VHDL Based Design of Graphics ASICs

M. White, G. J. Dunnett, P. F. Lister, R. L. Grimsdale *

Abstract

The design of graphics ASICs for geometry and rasterisation processing has traditionally involved the use of schematic design entry whereby functional blocks are netlisted and instantiated on the schematic. This methodology is fine at the top most hierarchical levels of a design but becomes tedious and error prone at the lower gate levels. Often these designs are targetted at custom ASICs through the use of silicon compiler technology. Unfortunately, this is an expensive and risky approach to implementing these ASICs, particularly for University research laboratories where additional funding may not be available to cover non-recurring engineering costs, such as multiple mask runs, which may be needed due to design errors. This paper presents an alternative to these traditional approachs. A new approach, top down ASIC design with logic synthesis and optimisation targetting FPGA ASICs, is presented. We demonstrate through some examples of our texturing and scan conversion hardware the benefits of this new approach.

1 Introduction

The VLSI and Computer Graphics Research Group at Sussex have been primarily involved in semi-custom VLSI ASIC design of both graphics geometry and rasterisation hardware [7, 6, 5]. However, this semi-custom VLSI design has still traditionally involved large non-recurring engineering (NRE) costs, long prototype delivery times, and inherent risks which are unacceptable in our research environment. Ideally, we require all the benefits of full or semi-custom circuits, i.e. high density and speed, with low cost, low risk, low prototype time and a quick route to silicon. Further, in our research environment it is difficult to build up and keep the experience required to successfully carry through full or semi-custom masked designs. This implies even greater costs and has led us to consider alternative routes to silicon.

This paper discusses our current approach to designing graphics hardware without the complexities of targetting mask based ASICs. It sets out our views on the use of VHDL, logic synthesis and optimisation as a design strategy for targetting field programmable gate arrays (FPGAs). We conclude with some examples from our current texturing and shading hardware designs.

2 Top Down ASIC Design with Logic Synthesis and Optimisation

Top down ASIC design requires a consistent high level design definition and specification medium. This requirement is satisfied by a high level hardware description language (HDL). There are many to choose from, the two most common being VHDL and Verilog. Technology vendors often

*VLSI and Computer Graphics Research Group, School of Engineering, University of Sussex, Falmer, Brighton, BN1 9QT, England.

have their own HDLs too, e.g. Altera have AHDL. We have chosen to adopt the IEEE 1076 VHDL standard [1]. Designs can be quickly defined in VHDL and proved through simulation. Tedious gate level implementations can be eliminated by using logic synthesis to automatically convert the VHDL to a generic gate level. Logic optimisation can then optimise and map this generic netlist to a target technology. Microarchitectural selection during the optimisation phase allows the designer to optimise for area and speed tradeoffs before targetting the technology. There are many advantages to using this top down approach:

- Advantages of top down ASIC design
  - VHDL provides a consistent and portable design medium
  - Designs are quickly defined, VHDL can be used in specification and implementation
  - Synthesis rapidly creates the gate level description
  - Designers productivity dramatically increases
  - VHDL allows the designer to focus on higher level abstract functionality rather than tedious gate level implementation
  - Automatic VHDL generation from parameterised logic blocks, e.g. Autologic Blocks
  - Design process is technology independent
  - Design decisions and architectural tradeoffs made independent of technology
  - Retargetting technologies is easy, e.g. gate array ASICs, FPGA ASICs, PLDs, etc.
  - Design changes easily and rapidly made, e.g. datapath widths.
  - Production schedules only affected by time taken to modify VHDL due to automated synthesis and optimisation.
  - Inherent documentation with VHDL
  - ...

This top down ASIC design strategy is illustrated in figure 1. We now present a brief overview of the VHDL modelling, logic synthesis and optimisation parts of this strategy.

2.1 VHDL Overview

VHDL is a hardware description language supporting many of the features available in high level programming languages. Components can be described using constructs such as `CASE`, `IF-THEN-ELSE`, `LOOP`, functions and subroutine calls. Concurrent execution of statements simplifies the modelling of components. Once components have been created they can be instantiated into other VHDL models in an object oriented style. This offers designers real scope for component reuse.

2.1.1 Entity Description

A VHDL component model comprises an entity and architecture description. The components interface description—signal names, directions and types—is declared in the VHDL entity. Generic parameters such as time delays can also be supplied in the entity description. For example:

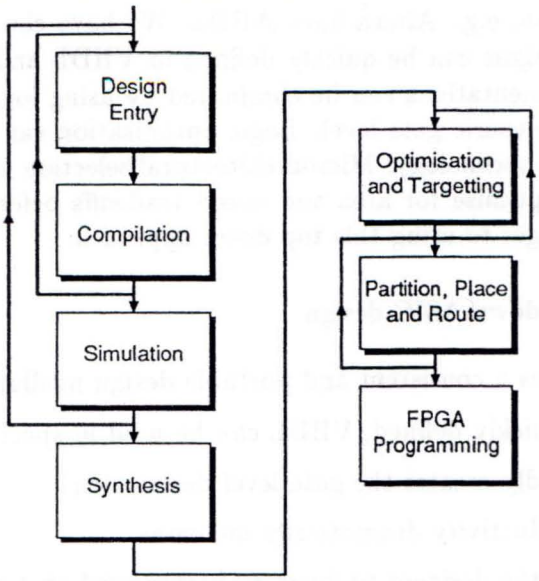

Figure 1: Design strategy incorporating VHDL design entry, simulation, synthesis, optimisation and FPGA targetting

```
ENTITY mux16 IS
  PORT (x, y : IN word16;
        s    : IN bit;
        z    : OUT word16
  );
  GENERIC (delay : time);
END mux16;
```

describes the interface for a 2:1 16 bit wide multiplexer, which can be parameterised with a variable propagation delay.

2.1.2 Architecture Description

The functionality or structural implementation of a component is given in the architecture description. A component can have many architectures, specifying different levels of abstraction, however all architectures share a common entity description. The following example shows both structural and behavioural models for the multiplexer above.

```
ARCHITECTURE structure OF mux16 IS
  COMPONENT mux8 PORT (a, b : IN word8; s : IN bit; c : OUT word8);
  END COMPONENT;
BEGIN
  m1 : mux8
      PORT MAP (x(15 DOWNTO 8), y(15 DOWNTO 8), s, z(15 DOWNTO 8));
  m2 : mux8
      PORT MAP (x( 7 DOWNTO 0), y( 7 DOWNTO 0), s, z( 7 DOWNTO 0));
END structure;

ARCHITECTURE behaviour OF mux16 IS
```

```
BEGIN
  mymux16:
  PROCESS (x, y, s)
  BEGIN
    IF s = '0' THEN
      z <= x AFTER delay;
    ELSE
      z <= y AFTER delay;
    END IF;
  END PROCESS mymux16;
END behaviour;
```

The `structure` architecture instantiates two 8 bit multiplexers to build the 16 bit multiplexer. The port map contruct specifies the connectivity. In the second architecture body a `PROCESS` statement is used to force sequential execution of its in scope statements. These instructions are executed once each time any of the signals `x`, `y` or `s` change state. These signals form the *sensitivity list* for the process. Signal assignments (denoted by `<=` can be specified to occur at some future time in the simulation, as shown here with the `AFTER` clause. This timing information is ignored by the synthesis process because it has no meaning to the final technology. When the mux16 component is bound to other components in another VHDL file the architecture to use can be specified. It is beyond the scope of this paper to describe VHDL in any more detail, the reader is referred to the many texts available [10, 4].

2.2 Synthesis

To synthesise VHDL certain guidelines need to be followed. Style, syntax, modeling and design methods are some of these guidelines. Guidelines are needed because a subset of VHDL is commonly used for synthesis. This is required because there are certain elements of VHDL that are not possible to synthesise for obvious reasons, e.g. textio, while loops, generics, etc. In general the designer will have modelled in behavioural VHDL. This is not synthesisable, so the next step is to decompose this behavioural model into a synthesisable model. This synthesisable model need not go down as far as the structural level described above. The synthesisable model is referred to as a register transfer level (RTL) model. In fact, this RTL model is highly readable and in many cases the designer will opt to only write VHDL in this style.

Before synthesis begins global and process constraints are set. These include setting the type of flip flops use, state encoding schemes, carry look ahead. This enables the designer to make architectural tradeoffs early in the design cycle. The synthesis process examines the RTL description for mappable constructs and produces a generic gate level netlist. Parts of the VHDL will be sensitive to clock edges, which explicitly requires latches. Alternatively variables and signals may imply physical storage is required. Here the synthesiser will insert storage elements into the generic gate-list.

A synthesiser will also understand a subset of arithmetic operations such as addition, subtraction and multiplication. From examining the operands a synthesiser will be able to build logic with the appropriate data-widths. Signed and unsigned versions can be built again by examining the number ranges in use. Typically the arithmetic units can be globally optimised by specification of the degree of carry look ahead to use. This is very useful because it moves some of the architectural tradeoff decisions even higher up the design strategy.

Other VHDL contructs which can be mapped into hardware include the `when` and `case` statements, which result in synthesis of multiplexers. Also, it is possible to synthesise `for` loops.

It is both possible for the synthesis process to produce gates like confetti for some components while for others produce a gate level description that defies optimisation. For example, a synthesised register is much the same as the optimised version. A synthesised ALU can run to 37 pages of schematics—approximately 7 or 8 thousand gates—an area optimisation reduces this to about a thousand gates.

2.3 Optimisation

Once the synthesis process has completed the gate list generated must be optimised and mapped to the target technology. Target timing constraints such as clock cycle and input arrival and output setup times must be specified before optimisation is performed . The design can then be optimised, typically for speed, or area, or both. Further, optimisation can be specified to respect hierarchy in the design, or to flatten the design and perform global optimisation. Hierarchy is easily expressed within the VHDL model by the use of `BLOCK` statements. Optimisation for area will involve the following steps:

- Re-use of logic. Repeated logic will be recognised and eliminated. Logic such as adders will be re-used many times.
- Logic factoring. Serial implementations will be created by compressing random-logic into the minimum number of gates. Transduction is used to remove the redundant logic.
- Use of macro-cells. When possible the optimiser will substitute ASIC vendor macrocells which tend to be optimised for the target technology.

For speed optimisation, gates are examined in turn and replaced with equivalent logic with shorter gate delays (gate sizing). Capacitive loads are considered in each case and gate drives are sized accordingly. Additionally controllability factoring is performed. Controllability factoring determines which input signals to a section of logic contribute most to the final circuit output and ensures that these signals do not lie on the critical path for that logic.

Parts of the logic optimisation can be performed using the generic representation of the circuitry. This is particularly true of area optimisation. To map generic gates to ASIC vendor supplied macrocells or to reduce propagation delays for performance optimisation an optimiser will use signature analysis. This is analogous to peep-hole optimisation used in programming language compilers. Subcircuits in the design are selected and the truth tables for these peep-holes are constructed. Equivalent subcircuits provided in the technology libraries are then substituted using a technology rule database.

3 Field Programmable Gate Array ASICs

The market for FPGA ASICs, commonly referred to as just FPGAs, is expected to increase substantially over the next few years. Along with gate arrays ASICs, FPGA are expected to take a major share of the semiconductor market [12]. Further, the size and speed of these FPGA is increasing, making them more attractive for large designs. Consider also that programmable crossbar switches are now available [8, 3]. We thus have the potential to design reconfigurable system architectures for a wide variety of applications. These applications include but are not limited to, gate array ASIC emulation, prototyping gate array ASICs, but more importantly FPGAs offer viable production alternatives for smaller designs. For us in particular, they offer the potential for research into reprogrammable graphics architectures. These reprogrammable architectures will allow tradeoffs in terms of cost, size, speed, etc. and provide some of the flexibility that is currently enjoyed by microprocessor based architectures.

3.1 Estimating an FPGA design fit

Vendors tend to advertise the density of their FPGAs in terms of gate and flip flop counts. The gate counts are generally taken to be gate array equivalents. Quantifying FPGA density in this way is difficult and can be confusing for the design engineer. In general these gates and flip flop counts bear no resemblance to gate or flip flop utilisation rates. This problem is addressed by the the PREP [9] consortium whose goal is to clarify vendors claims using a selection of benchmarks. Utilisation depends on the FPGA architecture and on specific designs to be implemented.

Table 1 gives an estimate of how many how many Xilinx 4013 FPGAs it might take to implement a Gouraud shader ASIC similar in complexity to the IMAGE chip [5]. A detailed discussion on estimating a design fit for FPGAs can be found in [14]. Briefly, this involves counting up the macrocell usage, checking design I/O against FPGA I/O and analysing the designs delays along critical paths.

Logic Blocks	Xilinx XC4013 (CLB)
Decoders	280
CTU	12
FSM_controller	24
Registers	2556
Multiplexers	500
Add/Sub	141
ALU	800
Comparator	292
Total LMs	4478
Equivalent Gates	98516
Usable Gates	62692
FPGA Count	12

Table 1: Size estimates for a Gouraud shading architecture

For the Xilinx XC4013 FPGA we can see that the total number of CLBs required for this design is approximately 4478. The maximum number of CLBs for an XC4013 is 576. Therefore, we are into multiple FPGA designs. Each Xilinx CLB is equivalent to about 22 gates and from Xilinx benchmarks an average of 14 gates per CLB are used [2]. This is a utilisation rate of 64%. Considering that a gate is worth about 2.5 transistors, then we are looking for a device with $22x4478x2.5x0.64 = 157625$ transistors. This equates to approximately 62692 used gates. Using this utilisation rate we can see that this design will require about 12 XC4013 FPGAs. Manual place and route and other tricks may reduce this FPGA count. Further, by sometime next year the XC4020 should be available which should reduce the FPGA count down to about 8 or less.

This FPGA count analysis is based on our Pixel Parameter Interpolator (PPI) technology used for implementing the Gouraud shader [15]. As such the PPI datapaths are not optimised leading in some cases to excess CLB usage. The IMAGE chip has about 130000 transistors, so this design is about 20000 larger. Although the functionality is the same this increased size can be accounted for by the generalisation of the Parameter Register Unit and Parameter Formatting Unit of this PPI based design. Optimising the datapath bit widths could lead to a five XC4020 FPGA solution by sometime next year. We consider that a Gouraud shader implemented on five

FPGA ASIC is an acceptable alternative to a single masked gate array ASIC solution. Analysis of the designs critical paths suggest a 20 MHz system performance will be achieved.

4 Examples of Logic Synthesis and Optimisation

We illustrate here some early results of our logic synthesis and optimisation attempts. We use MGCs Falcon Framework version 8.2 with Autologic VHDL, Autologic Blocks and Autologic, etc. This is a very large concurrent engineering environment that not only provides the top down ASIC design approach discussed above but also with a multitude of other tools integrated into the design environment, e.g. PCB board tools, simulation tools, etc. The first example is part of our STEP architecture and the second is parts of our PPI architecture.

4.1 The Sussex Texture Processor

A current project is to design and build texture mapping hardware to augment the functionality of the IMAGE chip [5]. The specification of the Sussex TExture Processor) (STEP) requires mipmap based texture filtering [16]. The mipmap approach uses multiple copies of the texture image prefiltered to lower levels of detail. In our implementation, these levels are stored consecutively in texture memory. Computation of the texel address for each pixel is one of the tasks of the texture memory management unit (TMMU) in STEP. The inputs to the TMMU are texture coordinates supplied on two 10-bit busses, and the *level* of detail required, supplied on a 4-bit bus. The base address for the mipmap pyramid is stored internally by the TMMU, and is updated whenever the texture environment changes. The TMMU synthesises the address using the following equation:

$$a = b + f(l) + tg(l) + s \tag{1}$$

Where, a is the computed address, b is the base address for the mipmap, l is the required level of detail, $f()$ is a function returning the level offset, and s, t are texture coordinates in the range $0..f(l)$, $g()$ is a function returning the side-length of each level.

The level and texture coordinates are provided to the TMMU on each rising clock edge. The TMMU selects either the incoming level or a user supplied level-of-detail according to the state of the `ismipmap` control bit. The address synthesis proceeds as indicated in 1. The use of a barrel shifter to perform the multiply operation simplifies the process[1]. Also, true addition of the s coordinate is not necessary and can be replaced with a unit selecting bits from either s or $tg(l)$. The RTL VHDL code implementing this functionality is shown in figure 3 in appendix C.

4.1.1 Optimisation of the Offset Generator

The offset of any level in memory from the mipmap base address is found by summing together the sizes of all levels preceeding the required level. Our classification treats level 1 as a 512×512 image, level 2 as a 256×256 image, etc. In general level n has size $2^{10-n} \times 2^{10-n}$. The offset for level q is given as:

$$o = \sum_{i=1}^{q-1} 2^{2(10-i)} \tag{2}$$

When examined in a binary notation, these sums are seen to be strings of `01` repeated *level* times and padded to the right with zeroes to create a 20-bit word. The offset can therefore be

[1] Mipmap texture images are always $2^i \times 2^i$ in size

created by generating these words keyed by the incoming level value. The VHDL code shown in figure 3, see appendix A.2, shows this in the block `offsetGen`.

The synthesised logic for this block is most complicated. Large numbers of random gates are used to encode the offset from the level. The 400 or so gate design is far from optimal and is purely combinatorial. Area optimisation applied to the synthesised circuitry produces a much improved design, see figure 9 in appendix C. The optimiser spots that many bits are always zero and eliminates these from the logic; also that feedback can be used to further reduce the gate count. Speed optimisation might eliminate this feedback and increase the area size accordingly.

The major advantage of this approach for the design of the offset generation sub-unit is that larger mipmaps can be accomodated by changing the source code, and different distributions of texture image levels in memory can be explored quickly. The time consuming process of netlisting gates is totally eliminated.

4.2 Pixel Parameter Interpolator

Common to scan conversion is the need to incrementally linearly interpolate arbitrary vertex parameters across primitives such as triangles [13]. Work at Sussex has focussed on generalising this requirement so that any vertex parameter can be interpolated. This has led to the design of the PPI. Figure 10 In appendix D illustrates the second level hierarchy of the PPI. The first level is the chip or ASIC definition.

We can see that the PPI is composed of a control and timing unit, some decoding logic and a bank of general purpose Parameter Interpolation Units (PIU). Each PIU has the same structure, except the edge, window and depth units which have dedicated data paths. The rest are identical.

4.2.1 Parameter Interpolator Unit

It is useful to consider as an example, of the power of using VHDL descriptions over gate level descriptions, the PIU which is the core of the PPI. This has been implemented in the traditional way, using schematic capture and instantiation of library parts from our generic library called GENLIB. This library is available in other CAD tools besides Mentor Graphics and thus provides some degree of portability. However, the PIU for depth interpolation requires 48 bit data paths. Unfortunately, these data paths require multiplexers and an adder/subtractor. This leads to tedious gate level implementations because GENLIB does not have a parameterisable multiplexer. Thus, the 2:1 48 bit wide multiplexers have to be implemented by netlisting 48 2:1 single bit wide multiplexers. Even, more tedious is the fact that GENLIB only contains single bit full adders and half adders. Thus, the adder/subtractor has to be netlisted in the same manner. However, architectural tradeoffs make this process even more time consuming, e.g. ripple carry, carry look ahead, etc. imply more effort in creating models.

One solution is to invest in another library but this is costly. It is far simpler to model these components in VHDL. Appendix B details these netlisted components and the equivalent VHDL models in figures 4, 5, 6 and 7.

4.2.2 Parameter Formatting Unit

The Parameter Formatter Unit (PFU) has the job of formatting the arbitrary vertex parameters according to the corresonding identification code that accompanies the parameter. Some of these formatting operations are:

- Pass interpolated parameter

- Pass background colour
- Blend interpolated colour with background colour
- Clamp interpolated parameter to zero
- Clamp interpolated parameter to maximum positive number

This leads us to the design of a PFU which incorporates an ALU and a finite state machine (FSM) to generate the ALU opcodes, see figure 11 in appendix D. For the ALU we have selected the ALU '181 which is modelled by the Autologic Blocks library. Using this ALU we can see that we only need to generate 5 opcodes or states to drive this ALU. This means we only have to design a five state FSM. Mentor Graphics Autologic Blocks has a KISS compiler which enables rapid design of FSMs. The KISS FSM description is compiled into VHDL code which is further compiled by the VHDL compiler. However, because this FSM was reasonably simple it was written directly in VHDL using Mentor graphics synthesis guidelines. Appendix A.1 gives the VHDL code for this particular FSM.

Netlist statisics show the gate equivalence of the PFU after synthesis to be approximately 6500 gate equivalents and after optimisation to be approximately 2000 gate equivalents. Note that this area optimisation has been done on each instance in the PFU. Optimising instances into groups has not been done. This may easily lead to 1000 gate equivalents because, for example, the state machine has not been optimised with the pre-state decoding and ALU181 logic. We estimate that nine PFUs required for a Gouraud shader will fit on one or two FPGAs.

5 Conclusion

We have presented a new top down ASIC design with logic synthesis and optimisation strategy which is superior to the old traditional ASIC design strategy. It is superior because above all it enables the designer to get his product to market in a much shorter time scale due to the automation of the low level gate netlisting. This taken with all the benefits of targetting FPGA ASICs means the product is more versatile and less risky to produce.

We have adopted this top down strategy at Sussex. So far we have explored VHDL, logic synthesis and optimisation and remain impressed. With the examples shown we have demonstrated the power of VHDL, logic synthesis and optimisation. We believe this is the way forward and will provide many benefits for graphics ASIC designers.

6 Acknowledgments

This work has been funded partly by the European Commission through the Esprit projects Spirit and Spectre. We wish to acknowledge all past and present members of the Spirit and Spectre consortiums for their valuable contributions to our work.

References

[1] *The VHDL Reference Manual.*

[2] ACTEL. *The FPGA Design Guide.*

[3] APTIX. *Aptix the Programmable Interconnect Company Data Book.*

[4] Peter J. Ashenden. The VHDL cookbook. ftp from chook.adelaide.edu.au (129.127.8.8), directory pub/VHDL-Cookbook (as bin-hex or apple PostScript) or bears.ucsb.edu in directory pub/VHDL, Dept. Computer Science, University of Adelaide, South Australia, July 1990.

[5] Graham Dunnett, Martin White, Paul Lister, Richard Grimsdale, and France Glemot. The IMAGE chip for high performance 3D rendering. *IEEE Computer Graphics and Applications*, 12(6):41–52, November 1992.

[6] S. R. Evans, P. F. Lister, R. L. Grimsdale, and A. D. Nimmo. *The AIDA Advanced Image Display Architecture.*

[7] H. R. Finch, A. Agate, P.F. Lister, and R. L. Grimsdale. *A Multiple Application Graphics Integrated Circuit MAGIC II.*

[8] I-CUBE. *IQ160 Field Programmable Interconnect Device Data Book.*

[9] David Manners. Programmable logic devices are compared for speed and density. *Electronics Weekly*, April 1993.

[10] Douglas L. Perry. *VHDL.* McGraw-Hill, 1991.

[11] Silicon Compiler Systems. *Genesil Designer Volumes I & II.*

[12] TI. *Texas Instruments FPGA Applications Handbook*, 1993.

[13] Steve Upstil. *The RenderMan Companion, A Programmers Guide to Realistic Computer Graphics.* Addison Wesley, 1989.

[14] M White, G. Dunnett, P. Lister, and R Grimsdale. Field programmable gate arrays—computer graphics imaging.

[15] Martin White. Deliverable of spirit task gh.s8 aceleration of shading and texturing. Technical report, The University of Sussex, 1993.

[16] L. Williams. Pyramidal parametrics. *ACM Computer Graphics*, 17, July 1983.

A Example VHDL Models and Schematics

A.1 VHDL Finite State Machine

A.2 STEP Behavioural Description

B Comparison of PPI GENLIB gate level schematics and VHDL equivalents

C Optimised Offset Generator

D Pixel Parameter Interpolator

```
--Finite State Machine controller for the '181 ALU.
--Written by Martin White 2nd Aug 1993
--
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_1164_extensions.all;

--
-- Written by Martin White at Mon Aug 2nd 1993
--
ENTITY fsm IS
  PORT (
      depth    : IN        std_ulogic ;
      blend    : IN        std_ulogic ;
      one      : IN        std_ulogic ;
      zero     : IN        std_ulogic ;
      clk      : IN        std_ulogic ;
      reset    : IN        std_ulogic ;
      shift_r  : OUT       std_ulogic ;
      mode     : OUT       std_ulogic ;
      colour   : IN        std_ulogic ;
      edge     : IN        std_ulogic ;
      window   : IN        std_ulogic ;
      opcode   : OUT
        std_ulogic_vector(3 DOWNTO 0) ;
      hidden   : IN        std_ulogic ;
      pineda   : IN        std_ulogic
      ) ;
END fsm ;

ARCHITECTURE controller OF fsm IS
  type states is (s0, s1, s2, s3, s4);
  --Pass B data
  signal state : states := s0;
  --Default to pass B data
  signal next_state : states := s0;
BEGIN

  clock:
  PROCESS (clk,reset)
  BEGIN
    IF reset = '1' THEN
      state <= s0;
    ELSIF clk'event and clk = '1'
          and clk'last_value = '0' THEN
      state <= next_state;
    END IF ;
  END PROCESS clock ;

  state_transitions:
  PROCESS (state, one, zero, pineda, blend,
           hidden, edge, window, depth, colour)
  BEGIN
    next_state <=  s0 ;
    CASE state IS
      WHEN s0 =>
        IF (pineda = '1' and (edge = '1' or window = '1'
                    or depth = '1' or colour = '1')) THEN
          --Pineda = '1' when we are interpolating normal triangles, as
          --as opposed to hiddenline triangles.  It is also valid when
          --the edge of an hiddenline triangle is reached, under these
          --conditions the current interpolated edge, window, depth or
          --colour is passed.
          next_state <= s0;
        ELSIF (pineda = '1' and edge = '0' and window = '0'
                    and depth = '0' and colour = '0') THEN
          --Pass arbitrary interpolated parameter
          next_state <= s0;
        ELSIF (hidden = '1' and colour = '1') THEN
          --Hiddenline triangle and pixel inside so pass background colour
          next_state <= s1;
        ELSIF (zero = '1') THEN
          --Parameter or colour overflowed
          next_state <= s2;
        ELSIF (one = '1') THEN
          --Parameter or colour underflowed
          next_state <= s3;
        ELSIF (blend ='1' and colour = '1') THEN
          --Do a 50% blend of interpolated colour and preloaded background colour
          next_state <= s4;
        END IF ;
      WHEN s1 =>
          --Similar code to state zero
      WHEN s2 =>
          --Similar code to state zero
      WHEN s3 =>
          --Similar code to state zero
      WHEN s4 =>
          --Similar code to state zero
    END CASE ;
  END PROCESS state_transitions ;

  state_outputs:
  PROCESS (state)
  BEGIN
    CASE state IS
      WHEN s0 =>
        opcode  <= "1010";
        mode    <= '1';
        shift_r <= '0';
      WHEN s1 =>
        opcode  <= "1111";
        mode    <= '1';
        shift_r <= '0';
      WHEN s2 =>
        opcode  <= "1001";
        mode    <= '1';
        shift_r <= '0';
      WHEN s3 =>
        opcode  <= "1100";
        mode    <= '1';
        shift_r <= '0';
      WHEN s4 =>
        opcode  <= "1001";
        mode    <= '1';
        shift_r <= '1';
    END CASE ;
  END PROCESS state_outputs ;
END controller;
```

Figure 2: VHDL finite state machine controller for a PPI formatter

```
-------------------------------------------------------------
-- VLSI and Computer Graphics Research Group 1993.
-- University of Sussex.
-- All rights reserved.
-------------------------------------------------------------
--
--  Model Title:  coordAdjust layer
--  Date Created: 25/03/93
--  Author:       Graham Dunnett
--
-------------------------------------------------------------
LIBRARY mgc_portable ;
USE mgc_portable.qsim_logic.ALL ;
ENTITY as IS
  PORT (
      s,t       : IN  qsim_state_vector(9 DOWNTO 0) ;
      baseaddr  : IN  qsim_state_vector(19 DOWNTO 0) ;
      clk       : IN  qsim_state ;
      userlevel : IN  qsim_state_vector(3 DOWNTO 0) ;
      level     : IN  qsim_state_vector(3 DOWNTO 0) ;
      ismipmap  : IN  qsim_state ;
      addr      : OUT qsim_state_vector(19 DOWNTO 0)
      ) ;
END as ;
ARCHITECTURE rtl OF as IS
  SIGNAL theLevel     : qsim_state_vector(3 DOWNTO 0) ;
  SIGNAL levelin      : qsim_state_vector(3 DOWNTO 0);
  SIGNAL sin, tin     : qsim_state_vector(9 DOWNTO 0);
  SIGNAL shiftamount  : qsim_state_vector(3 DOWNTO 0);
  SIGNAL shift8t      : qsim_state_vector(19 DOWNTO 0);
  SIGNAL bando        : qsim_state_vector(19 DOWNTO 0);
  SIGNAL offset,a     : qsim_state_vector(19 DOWNTO 0);
  SIGNAL selector     : qsim_state_vector (9 DOWNTO 0) ;

BEGIN
--
---- latch data on rising edge
--
  latchin:
  PROCESS (clk)
  BEGIN
    IF (clk = '1' AND clk'last_value = '0' AND clk'event) THEN
      sin <= s; tin <= t; levelin <= level;
    END IF;
  END PROCESS latchin ;
--
---- ismipmap selects which of the two level inputs to use
--
  doLevel:
  PROCESS (userLevel, levelin, ismipmap)
  BEGIN
    CASE ismipmap IS
      WHEN '0' =>       -- not mipmapping, use user supplied level
        theLevel <= userLevel;
      WHEN OTHERS =>
        theLevel <= levelin;
    END CASE ;
  END PROCESS doLevel ;

--
---- decode level into a shift amount shiftamount = 10-level
--
  shiftdecode :
  BLOCK
  BEGIN
    shiftdecodeprocess :
    PROCESS ( theLevel )
    BEGIN
      CASE theLevel IS
        WHEN "0000" => shiftamount <= "1010";
        WHEN "0001" => shiftamount <= "1001";
        etc
        WHEN "1001" => shiftamount <= "0001";
        WHEN OTHERS => shiftamount <= "0000";
      END CASE ;
    END PROCESS shiftdecodeprocess ;
  END BLOCK shiftdecode ;

--
---- We need to shift t left by between 0 and 10 positions.
---- To do this we shift by 1, 2, 4, and/or 8 in the necessary
---- combinations.
--
  myshifter:
  BLOCK
    -- barrel shift code deleted for briefness
  END BLOCK myshifter;
--
---- decode the level into the texture image offset
--
  offsetGen:
  BLOCK
  BEGIN
    offsetGenProcess :
    PROCESS ( theLevel )
    BEGIN
      CASE theLevel IS
         WHEN "0000" => offset <= "00000000000000000000";
         WHEN "0001" => offset <= "00000000000000000000";
         WHEN "0010" => offset <= "01000000000000000000";
         WHEN "0011" => offset <= "01010000000000000000";
         WHEN "0100" => offset <= "01010100000000000000";
         WHEN "0101" => offset <= "01010101000000000000";
         WHEN "0110" => offset <= "01010101010000000000";
         WHEN "0111" => offset <= "01010101010100000000";
         WHEN "1000" => offset <= "01010101010101000000";
         WHEN "1001" => offset <= "01010101010101010000";
         WHEN OTHERS => offset <= "01010101010101010100";
      END CASE ;
    END PROCESS offsetGenProcess ;
END BLOCK offsetGen;

--
---- Add the base address to the offset which varies with the level
--
  baseAndOffsetAdd:
  PROCESS ( baseAddr, offset )
    VARIABLE answer : qsim_state_vector(19 DOWNTO 0);
  BEGIN
    answer := baseAddr + offset;
    bando <= answer(19 DOWNTO 0);
  END PROCESS baseAndOffsetAdd ;

--
---- We want to add v*N + u. v*N is available as shift8t and u is sin.
---- Now, N = 2 \^ i and u < N. Therefore (v*N) has a string of zeros in
---- its ls bits which u will replace. The addition can therefore be
---- implemented with muxes for each bit selecting either u(j), v(j).
--
  selectorBlock :
  BLOCK
  BEGIN
    selectorBlockProcess :
    PROCESS ( theLevel )
    BEGIN
      CASE theLevel IS
        WHEN "0000" => selector <= "1111111111"; -- all bits from u
        WHEN "0001" => selector <= "0111111111";
        etc
        WHEN "1001" => selector <= "0000000001";
        WHEN OTHERS => selector <= "0000000000";
      END CASE ;
    END PROCESS selectorBlockProcess ;
  END BLOCK selectorBlock ;

  coordBlend :
  PROCESS ( selector, shift8t , sin )
  BEGIN
    FOR j IN 0 TO 9 LOOP
      IF selector(j) = '1' THEN
        a(j) <= sin(j);
      ELSE
        a(j) <= shift8t(j);
      END IF ;
    END LOOP ;
    a(19 DOWNTO 10) <= shift8t(19 DOWNTO 10);
  END PROCESS coordBlend ;

--
---- Compute the addr from partial results a and bando
--
  addrGen:
  PROCESS ( a , bando )
    VARIABLE result : qsim_state_vector(19 DOWNTO 0) ;
  BEGIN
    result := a + bando;
    addr <= result(19 DOWNTO 0);
  END PROCESS addrGen ;
END rtl;
```

Figure 3: VHDL Code For The Offset Generator

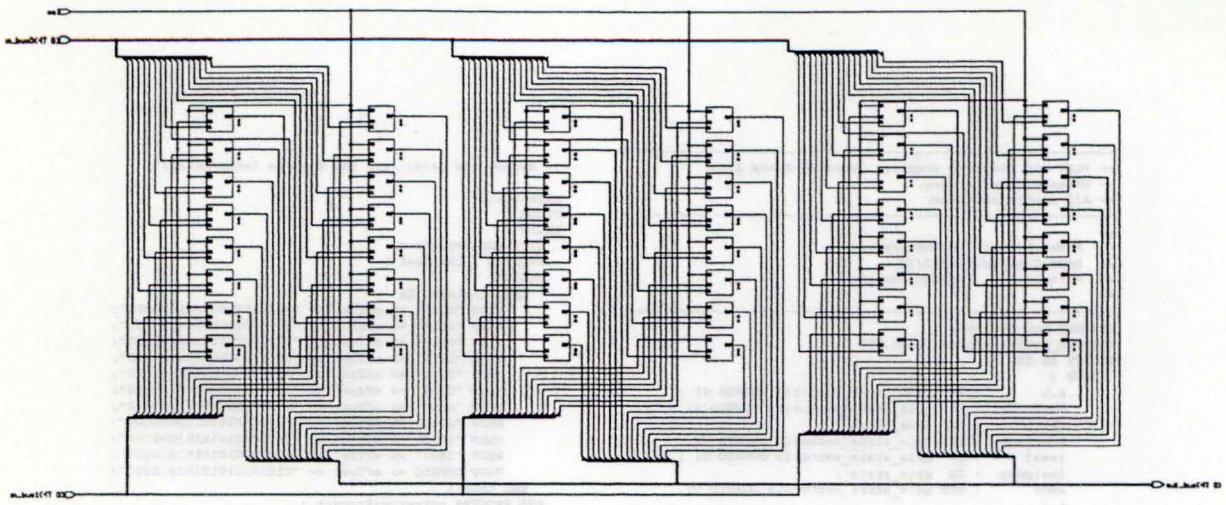

Figure 4: Schematic of a 2:1 48 bit wide multiplexer based on Genlib parts

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_1164_extensions.all;

--
-- Written by LL_to_VHDL at Fri Jul  2 10:02:05 1993
-- Parameterized Generator Specification to VHDL Code
--
--
-- LogicLib generator called: MULTIPLEXER
-- Passed Parameters are:
--   tinst name = mux0
--   parameters are:
--      type = SIMPLE
--      W = 48
--      numin = 2
--      SW = 1
--      bus_mask = 0
--      comp_out = NO
--

-- mux0 Entity Description
entity mux0 is
   port(
      IN0: in std_ulogic_vector(47 downto 0);
      IN1: in std_ulogic_vector(47 downto 0);
      SEL: in std_ulogic_vector(0 downto 0);
      DOUT: out std_ulogic_vector(47 downto 0)
   );
end mux0;

-- mux0 Architecture Description
architecture rtl of mux0 is

begin
   mux0_Process: process(IN0,IN1,SEL)
      variable iaddress : integer range 0 to 1;
      variable state : std_ulogic_vector(47 downto 0);
   begin
         iaddress := to_Integer('0' & SEL,0);
         case iaddress is
               when 0 =>
                  state := IN0;
               when 1 =>
                  state := IN1;
               when others =>
                  state := (OTHERS => 'X');
         end case;

      -- Assign outputs
      DOUT <= state;
   end process mux0_Process;
end rtl;
```

Figure 5: VHDL version of 2:1 48 bit wide multiplexer

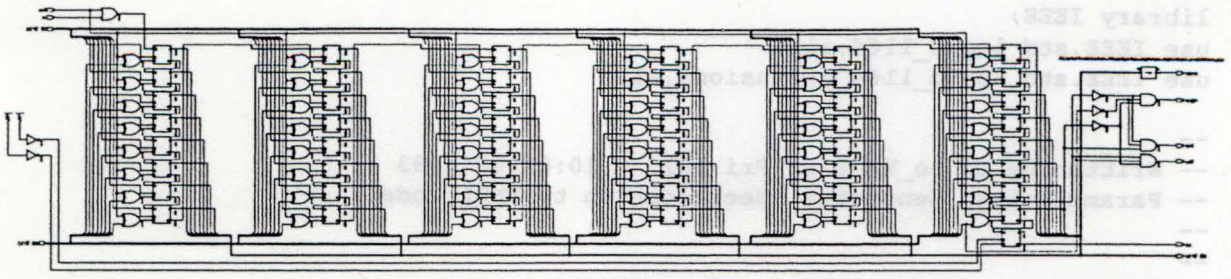

Figure 6: Schematic of a 48 bit wide adder/subtractor based on Genlib parts

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_1164_extensions.all;

--
-- Written by LL_to_VHDL at Tue
-- Jul 27 15:45:57 1993
-- Parameterized Generator
-- Specification to VHDL Code
--
--
-- LogicLib generator called:
-- ARITHMETIC
-- Passed Parameters are:
--   tinst name = add_sub
--   parameters are:
--      type = ADDSUB
--      W = 48
--      look = 16
--      carryin = YES
--      carryout = YES
--      ov = YES
--      lt = YES
--      gt = YES
--      eq = NO
--

-- add_sub Entity Description
entity add_sub is
   port(
      A: in std_ulogic_vector
               (47 downto 0);
      B: in std_ulogic_vector
               (47 downto 0);
      D: out std_ulogic_vector
               (47 downto 0);
      CIN,SUB: in std_ulogic;
      COUT,GT,LT,OV: out std_ulogic
   );
end add_sub;

-- add_sub Architecture Description
architecture rtl of add_sub is
   signal pre_D  : std_ulogic_vector
               (48 downto 0);
   signal pre_OV : std_ulogic;
   signal pre_EQ : std_ulogic;
   signal pre_LT : std_ulogic;
begin
   ARITHMETIC_Process:
   process(A,B,CIN,SUB)
      variable fct_out
        : std_ulogic_vector(48 downto 0);
      variable a_ext,b_ext
        : std_ulogic_vector(48 downto 0);
      variable carry_ext
        : std_ulogic_vector(1 downto 0);
      variable msb : integer;
   begin
      -- zero extend inputs to
      -- include carry bit
      a_ext := '0' & A;
         if (SUB = '1') then
            b_ext := '0' & not B;
         else
            b_ext := '0' & B;
         end if;
      carry_ext := '0' & CIN;

      -- ADDSUB
      fct_out := a_ext + b_ext
                       + carry_ext;

      -- Assign to signal for use
      -- outside process
      pre_D <= fct_out;

      -- Calculate overflow bit
      if (a_ext(47) = b_ext(47)
               and fct_out(47)
               = not a_ext(47)) then
         pre_OV <= '1';
      else
         pre_OV <= '0';
      end if;
   end process ARITHMETIC_Process;

   -- Assign the outputs
   D <= pre_D(47 downto 0);

   -- Assign flags
   COUT <= pre_D(48);
   pre_EQ <= '1'
        when (pre_D(47 downto 0)
        = "000000000000000000000000
        000000000000000000000000")
                else '0';
   pre_LT <= (pre_OV xor pre_D(47));
   GT <= not pre_EQ and not pre_LT;
   LT <= pre_LT;
   OV <= pre_OV;
end rtl;
```

Figure 7: VHDL version of a 48 bit wide adder/subtractor

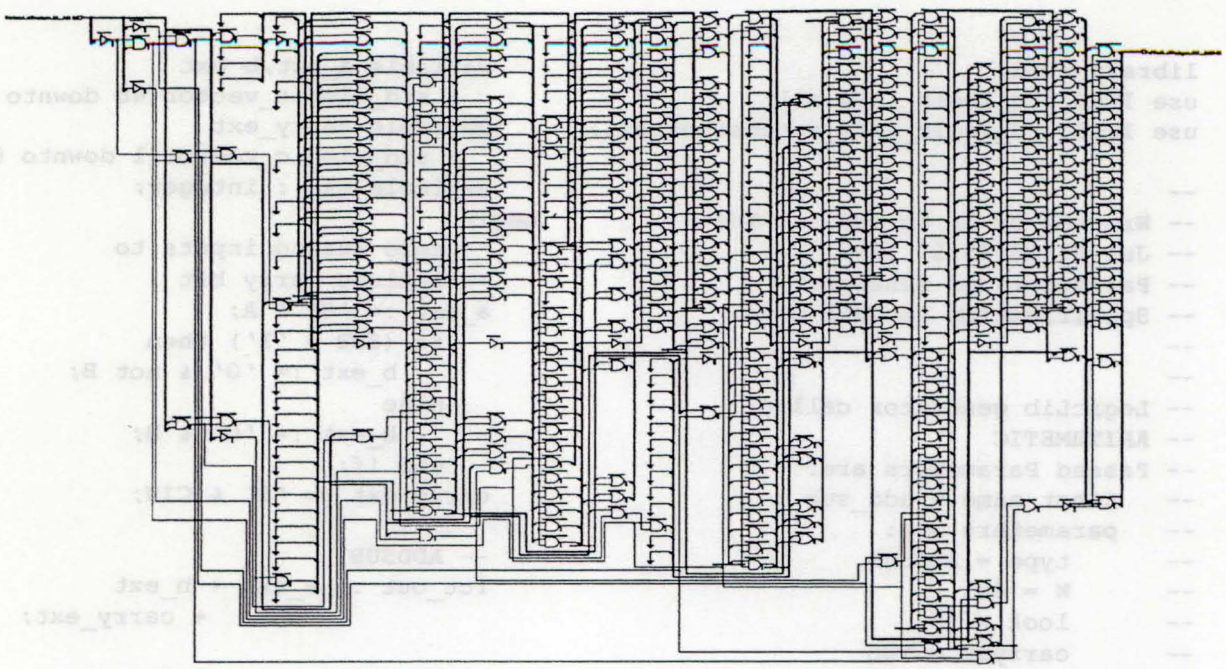

Figure 8: Synthesised Offset Generator

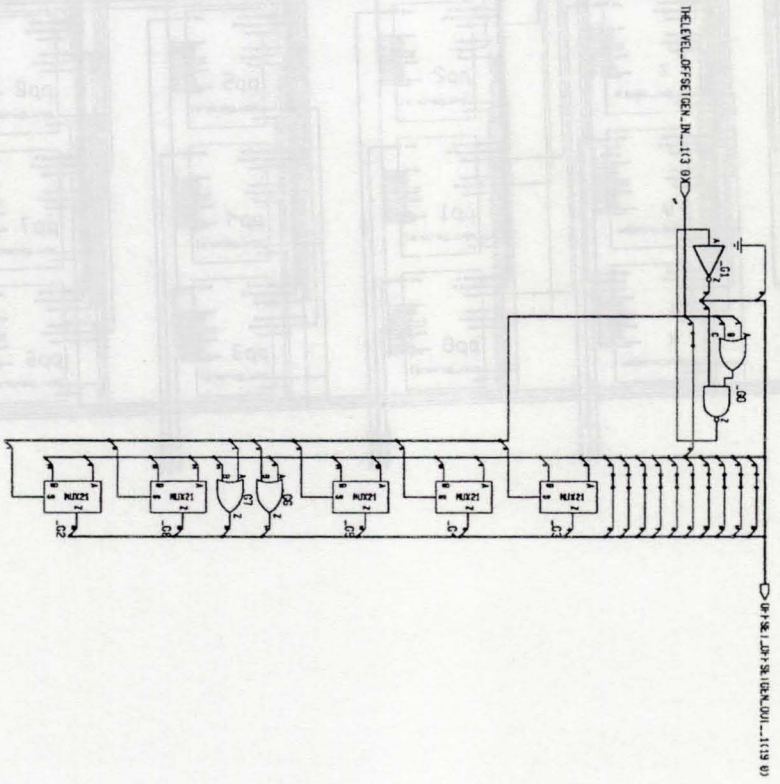

Figure 9: Optimised Offset Generator

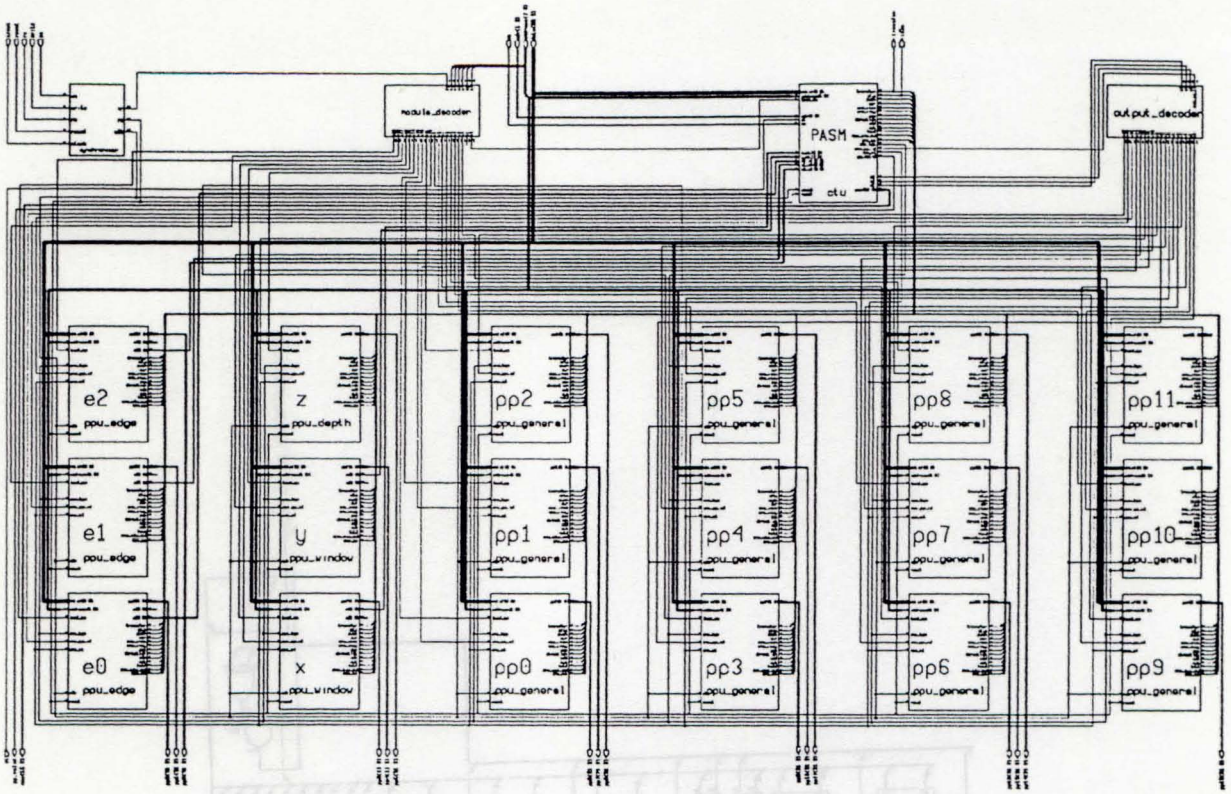

Figure 10: Pixel Parameter Interpolator 2nd level hierarchy

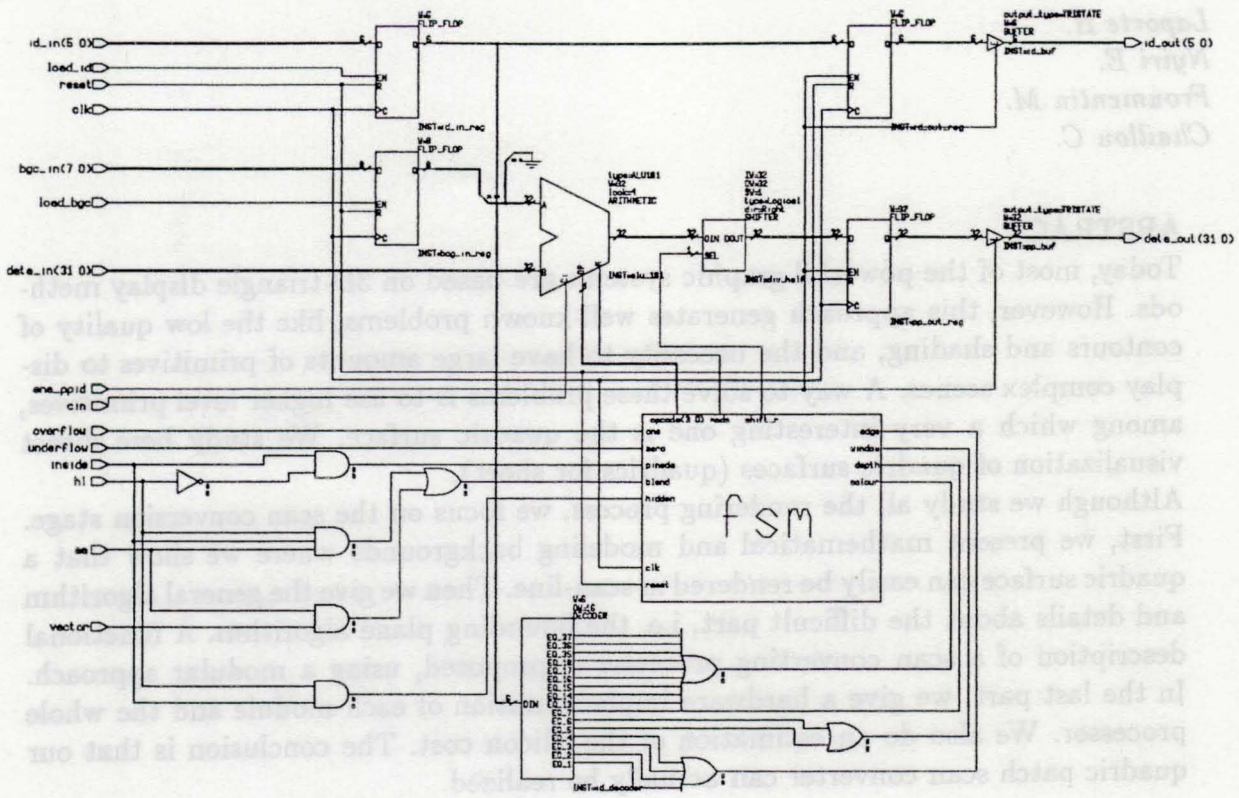

Figure 11: Parameter Formatting Unit