Program execution will continue at address `r2(imm)`.

Operation:

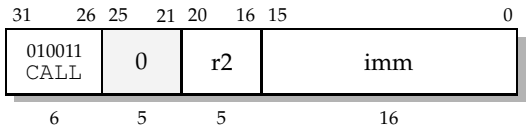
```
pc = reg[r2] + imm;
```

CALL

UNCONDITIONAL SUBROUTINE CALL

Format:

```
CALL r2(imm)
```



Description:

The program counter is stored in register `r31` and program execution will continue at the address specified.

Operation:

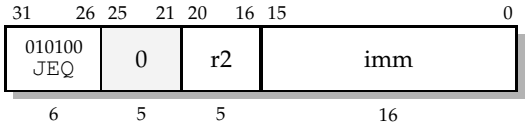
```
reg[31] = pc;
pc = reg[r2] + imm;
```

JEQ

JUMP IF EQUAL

Format:

```
JEQ r2(imm)
```



Description:

Program execution will continue at the address specified if the `Z` bit of the status word is set.

Operation:

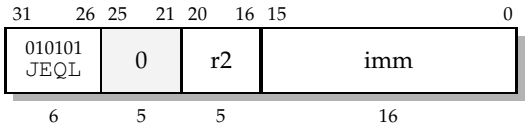
```
if (sw & Z) {
    pc = reg[r2] + imm;
}
```

JEQL

JUMP IF EQUAL LIKELY

Format:

```
JEQL r2(imm)
```



Description:

Program execution will continue at the address specified if the `Z` bit of the status word is set. The "Likely" is a hint to the implementation that the jump is likely to take place and the instruction pipeline should be fed from the destination address. The semantics are the same as those of `JEQ`.

Operation:

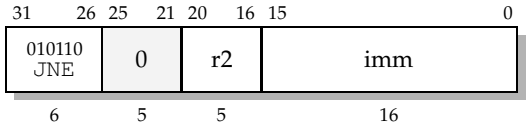
```
if (sw & Z) {
    pc = reg[r2] + imm;
}
```

JNE

JUMP IF NOT EQUAL

Format:

```
JNE r2(imm)
```



Description:

Program execution will continue at the address specified if the `Z` bit of the status word is reset.

Operation:

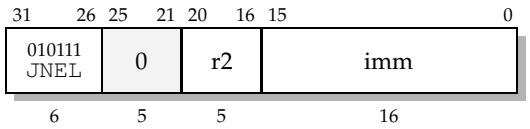
```
if (!(sw & Z)) {
    pc = reg[r2] + imm;
}
```

JNEL

JUMP IF NOT EQUAL LIKELY

Format:

```
JNEL r2(imm)
```



Description:

Program execution will continue at the address specified if the `Z` bit of the status word is reset. The "Likely" is a hint to the implementation that the jump is likely to take place and the instruction pipeline should be fed from the destination address. The semantics are the same as those of `JNE`.

Operation:

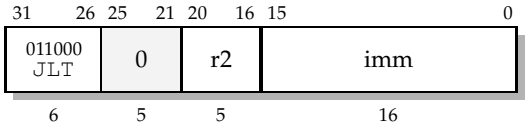
```
if (!(sw & Z)) {
    pc = reg[r2] + imm;
}
```

JLT

JUMP IF LESS THAN

Format:

```
JLT r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` bit of the status word is set.

Operation:

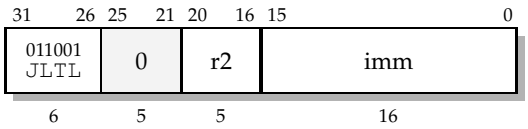
```
if (sw & N) {
    pc = reg[r2] + imm;
}
```

JLTL

JUMP IF LESS THAN LIKELY

Format:

```
JLTL r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` bit of the status word is set. The "Likely" is a hint to the implementation that the jump is likely to take place and the instruction pipeline should be fed from the destination address. The semantics are the same as those of `JLT`.

Operation:

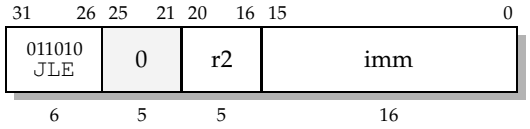
```
if (sw & N) {
    pc = reg[r2] + imm;
}
```

JLE

JUMP IF LESS THAN OR EQUAL TO

Format:

```
JLE r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` and `Z` bits of the status word are set.

Operation:

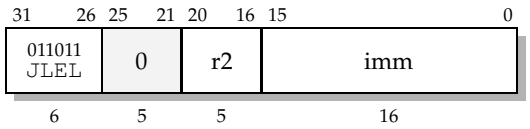
```
if (sw & (N | Z)) {
    pc = reg[r2] + imm;
}
```

JLEL

JUMP IF LESS THAN OR EQUAL TO LIKELY

Format:

```
JLEL r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` and `Z` bits of the status word are set. The "Likely" is a hint to the implementation that the jump is likely to take place and the instruction pipeline should be fed from the destination address. The semantics are the same as those of `JLE`.

Operation:

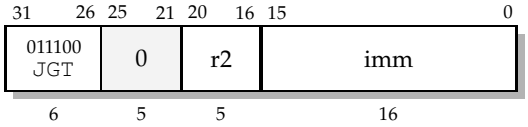
```
if (sw & (N | Z)) {
    pc = reg[r2] + imm;
}
```

JGT

JUMP IF GREATER THAN

Format:

```
JGT r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` and `Z` bits of the status word are reset.

Operation:

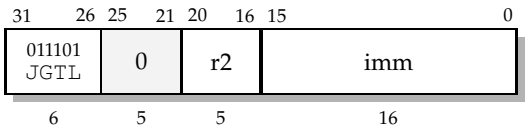
```
if (!(sw & (N | Z))) {
    pc = reg[r2] + imm;
}
```

JGTL

JUMP IF GREATER THAN LIKELY

Format:

```
JGTL r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` and `Z` bits of the status word are reset. The "Likely" is a hint to the implementation that the jump is likely to take place and the instruction pipeline should be fed from the destination address. The semantics are the same as those of `JGT`.

Operation:

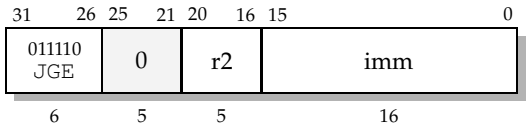
```
if (!(sw & (N | Z))) {
    pc = reg[r2] + imm;
}
```

JGE

JUMP IF GREATER THAN OR EQUAL TO

Format:

```
JGE r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` bit of the status word is reset.

Operation:

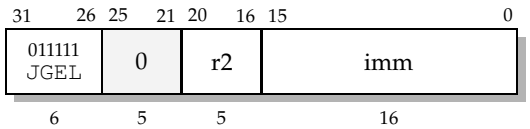
```
if (!(sw & N)) {
    pc = reg[r2] + imm;
}
```

JGEL

JUMP IF GREATER THAN OR EQUAL TO LIKELY

Format:

```
JGEL r2(imm)
```



Description:

Program execution will continue at the address specified if the `N` bit of the status word is reset. The "Likely" is a hint to the implementation that the jump is likely to take place and the instruction pipeline should be fed from the destination address. The semantics are the same as those of `JGE`.

Operation:

```
if (!(sw & N)) {
    pc = reg[r2] + imm;
}
```

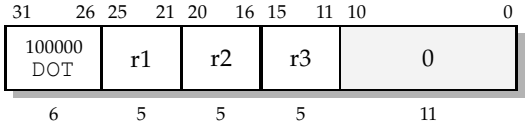
3.4 Special Purpose Arithmetic

DOT

FIXED-POINT VECTOR DOT-PRODUCT

Format:

```
DOT r1, r2, r3
```



Description:

Registers `r2` and `r3` point to two four-element vectors in data memory. Each element is in `s15.16` format. The vectors' dot product, also in `s15.16` format, is stored in register `r1`.

Operation:

```
reg[r1] = 0;
for (i = 0; i < 4; i++) {
    reg[r1] += (dMem[reg[r2] + i] * dMem[reg[r3] + i]) >> 16;
}
```

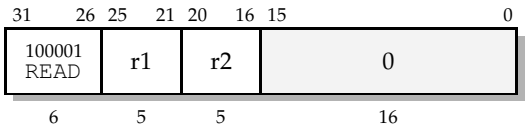
3.5 Direct Memory Access to External RAM

READ

DMA READ FROM EXTERNAL MEMORY

Format:

```
READ r1, r2
```



Description:

The 16 words starting at location in external (shared) memory pointed to by `r2` are copied to the 16 words starting at location in data memory pointed to by `r1`.

Operation:

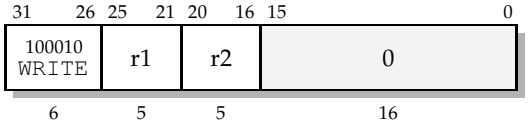
```
for (i = 0; i < 16; i++) {
    dMem[reg[r1] + i] = eMem[reg[r2] + i];
}
```

WRITE

DMA WRITE TO EXTERNAL MEMORY

Format:

```
WRITE r1, r2
```



Description:

The 16 words starting at location in data memory pointed to by `r2` are copied to the 16 words starting at location in external (shared) memory pointed to by `r1`.

Operation:

```
for (i = 0; i < 16; i++) {
    eMem[reg[r1] + i] = dMem[reg[r2] + i];
}
```

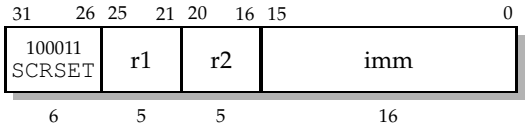
3.6 Rasterization

SCRSET

SET SCREEN LOCATION

Format:

```
SCRSET r1, r2, imm
```



Description:

Registers `r1` and `r2` contain the number of horizontal and vertical pixels on the screen, respectively. The word in data memory pointed to by the immediate operand is used to determine the location of the frame buffer in external memory. The high half-word specifies in kilobytes the starting address of the color frame buffer, and the low half-word specifies in kilobytes the starting address of the Z buffer.

Operation:

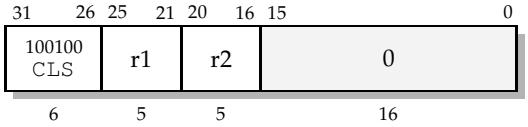
```
width = reg[r1];
height = reg[r2];
fbAddr = (dMem[imm] >> 16) << 10;
zAddr = (dMem[imm] & 0xFFFF) << 10;
```

CLS

CLEAR SCREEN

Format:

```
CLS r1, r2
```



Description:

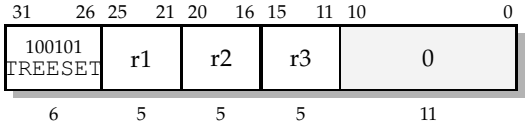
The frame buffer and Z buffer are cleared. Every half-word of the color buffer is set to the low half-word of `r1` and every half-word of the Z buffer is set to the low half-word of `r2`. This instruction is non-blocking and may return before the clear is finished, but the `TREEPUT` instruction will block if `CLS` has not yet terminated.

TREESET

SETUP TREE AND CLEAR ENABLE FLAGS

Format:

```
TREESET r1, r2, r3
```



Description:

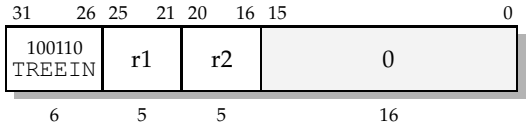
The tree is setup so that its left-most pixel will be at X coordinate `r1` and Y coordinate `r2`. The enable registers of the tree are set to the complement of the lower 16 bits of `r3`. Specifically, the `O` registers will be set to their corresponding bit in `r3` and the `I` registers will be set to the complement of that. Using `r0` enables all of the pixels. The least significant bit corresponds to the right-most pixel in the tree. The X coordinate `r1` must be a multiple of 16.

TREEIN

EDGE EQUATION TEST

Format:

```
TREEIN r1, r2
```



Description:

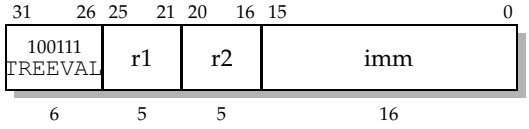
The value in register `r1` is placed at the root of the tree with adder `r2`. After the data has trickled to the bottom of the tree, the left-most leaf will contain `r1`, the next leaf will contain `r1 + r2`, and so on until the right-mode leaf will contain `r1 + 15*r2`. At the leaf, the `I` register will stay a 1 if the result is non-positive. The `O` register will stay a 0 if the result is non-negative. Register `r1` will then be incremented by 16 times `r2` to get it ready for the next call to `TREEIN`.

TREEVAL

PARAMETER INTERPOLATION

Format:

```
TREEVAL r1, r2, imm
```



Description:

The value in register `r1` is placed at the root of the tree with adder `r2`. After the data has trickled to the bottom of the tree, the left-most leaf will contain `r1`, the next leaf will contain `r1 + r2`, and so on until the right-mode leaf will contain `r1 + 15*r2`. The constant `imm` specifies what will happen to the resulting data:

0 The high half-word of the result will be treated like a Z value and will be compared with the Z value for that leaf's pixel. If the computed Z value is greater than the Z-Buffer's value, then the `I` register is cleared and the `O` register is set. The Z value is also stored for later write to the Z-Buffer.

1 Bits 16 through 20 of the result are stored in bits 11 through 15 of the color register at the leaf. This will be displayed as the red color when the color buffer is displayed.

2 Bits 16 through 21 of the result are stored in bits 5 through 10 of the color register at the leaf. This will be displayed as the green color when the color buffer is displayed. This component has one extra bit because the eye is more sensitive to green than it is to red and blue.

3 Bits 16 through 20 of the result are stored in bits 0 through 4 of the color register at the leaf. This will be displayed as the blue color when the color buffer is displayed.

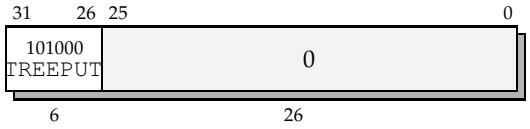
Register `r1` will then be incremented by 16 times `r2` to get it ready for the next call to `TREEVAL`.

TREEPUT

TREE WRITE TO FRAME BUFFER

Format:

TREEPUT



Description:

At each leaf, the value `I | !O` is calculated and used as the enable register, and enabled leaves write their color and Z values to their pixels.

Instruction	Operands	Format	Description
LD	r1, r2(imm)	RA	Load Register
ST	r2(imm), r1	AR	Save Register
LI	r1, r2(imm)	RA	Load immediate
ADD	r1, r2, r3	RRR	Fixed-point Add
SUB	r1, r2, r3	RRR	Fixed-point Subtract
MUL	r1, r2, r3	RRR	Integer Signed Multiplication
MULF	r1, r2, r3	RRR	Fixed-point Signed Multiplication
RECP	r1, r2	RR	Fixed-point Signed Reciprocal
AND	r1, r2, r3	RRR	Bit-wise And
OR	r1, r2, r3	RRR	Bit-wise Or
XOR	r1, r2, r3	RRR	Bit-wise Xor
NOT	r1, r2	RR	Bit-wise Not
LSL	r1, r2, r3	RRR	Logical Shift Left
LSR	r1, r2, r3	RRR	Logical Shift Right
ASR	r1, r2, r3	RRR	Arithmetic Shift Right
LSLI	r1, r2, imm	RRI	Logical Shift Left Immediate
LSRI	r1, r2, imm	RRI	Logical Shift Right Immediate
ASRI	r1, r2, imm	RRI	Arithmetic Shift Right Immediate
JMP	r2(imm)	A	Unconditional Jump
CALL	r2(imm)	A	Unconditional Subroutine Call
JEQ	r2(imm)	A	Jump If Equal
JEQL	r2(imm)	A	Jump If Equal Likely
JNE	r2(imm)	A	Jump If Not Equal
JNEL	r2(imm)	A	Jump If Not Equal Likely
JLT	r2(imm)	A	Jump If Less Than
JLTL	r2(imm)	A	Jump If Less Than Likely
JLE	r2(imm)	A	Jump If Less Than or Equal To
JLEL	r2(imm)	A	Jump If Less Than or Equal To Likely
JGT	r2(imm)	A	Jump If Greater Than
JGTL	r2(imm)	A	Jump If Greater Than Likely
JGE	r2(imm)	A	Jump If Greater Than or Equal To
JGEL	r2(imm)	A	Jump If Greater Than or Equal To Likely
DOT	r1, r2, r3	RRR	Fixed-point Vector Dot-Product
READ	r1, r2	RR	DMA Read from External Memory
WRITE	r1, r2	RR	DMA Write to External Memory
SCRSET	r1, r2, imm	RRI	Set Screen Location
CLS	r1, r2	RR	Clear Screen
TREESET	r1, r2, r3	RRR	Setup Tree and Clear Enable Flags
TREEIN	r1, r2	RR	Edge Equation Test
TREEVAL	r1, r2, imm	RRI	Parameter Interpolation
TREEPUT		OP	Tree Write to Frame Buffer

Table 3.1: Instruction Set Summary

4 Rasterization

The GE uses the *plane equation* method of triangle rasterization, which means that every pixel in the bounding box of the triangle is tested, and those that are inside have their parameters (color in this case) interpolated. The linear function $Ax + By + C$ is at the heart of this method, where x and y are pixel coordinates and A, B, and C are `s15.16` fixed-point coefficients.

4.1 Edge Equations

Each edge of a triangle can be described by the equation $Ax+By+C = 0$. Given the three vertices (x_0, y_0), (x_1, y_1), and (x_2, y_2), we can calculate the three edge equations as follows:

$$A = y_0 - y_1$$

$$B = x_1 - x_0$$

$$C = y_1 x_0 - y_0 x_1$$

for each pair of vertices. (This is two multiplies and three adds per vertex. The edge equations can be reused for triangle meshes.) Each pixel inside the bounding box is plugged into the equations. If all three signs are the same (counting zero as both signs), then the pixel is inside the triangle. The equations can be computed incrementally as the row is being traversed.

4.2 Parameter Interpolation

Once a pixel is determined to be inside the triangle, its color needs to be interpolated. Color is specified as a color triplet per vertex, and a linear interpolation is used between the vertices. The function $Ax + By + C$ is used—it is essentially a plane through the three color points. The constants A, B, and C can be found by solving a system of linear equations, since we have three equations (one for each vertex) and three unknowns:

$$Ax_1 + By_1 + C = Red_1$$

$$Ax_2 + By_2 + C = Red_2$$

$$Ax_3 + By_3 + C = Red_3$$

(With similar equation for green and blue.) Cramer's Rule can be used to analytically solve the system:

$$det = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

$$A = \frac{Red_1y_2 - Red_2y_1 - Red_1y_3 + Red_3y_1 + Red_2y_3 - Red_3y_2}{det}$$

$$B = \frac{x_1Red_2 - x_2Red_1 - x_1Red_3 + x_3Red_1 + x_2Red_3 - x_3Red_2}{det}$$

$$C = \frac{(x_2y_3 - x_3y_2)Red_1 - (x_1y_3 - x_3y_1)Red_2 + (x_1y_2 - x_2y_1)Red_3}{det}$$

Calculating det takes 2 multiplies, one add, and one reciprocal. A and B each take 7 multiplies and 5 adds. C takes 10 multiplies and 5 adds. This needs to be done once per triangle and almost none of it can be re-used in triangle meshes.

4.3 Rasterization Hardware

The inner loop of the plane equation rasterizer must perform the following operations:

1. Test three planes to see if they have the same sign;
2. If same sign, write color and Z to frame buffer; and
3. Recalculate R, G, B, Z, and three plane equations.

If the operations were performed with the general-purpose RISC instructions, only about 30,000 triangles per second would be possible with a 100MHz clock. Instead, we rasterize 16 horizontal adjoining pixels at a time using a PixelPlanes-like linear expression evaluation tree.

4.4 Linear Expression Tree

A linear expression tree is a binary tree whose root holds an input constant C and whose leaves calculate the constant plus some multiple of another constant

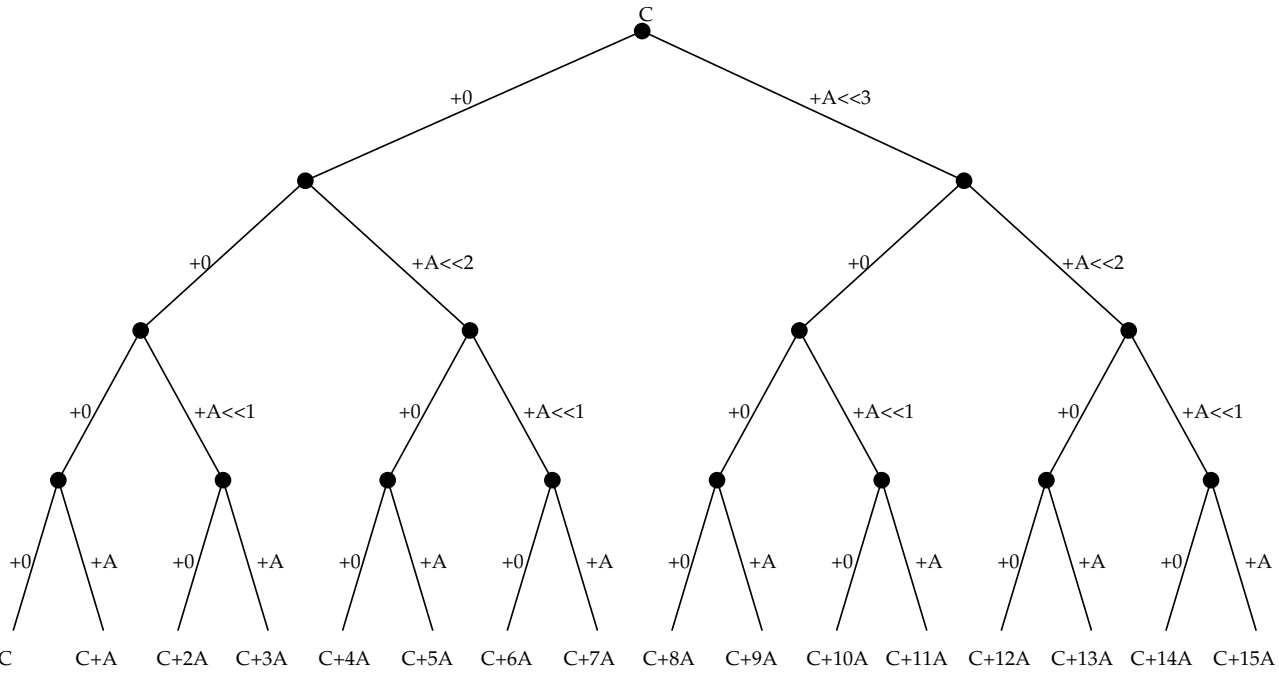


Figure 4: Linear Expression Tree

A. In our case, we use a 4-level tree and the leaves hold the value $C + iA$, where i is an integer from 0 to 15. The tree is implemented roughly as in Figure 4.

Each right child represents an addition of the parent's value with the multiplicant A shifted by some amount. Each left child is a delay of the parent's value. The additions are performed by 15 32-bit fixed-point adders. The tree is pipelined so as to produce one tree evaluation per clock cycle with a 3 clock cycle lead.

4.5 Enable Registers

Each tree node has one enable register, which determines whether the color and Z values should be written to memory. This enable register `E` is not an actual memory bit, but instead is the equation `E = I | !O`, where `I` and `O` are one-bit registers. On a `TREECLR`, the `I` registers are set to 1 and the `O` registers to 0 according to the specified register. On `TREEIN` instructions, the sign bits of the results are ANDed with the `I` register and ORed with the `O` register. (A result of zero is counted as positive for the `O` register and negative for the `I` register. This insures that vertex ordering does not affect the triangle.) The `E` register are then testing whether all three `TREEIN` commands have the same sign.

4.6 Tree Setup

The frame buffer is setup with the `SCRSET` instruction, which specifies the width, height, address of the color buffer, and address of the Z buffer. The `CLS` instruction is used to set the contents of the color and Z buffers to a particular value.

The tree setup instruction (`TREESET`) tells the tree where the 16-pixel span is located on the screen. When the instruction is executed, the Z values for that span are read from memory (asynchronously with the rest of the tree operations) so that the first `TREEVAL` instruction with `n == 0` can do a compare.

4.7 Pipeline

The tree is pipelined so that each level can be used independently of the others. The timeline is displayed in Table 4.1.

Instruction	Level 0	Level 1	Level 2	Level 3	Actions at leaves
`TREESET`					Read Z from Z-Buffer
`TREEIN`	1				
`TREEIN`	2	1			
`TREEIN`	3	2	1		
`TREEVAL`	Z	3	2	1	Modify enable
`TREEVAL`	R	Z	3	2	Modify enable
`TREEVAL`	G	R	Z	3	Modify enable
`TREEVAL`	B	G	R	Z	Compare, modify enable
`TREEPUT`	?	B	G	R	Move bits into color
`X += 16`		?	B	G	Move bits into color
`IF X < XMAX`			?	B	Move bits into color
`REPEAT`				?	Write enabled colors

Table 4.1: Tree pipeline

Assuming a 100 MHz clock and one cycle per tree instruction, there is a 110 ns delay from the time when the frame buffer's Z values can be read to the time when they are needed. In practice, it is likely that the color values for that span in the frame buffer will also be read to take advantage of modern packet-oriented memory busses. See the implementation chapter for details.

5 Implementation

This document describes the GE's architecture and not its implementation. However, since the GE has specific performance requirements, this chapter makes a few implementation suggestions, both to guide the implementer and to support the architects' design decisions.

5.1 Memory Interface

In the last twenty years, processor performance has increased by a factor of 100. Unfortunately, memory access times have only increased by a factor of 10, and nowhere is this discrepency more visible than in graphics systems, where frame-buffer bandwidth is almost always the performance bottleneck. The Graphics Engine is no exception, and although it can rasterize pixels at a high rate using the tree, it is not clear that this rendering power is fully usable.

The rasterization tree allows the GE to generate a peak of 16 pixels every 13 clock cycles. For each pixel, two bytes of Z have to be read from memory and a maximum of 4 bytes (Z and color) have to be written back (assuming the Z compare succeed, and they almost always do[1]). Assuming a 100 MHz clock, this requires an average of

$$\frac{2+4 \text{ bytes}}{1 \text{ pixel}} \times \frac{16 \text{ pixels}}{13 \text{ cycles}} \times \frac{10^8 \text{ cycles}}{1 \text{ second}} \approx 738 \text{ Mbytes/second}$$

or one byte of memory access every 1.3 ns. This is far faster than any modern memories are capable of providing. The fastest standard DRAMs can be run at speeds up to 100 ns. Synchronous DRAMs (SDRAMs) can be configured to deliver bytes every 10 ns. RDRAM[4] by Rambus Inc., a recent promising technology, uses a packet-oriented bus and memory package that can deliver bytes every 2 ns. Even assuming that this 2 ns rate could be sustained, which it cannot because of the protocol overhead and cache miss ratio, it would still lag behind the tree.

[1] The concept of *depth complexity* describes how many times a pixel on the screen is covered by a triangle. A well-designed database will have an average depth complexity close to 1, and therefore most Z comparisons will succeed.

The Rambus technology, however, comes close enough that we feel that the tree is justified. For example, although the cache miss ratio would be fairly high, the Rambus standard allows the bus master (in this case the GE) to fill the cache lines ahead of time. If the frame buffer were properly interleaved across the memory chips, the GE could fill the cache of the next horizontal span while the current span was being rasterized.

Another recent memory technology is 3DRAM[1] (formally known as FBRAM) from Sun and Mitsubishi. With 3DRAM, the Z compare operation is performed by an ALU on the memory chip. This reduces bandwidth by converting an essentially Read-Modify-Write operation into a Write-Mostly operation. Current 3DRAMs support a configuration for a 640x512x8 double-buffered display with 16 bits of Z. This configuration has half as many bits for color as is generated by the graphics engine, but it might be possible to organize the 3DRAM differently to allow for 16 bits of color.

In addition, to make up for low memory bandwidth, the interface between the GE's RISC core and the tree can be implemented as a first-in-first-out queue to allow the GE to continue processing the next triangle asynchronously if delays in memory accesses block the tree's operation.

6 Application Interface

6.1 Display Lists

The display lists will be stored in 4-byte words. The high-order byte of the first word is the "op-code", to be interpreted by the program running on the GE. Some op-codes will require additional words as arguments.

`00 00 00 00`
: Return from subroutine call. If the stack is empty, processing of the display list terminates and the video buffer is swapped. This word is entirely zero so that attempting to run empty memory will end processing.

`01 AA AA AA`
: Subroutine call. AAAAAA is the address of the next display list entry. Three bytes can address 24 megabytes, which is more than enough. There will be a limited stack on the GE for return addresses.

`02 VV XX XX  YY YY ZZ ZZ`
: Vertex. XXXX, YYYY, and ZZZZ are coordinates in `s15.0` format. VV is the number of the vertex, starting at 0. A normal triangle would have three vertices, numbered 0, 1, and 2. A vertex numbers greater than 2 will be taken modulo 3. One triangle will be rasterized for every vertex numbered 2 or above. Triangle meshes therefore have a limit of 254 triangles.

`03 00 00 00`
: Push model-view matrix. The matrix stack can hold eight matrices. (*I.e.*, there can only be seven pushes.)

`04 00 00 00`
: Pop model-view matrix.

`05 00 00 WW  MM MM MM MM  ...  MM MM MM MM`
: Load matrix. WW is 0 when the projection matrix should be loaded and 1 when the model-view matrix should be loaded. The op-code word is

followed by 16 words for the matrix stored in row-major order. MMMMMMMM is a 32-bit fixed-point number in `s15.16` format. The current projection or model-view matrix is replaced by the new matrix.

`06 00 00 WW  MM MM MM MM  ...  MM MM MM MM`
Multiply matrix. Identical to Load Matrix except that the current projection or model-view matrix is post-multiplied by the given matrix.

`07 LX LY LZ  00 RR GG BB`
Specify Lighting. LX, LY, and LZ are elements (in `s0.7` format) of a normalized vector pointing in the direction of the light. RR, GG, and BB are the color of the light in `.8` format.

`08 XX XX XX`
Set mode. The modeword is OR'ed with XXXXXX.

`09 XX XX XX`
Reset mode. The modeword is AND'ed with the inverse of XXXXXX.

`0A RR GG BB`
Set color. Sets the current surface color for lighting mode. Elements are in `.8` format.

`0B NX NY NZ`
Set normal. Sets the current surface normal for lighting mode. Elements are in `s.7` format.

6.2 Mode Word

The GE will keep an internal modeword of 32 bits, 24 of which will be accessible by the user. The bits will have the following meaning.

BIT 0 — ENABLE LIGHTING

If 0 lighting is disabled, color in the vertex structure is used for the vertex's color.

If 1 lighting is enabled, normal in the vertex structure is transformed, possibly normalized, and used to determine vertex color.

BIT 1 — NORMALIZE NORMALS

If 0 normals are not normalized after being transformed by the model-view matrix. Normalization is not necessary if they are already normalized in the structure and non-uniform scaling is not used in the model-view matrix.

If 1 normals are normalized after being transformed by the model-view matrix

BIT 2 — GOURAUD SHADING

If 0 the color of the triangle will be that of the the last vertex.

If 1 the color of the vertices will be interpolated across the triangle. (Note that this mode is independent of lighting.)

BIT 3 — ENABLE Z-BUFFER READING

If 0 the Z-Buffer will not be read to determine visibility of a pixel. The pixel will always be written.

If 1 the Z-Buffer's value will be compared with the new pixel's depth to determine visibility. If the new pixel's Z value is less than or equal to the Z-Buffer's value, the pixel will be written.

BIT 4 — ENABLE Z-BUFFER WRITING

If 0 the Z-Buffer will not be updated with the new Z value if a pixel is written to it.

If 1 the Z-Buffer will be updated with written pixels' Z values.

BIT 5 — ENABLE BACKFACE CULLING

If 0 there will be no backface culling.

If 1 backfaced polygons will be culled. A backface polygon is one whose vertices are oriented clockwise when viewed in camera space.

7 Rationale and Discussion

7.1 High-level Architecture

Seperating the main CPU and the Graphics Engine allows the GE to be optimized for rasterization without regard or speculation about the kinds of application that the game writer will want to use for game dynamics. It also adds a level of parallellism necessary to achieve the performance goals stated in the first chapter. Communication through shared memory is practical and efficient, and a hierarchical display list is the natural choice for database specification.

7.2 Storage

As the sample application demonstrates (see the appendix), 1024 words of on-chip instruction memory is sufficient for basic rasterization, with plenty of room left over for advanced features such as full Phong lighting, support of non-uniform scaling, level-of-detail support, and complex primitives (*e.g.*, splines or spheres). The data memory's 1024 words is more than enough for the sample application and a typical program on this architecture is likely to run out of instruction space before it runs out of data space.

Many current RISC architectures have 32 general-purpose registers, and we follow this trend. The number of bits required to address them (5) fits well into the instruction format, and even at their fullest use (while rasterizing a triangle) there is little swapping to memory. The size of a word (32 bits) allows for good fixed-point precision while minimizing memory and bandwidth waste. Only allowing word-length accesses to memory simplifies the instruction set and minimizes the addressing bits while finessing the question of endianess. This came at virtually no cost: in very few places in the sample application would a sub-word access have saved an instruction.

7.3 Instruction Set

Preliminary simulations of the hardware and software indicated that the most common instructions are those that manipulate the rasterization tree. These instructions are therefore optimized to do the most amount of work in the least amount of time. For example, they automatically increment one of the operand registers by 16 times the other operand register, so as to get the registers ready for the next iteration of the loop. This saves 7 instructions used to increment these registers.

Computing the plane equations for each triangle requires about 70 additions, 90 multiplications, and one reciprocal per triangle. Because of the choice of fixed-point data formats (s15.16), a fixed-point multiply is useful in addition to the integer multiply. The reciprocal is implemented in software because it is almost as fast as a hardware implementation and relatively rarely used.

A common operation, both in transformations and in lighting, is the dot product. A sequential 4-element vector dot product requires four multiplications and 3 additions. If we guess that additions take one clock cycle and multiplications take ten (based on the MIPS R4000 RISC chip specifications[3]), a dot product will take 43 clock cycles, not counting the work required to load the matrix row into registers. Instead, this common operation is implemented on the GE as an instruction that multiplies two vectors directly in data memory. On-chip data memory is likely to be as fast as registers, and the four multiplications can occur simultaneously. The additions can happen in two steps, leaving the sum in a register, for a total time of 12 clock cycles, without any additional need for memory loads.

The single addressing mode (base-offset) simplifies instruction decoding while allowing interesting uses of a few instructions. For example, the `LI` (Load Immediate) instruction effectively loads a register with the address referenced by the addressing register/immediate pair. This not only allows a register to be loaded with an immediate (using `R0` as the base register), but a register can be incremented by using itself as the base register. This is similar to the IBM System/360's `LA` (Load Address) instruction and the Intel 8088's `LEA` (Load Effective Address) instruction. Also, all of the jump and branch instructions take a base-offset pair for the destination address, allowing both absolute branches and jump tables.

The instruction format is very orthogonal. There are only two basic instruction formats: RRR (three registers) and RRI (two registers and a 16-bit immediate). All instructions use a subset of either, which greatly simplifies instruction decoding. All base-offset instruction use the RRI format with the second register as the base and the immediate as the offset. (Instructions that only need one register, such as branches, leave the first register unused.)

A graph of instruction frequencies executed when rendering a typical scene is included in the appendix.

7.4 Precision

7.5 Rasterization Hardware

The linear expression tree is an efficient way to rasterize using the plane-equation method: Adders are cheap and generating the parameters is easily done.

The `TREECLR` takes one argument whose bits specify which tree leaves should be enabled and which should be disabled. The register is usually `R0` (zero), or all leaves enabled, but can be set to another pattern to allow for screen-door transparency.

7.6 Peculiarities

7.6.1 Instruction Set

Several standard instructions are missing from the Graphics Engine's instruction set. These can either be replaced by other instructions or emulated easily in software. In the first case, it is up to the assembler whether it should provide pseudo-ops for the missing instructions.

No MOV instruction. A register-to-register move can be accomplished with `OR R1,R2,R2`.

No CMP instruction. A comparison between registers can be accomplished with `SUB R0,R1,R2`, effectively comparing `R1` and `R2` and putting the result in the zero register `R0`, which has no effect other than setting the status bits based on the result of the subtraction.

No NEG instruction. A register can be negated with a `SUB R1,R0,R1` instruction.

No TST instruction. A register's contents can be used to set the status bits with a `SUB R0,R1,R0` instruction, setting the `N` bit if `R1` is negative and the `Z` bit if it is zero.

No NOP instruction. A `LD R0,R0(0)` instruction can be used to perform no operations, since the result is placed into the zero register `R0` and the source address is always valid. This instruction has the additional benefit of being encoded as `0x00000000`. (This is similar to the Intel 8088's use of `XCHG AX,AX` as `NOP`.) Note this this is not strictly a no-op since the status bits get set according to the value loaded. This is probably not a problem since no-ops will mostly be used to fill the branch delay slot, where the result of the status bits has already been tested.

No RECP instruction. A reciprocal can be computed easily in software in almost the same amount of time as it would take a hardware instruction. (The MIPS R4000 chip requires 69 clock cycles to perform a division[3].) Profiling the sample application showed that a reciprocal instruction did not occur fre-

quently enough to be worth the hardware real-estate that would have to be devoted to it.

No SQRT instruction. Although finding the square root of a number is useful when normalizing vectors, again a software routine could compute the reciprocal of a square root as fast as a hardware implementation could. Not only are reciprocals of vectors more useful than plain square roots (since they are then multiplied by the components to normalize them), but they are easier to compute and the square root can be found by multiplying by the original number. (The lack of a square-root instruction is not especially peculiar, but this seemed like the right section to mention it.)

7.6.2 Architecture

A Harvard architecture serves the purposes of the application. The program does not generally need to write or read instruction memory, and splitting the instruction and data memory allows the implementation to fetch words from both simultaneously.

7.6.3 Register Space

Although 31 general-purpose registers is enough for most purposes, the inner loop of the rasterizer needs four registers per linear equation for seven equations, plus several registers for the bounding box. Only three registers per equation may be used if the fourth is stored in memory, with a 7 cycle hit per iteration of the Y (outer) loop. The register set could have been extended to 63 registers fairly easily, at the cost of the extra real-estate on the chip and the modification of the RRI instruction format to store 14 bits for the immediate instead of 16 bits. (The RRR format has enough free bits to accomodate the larger operand size.) Profiling the sample application revealed that loading 16 bits of immediate was frequent enough that the extra registers would not have been worth it. Although this decision may turn out to be a mistake, a future (incompatible) architecture could extend the register set to 63 with minimal impact on the rest of the architecture.

7.6.4 Fixed Rasterization Tree

Perhaps the biggest limitation of this architecture is that the result of the rasterization tree can only go straight to memory. The tree might have proved more useful if simple blending or other arithmetic operations were possible on the results of the tree.

7.6.5 No Texture Mapping

A small 32×32 texture map could have easily fit on the chip, and the perspective-correct interpolation of the texture coordinates would have been fairly straight-

forward. Yet, texture mapping was left out because it was not clear that the texture itself could be moved into the chip fast enough, possibly on a per-primitive basis. In addition, point-sampled texturing has a tendency to look awful and mipmapped textures would have significantly complicated the architecture. The architects realize that texture mapping is an important part of graphics, especially games, and the next revision of the architecture would certainly have textures as its first new feature.

8 Sample Code and Considerations

8.1 Performance Enhancements

This section outlines a few suggestions to application writers who want to maximize the performance of the graphics system.

8.1.1 Double-buffered Display Lists

In the system described above, at every frame the CPU generates the entire display list for that frame. In many applications, the display list will stay the same except for the transformation matrices. For example, a flight simulator might have transformation matrices for the camera, the airplanes, the landing gears, and the missles, but the rest of the geometry will not change between frames. To reduce bandwidth, the dynamic part of the display list could be double buffered while keeping the static part single-buffered.

To accomplish this, the matrix load and matrix multiply nodes of the display list should have pointers to the matrices instead of including them in the display list itself. During the synchronization step at the beginning of a frame, the CPU would provide the base of the current dynamic buffer, and matrix pointers would be added to this base when loaded during display list parsing.

Although this optimization will increase the burden on the graphics engine slightly, it will drastically reduce both the burden the CPU and the bandwidth between the CPU to main memory.

8.2 Programming Examples

The symbolic assembler supports naming of registers, where a symbolic name can be attached to a particular register. All upper-case identifiers in the sample code fragments below are register names, and all lower-case identifiers are addresses in memory.

8.2.1 Matrix Times Vector

A matrix is easily multiplied by a vector by using the DOT instruction. The matrix needs to be in row-major order, and the result is in four registers.

```
    LI      MTX, modelview      ; Address of matrix
    LI      VEC, vector         ; Address of vector
    DOT     V1, MTX, VEC
    LI      MTX, MTX(4)
    DOT     V2, MTX, VEC
    LI      MTX, MTX(4)
    DOT     V3, MTX, VEC
    LI      MTX, MTX(4)
    DOT     W, MTX, VEC
```

8.2.2 Matrix Times Matrix

The DOT instruction can be used here too, but the second matrix must be in column-major order. The most frequent time that two matrices are multiplied is when the user specifies a new matrix to post-multiply by the model-view matrix. The new matrix can be transposed as it is being loaded in with no penalty. Note good usage of the branch delay slot.

```
    LI      COUNT,4             ; Number of columns to copy
    LI      TMPDEST,tmat1       ; Transpose into tmp matrix
    LI      DEST,modelview      ; Else load the modelview
transpose:
    CALL    getinst             ; Get next matrix element
    ST      TMPDEST,INST        ; Store the element
    CALL    getinst             ; Get next matrix element
    ST      TMPDEST(4),INST     ; Store the element
    CALL    getinst             ; Get next matrix element
    ST      TMPDEST(8),INST     ; Store the element
    CALL    getinst             ; Get next matrix element
    ST      TMPDEST(12),INST    ; Store the element
    LI      COUNT,COUNT(-1)     ; Decrement count
    JNEL    transpose           ; If not done, next element
    LI      TMPDEST,TMPDEST(1)  ; Increment destination

    ; Now the transpose of the new matrix is in tmat1.

    LI      COUNT,4             ; # of rows to multiply
multmatrix:
    LI      SRC,tmat1           ; 1st column of new matrix
    DOT     PROD1,DEST,SRC      ; Row * column1
    LI      SRC,tmat1+4         ; Next column
    DOT     PROD2,DEST,SRC      ; Row * column1
```

```
    LI      SRC,tmat1+8         ; Next column
    DOT     PROD3,DEST,SRC      ; Row * column1
    LI      SRC,tmat1+12        ; Next column
    DOT     PROD4,DEST,SRC      ; Row * column1
    ST      DEST,PROD1          ; Store first element
    ST      DEST(1),PROD2       ; Store second element
    ST      DEST(2),PROD3       ; Store third element
    ST      DEST(3),PROD4       ; Store fourth element
    LI      COUNT,COUNT(-1)     ; Decrement count
    JNEL    multmatrix          ; If not done, next row
    LI      DEST,DEST(4)        ; Next row of source
```

8.2.3 Rasterization Loop

This loop uses the values generated for the plane and edge equations. It loops over the bounding box of the triangle, rasterizing 16 horizontal pixels at a time. Notice that the inner X loop is very tight, especially with the incrementing of X put in the branch-delay slot. The bounding box had to be shrunk by 16 horizontally (see first line) to allow the testing of X to occur before incrementing it.

```
    ; Loop over bounding box
    LI      MAXX,MAXX(-16)      ; To fill branch delay slot
yloop:
    LD      X,minx              ; Start of X loop

    OR      ZC,ZCC,ZCC          ; Load beginning of line values
    OR      RC,RCC,RCC
    OR      GC,GCC,GCC
    OR      BC,BCC,BCC
    OR      C1,CC1,CC1
    OR      C2,CC2,CC2
    OR      C3,CC3,CC3

xloop:
    TREESET X,Y,R0              ; Setup tree and clear enable bits
    TREEIN  C1,A1               ; First edge
    TREEIN  C2,A2               ; Second edge
    TREEIN  C3,A3               ; Third edge
    TREEVAL ZC,ZA,0             ; Interpolate Z
    TREEVAL RC,RA,1             ; Interpolate red
    TREEVAL GC,GA,2             ; Interpolate green
    TREEVAL BC,BA,3             ; Interpolate blue
    TREEPUT                     ; Stick it into memory

    SUB     R0,X,MAXX           ; Stop at right end of bounding box
```

```
JLEL    xloop
LI      X,X(16)             ; Increment X

LD      TMP, planes+1
ADD     ZCC,ZCC,TMP
LD      TMP, planes+4
ADD     RCC,RCC,TMP
LD      TMP, planes+7
ADD     GCC,GCC,TMP
LD      TMP, planes+10
ADD     BCC,BCC,TMP
LD      TMP, planes+12
ADD     CC1,CC1,TMP
LD      TMP, planes+13
ADD     CC2,CC2,TMP
LD      TMP, planes+14
; Addition of TMP to CC3 done in branch delay slow below

LI      Y,Y(1)              ; Increment Y
SUB     R0,Y,MAXY           ; Stop at top of bounding box
JLEL    yloop
ADD     CC3,CC3,TMP
```

Bibliography

[1] FBRAM Specification, *Mitsubishi Electric,* March 1994.

[2] Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice,* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

[3] Heinrich, Joseph. *MIPS R4000 User's Manual,* Englewood Cliffs, New Jersey: Prentice Hall, 1993.

[4] Rambus, Inc. *Architectural Overview*, 1992.

Index