1

A VLSI Design Strategy for Graphics

A.D. Nimmo, P.F. Lister, and R.L. Grimsdale

The tools available for ASIC design now offer the features and functionality necessary to permit ideas to be realised in silicon in a relatively short period of time. This paper introduces work undertaken at Sussex University intended to lead to a more complete VLSI Design Strategy, using ECAD packages provided by Mentor Graphics. In particular, it focuses on the use of Behavioural simulation tools and includes a worked example.

1. Introduction

VLSI Design for Graphics is different from that required for standard graphics processors, for example, the TMS34010 Graphics Processor [1]. An ASIC for graphics attempts to optimise operations, like a dot product, by careful consideration of the data types and organisation, the data paths and arithmetic units required, the type of control, possible parallel operations and storage types and memory allocation. The ASIC parts being designed at the University of Sussex are to perform database, geometry and display solutions at very high speeds for applications like flight simulator displays or very high performance workstation displays and coprocessors.

The classic example of a graphics ASIC implemented as *Algorithms in Silicon* was used in the IRIS Workstation [2] from Silicon Graphics, Inc., where one VLSI circuit, the Geometry Engine [3], essentially a four-component vector floating point processor is used to accomplishing three basic graphics operations — matrix multiplication, geometric clipping and mapping to screen coordinates. A similar approach was taken by Seillac Co., Ltd. for the Seillac-7 Display Station [4] where a mixture of standard microprocessors, bit-slice processors, multiplier circuits and custom VLSI were employed. An indication of where specialist processor parts are heading is shown by Silicon Graphics Iris GT Workstation [5] whose geometry subsystem employs five floating-point engines based on Weitek 3332 floating-point units, controlled by propriety i.c.'s.

An Approach to VLSI design must remain flexible to adapt to project requirements but should always provide a coherent framework. Ideally all the research, algorithm and code

development, data generation and simulation work, ASIC development and final hardware development should be performed using an integrated toolset. Although this is not possible, some common data transfer route should be found to facilitate data transfer between the different stages of design.

A flexible design framework which all members of a team adhere to aids cross verification, notification and completion of the necessary tasks. This is important when a design is to be modified so parallel activities can be updated and checked for data integrity.

The development of VLSI designs specifically for graphics encompasses several aspects of processor and system design from a variety of fields:

- Behavioural Specification of the Design.
- Architecture Design, including control options such as:
  - Dedicated (e.g. PLA).
  - Writable Control Store (e.g. Ram Store).
  - Writable Instruction Set Computer (WISC).
  - Complex Instruction Set Computer (CISC).
  - Reduced Instruction Set Computer (RISC).
  - Very Long Instruction Word (VLIW).
- Simulation — behavioural and functional.

2. The Choice of Design Tools

The chosen tools should be user-friendly (with a short response time), exhibit a consistent interface between the tools in a given package and should offer a high degree of comparability with similar tools, thus ensuring standard file interchange facilities.

There must be an awareness of the different silicon implementation routes available — these include Gate Array, Standard Cell and Silicon Compilation. This decision is actually coming to be of less importance because several integrated circuit manufacturers can now offer an implementation-independent design route — initial chips are produced with a gate array for evaluation, then the design can be transferred to standard cell, perhaps in a more dense technology to reduce die size and cost while improving speed and yield. For the majority of designers though, a decision, based on economics, has to be made early on in the design cycle.

Whilst the implementation methods described here and the tools used to create the VLSI part are sufficiently accurate and capable of providing a high quality integrated circuit, the end product will only be as good as the initial design work. It is crucial, for any design, to perform extensive simulation and testing of the circuit(s).

The design tools used will have some form of simulator, to check timing and interactively ensure the correct manipulation of data. Although simulation of a circuit may prove it to

be functional, it is only from experience or by some method of prototyping candidate architectures that the designer can be sure the optimal design has been chosen, given the limitations of the tools and manufacturing process.

3. A Design Framework

Our design strategy is shown in Figure 1. This is intended to provide a clear framework for the designer while remaining flexible enough to adapt with project requirements.

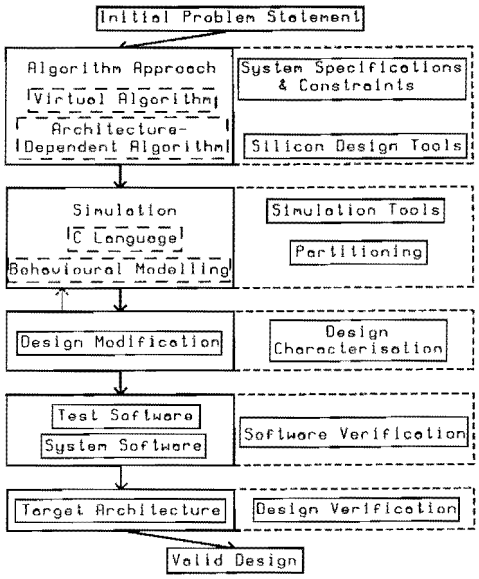

Figure 1: A Design Strategy.

3.1. The Initial Problem Statement

In the following sections it will be assumed that an algorithm performs a single well-defined function [6]. The idea of an *initial problem statement* introduces the requirements of the intended system — this might be to take data from a given 3-d database of polygons and display these on a monitor. There are several algorithms to be performed in this task [7], for example back facing surface removal, view transformation, clipping, perspective projection and gradient calculations. Already a hierarchical structure has developed — breaking a problem down into simpler units at this stage lends itself to structured development, which can be used in the following design stages in terms of both architecture design and simulation.

3.2. The Algorithm

3.2.1. The Virtual Algorithm. The Virtual Algorithm is the stage of development where the computational steps to be performed have been fully determined, e.g. pseudocode. If is system and architecture-independent.

3.2.2. The Architecture-Dependent Algorithm. This algorithm represents the association of the Virtual Algorithm steps with a particular architecture†.

3.2.3. Source Code Development.

3.2.3.1. Hierarchical Development. Top-down programming allows the designer to specify the overall structure of a program while hiding unnecessary detail. The lowest level of which a program is written depends on how much information is required. For example, a routine requires that two numbers are multiplied together yielding a result in a specified time. For simplicity, the multiply statement of the language in use could be used. However, this provides little information — information which would be needed for control purposes. This information would be made available if the programmer implemented his own multiply routine, yielding additional information at the expense of longer program development. Therefore it is suggested that source code be developed in several stages, at increasing levels of complexity. Starting out with this approach permits substitution of 'simple' operations with function calls and parameters to the necessary routines, obviating the need to continuously re-write code. The level where no further programming is necessary‡ should reflect the fulfilment of a set of data needed for testing and confirming the correct operation of the circuit. The method outlined here relies soley on the availability of a computer and a programming language — a way of integrating this method with the Mentor Graphics CAD system will be discussed later.

This strategy is particularly suited to graphics operations intended for ASCI's with a modular architecture and may also be used as the basis for related areas, such as a database processor, used for manipulation and ordering of graphical data.

3.2.3.2. Function Partitioning. Currently there are no widely available tools to automate function partitioning at either the system level or chip level. Deciding which system and chip architectures should be used is usually performed manually — often relying on experience. This is usually accomplished by the method shown in Figure 2.

There is no formal way of predicting how many iterations will be needed to produce a satisfactory architecture. Several ways have been identified to speed up this process.

---

† architecture refers to the particular level at which the designer is working — the system architecture or the chip architecture.

‡ this does not imply that the stage outlined here should comprise one task — it is iterative in nature.

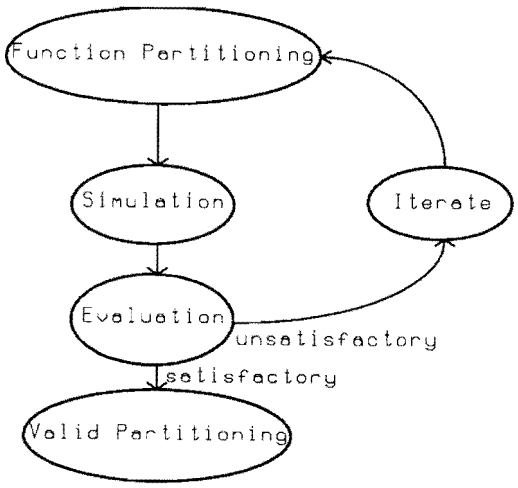

Figure 2: Manual Partitioning.

1. The use of graphics-based ECAD tools provide a standard environment in which to develop and refine an architecture according to Figure 2. This still relies on much manual work, but the ease with which functional blocks, whether standard parts or user-defined, can be manipulated and connected should decrease the development cycle. There will often be some form of hardware description language or behavioural language functionality built into the ECAD tools, for example, Mentor Graphics Behavioural Language Modelling (BLM) software. BLM allows the functionality of a block, in terms of inputs and outputs, to be described in C or Pascal
2. Automation of the partitioning process can be compared to optimising language compilers — data flow and/or control flow information can be taken from a high level specification or description of a circuit. The choice of language in which to write this specification could perhaps be a Functional language†, in order to control the operations per piece of data‡, or a language like C to yield information about a control strategy§. Output could be textual or graphical.
3. A.I. and Knowledge-based techniques could be used similarly to the automatic process given above, except a database of information would be built (and appended) and used to generate more 'intelligent' structures. There is much scope for investigation in the two automated approaches.

---

† for information, see [8].

‡ data flow optimisation.

§ control flow optimisation.

C is used at Sussex University for source code development because it is highly portable, it has the ability to create and manipulate data structures to bit level and is the language of choice for BLM. One deficiency is its inherent serialism, although this is overcome with BLM software tools. Candidates suitable for parallel system development include Occam [9] and Ada[10].

3.3. Design Simulation

3.3.1. The importance of Simulation. Simulation is the process of representing or modelling the behaviour of systems, on a computer, to record system responses. It involves designing a model of the system and conducting experiments on that model for the purpose of evaluating various strategies for the operation, and understanding the behaviour, of the system. In all areas of design, ECAD, MCAD or others, simulation allows the designer to test the operation of, for example, an electrical circuit, or the strains on a road bridge, or a growth projection for a population study.

Simulation of VLSI circuits can be used to predict performances, test various architectures, produce test data, help implement instruction sets, and other tasks which previously would have been done by breadboarding discrete versions of each design iteration.

The tools available for simulation can usually operate at several different levels depending on the results needed and the time available, although for something as complex and expensive as a custom chip set there is a definite need to identify suitable strategies for the design and simulation aspects. Identification of the processes and data required at each stage of a design is essential together with constant cross-checking and verification between the separate development paths of a project. This is shown in the design and simulation strategy of aq 16-bit processor by the Xerox Corporation [11] where specification and architecture work were performed by Xerox engineers and expertise on MOS implementation was done by Silicon Compilers Inc., suppliers of the GENESIL Silicon Compiler.

3.3.2. Simulation Techniques. There are several steps in the simulation of a design. Initial simulation using a programming language has been discussed, while the simulation provided by the silicon design tools used will often be to complex at this stage in development. A suggested intermediate step is to employ some form of hardware description language or an equivalent. The equivalent method being used at Sussex is based on Mentor Graphics BLM software. (It is possible to incorporate this simulation step and the task of function partitioning into one stage). This allows the designer to specify the behaviour of a design in C or Pascal and also to include as many checks throughout the simulation, as part of the simulation, as required. As much or as little architectural information can be included as is deemed necessary depending on what results and responses are required at a given stage. These tools can often make use of all

the workstation capabilities — using a graphics node to display simulation results, using interprocess communication and mailbox facilities to speed up operations through the use of parallelism at the workstation (UNIX) process level and using other nodes to simulate other parts of a complete system.

The use of the BLM tools is a method of prototyping under investigation at Sussex — it shows promise for simulating anything from the system level through board level to chip level. The use of these tools at this abstract level is not believed to be widespread throughout industry [12], probably due to the initial learning involved and lack of support from i.c. manufacturers.

4. Behavioural Language Modelling — A Worked Example

A worked example of a BLM is included, from specification to implementation, to show the relative simplicity and style in which they may be constructed. A relatively simple 32×32 bit multiplier has been chosen to illustrate several of the features available.

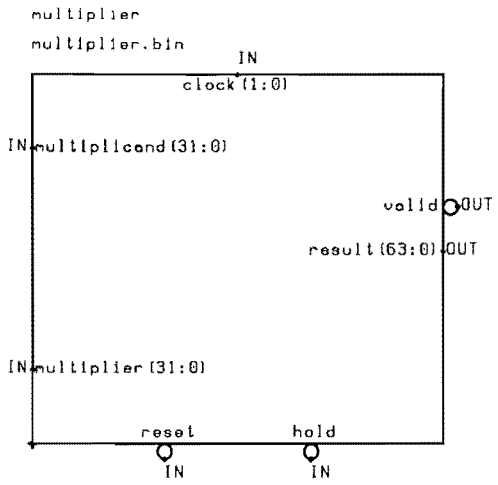

Figure 3: The Multiplier Symbol.

4.1. Specification

The basic specification decided upon was as follows,

1. 32×32 bit unsigned multiply, 64 bit result.
2. Asynchronous reset operation requires 3 clock cycle.
3. Operates with overlapping or non-overlapping 2-ϕ clock.

4. Inputs and calculations must occur when ϕ-0 set., outputs must occur when ϕ-1 set.
5. Has a VALID output (active low) to indicate a valid result.
6. Has an optional asynchronous HOLD (active low) input.

4.2. Symbol Creation and Schematic Capture

The Multiplier symbol was created using Mentor Graphics SYMED (Symbol Editor). It is shown in Figure 3. The design schematic is shown in Figure 4.

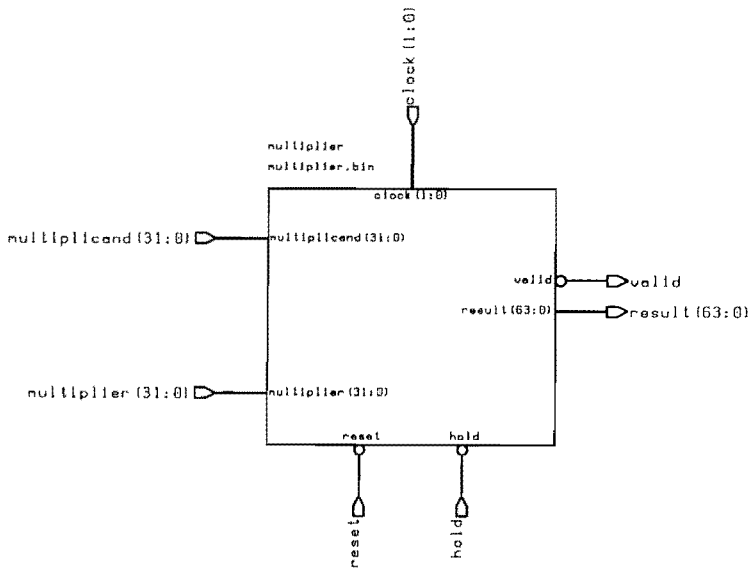

Figure 4: The Multiplier Schematic.

Symbol creation and schematic capture are standard features of most ECAD packages. These allow the designer to create a legible drawing. Together with hierarchical capture, a complex design can be shown more simply on several sheets†.

Of note are the bus naming convention (<bus_name>(msb:lsb)) and the use of connectors at the ends of busses and single nets to interface the real world.

† sheets are virtual pages used by the schematic capture package.

4.3. Code Development

The simulator is event driven — a change on any net connected to a pin or bus will cause the code associated with that pin or bus to be executed. This code performs the required internal operations and will often require to output values on the output pins. If multiple instances of the same block are used on a schematic, only one copy of the executable code is used therefore if internal data specific to given blocks is needed memory must be allocated for every instance of that block. Commented segments of code from the MULTIPLIER example are given on the following pages.

```
/* structure for 64-bit result */
typedef struct {
        unsigned long lsw;
        unsigned long msw;
} VLONG, *VLONG_PTR;
/* structure for user_data area */
typedef struct {
        unsigned long multiplicand;
        unsigned long multiplier;
        VLONG         result;
        short  p_phase_0;
        short  p_phase_1;
        short  clock_counter;
        short  latency_counter;
        short  reset_counter;
        BOOLEAN reset_flag;
        BOOLEAN hold_flag;
        BOOLEAN valid_flag;
        BOOLEAN result_flag;
        BOOLEAN multiplicand_flag;
        BOOLEAN multiplier_flag;
} state_info_t;
static state_info_t *state_ptr;


void multiplier_allocate()
{
        long struct_len;

        /* allocate storage for internal multiplier values */
        struct_len = sizeof(state_info_t);
        qsim_allocate)&struct_len, &state_ptr);

        /* structure NOT initialised — this will be done on reset */

        /* set instance pointer to user_data area */
        qsim_instance_ptr->user_data_area = (char *)state_ptr;
        return;
}
```

This allocates sufficient memory for each instance and allows each instance to access only its own user data. The declared structure acts as a template for the data.

The only external events of interest are the clock signals, RESET and HOLD. RESET and HOLD set flags within the user data area while the clock signals, together with a counter, call the necessary functions at the correct clock cycle. N.B. This is only one method of ensuring correct operation, other methods exist.

```
void multiplier_clock()
{
        /* phase_0 — input, calculate */
        /* phase_1 — output */
        short int phase_0, phase_1;
        /* set instance pointer to correct user_data area */
        state_ptr = (state_info_t*) qsim_instance_ptr->user_data_area;
        /* input value on clock[0] pin */
        phase_0 = qsim_con_value[(*(qsim_instance_ptr->multiplier_l_clock))->bits[0]];
        /* input value on clock[1] pin */
        phase_1 = qsim_con_value[(*(qsim_instance_ptr->multiplier_l_clock))->bits[1]];
        /* only do if either value == QSIM_ONE */
        if ((phase_0 == QSIM_ONE) || (phase_1 == QSIM_ONE)) {
                /* test input values — takes care of overlapping 2-phase clock */
                if (phase_0 == phase_1) {
                        /* find which phase has changed since previous clock event */
                        if ((phase_0 != state_ptr->p_phase_0) && (phase_0 == QSIM_ONE)) {
                                /* do all the phase_0 stuff */
                                phase_0_statements();
                        }
                        else if ((phase_1 != state_ptr->p_phase_1) && (phase_1 == QSIM_ONE)){
                                /* do all the phase_1 stuff */
                                phase_1_statements();
                        }
                }
                else {
                        /* handles non-overlapping clocks */
                        if ((phase_0 != state_ptr->p_phase_0) && (phase_0 == QSIM_ONE)) {
                                /* do phase_0 stuff */
                                phase_0_statements();
                        }
                        else if ((phase_1 != state_ptr->p_phase_1) && (phase_1 == QSIM_ONE)) {
                                /* do phase_1 stuff */
                                phase_1_statements();
                        }
                }
        }
        state_ptr->p_phase_0 = phase_0;
        state_ptr->p_phase_1 = phase_1;
        return;
}
```

The functions which dictate the operations per clock cycle are shown below, followed by routines for input, calculation and output.

```
void phase_0_statements()
{
        /* check for a reset condition */
        test_reset_flag();
        /* check that a reset has completed */
        if (state_ptr->reset_counter == 0xFF) {
                switch (state_ptr->clock_counter) {
                        case 0: input_multiplicand();
                                break;
                        case 1: input_multiplier();
                                break;
                        case 2: multiply_calculate();
                                break;
                }
        }
        return;
}
```

```
void phase_1_statements()
{
        if (state_ptr->reset_counter == 0xFF) {
                switch (state_ptr->clock_counter) {
                        case 0: if (state_ptr->hold_flag && state_ptr->result_flag) {
                                        output_result();
                                        output_valid(QSIM_ZERO);
                                }
                                else {
                                        output_valid(QSIM-ONE);
                                }
                                break;
                        case 1: if (state_ptr->hold-flag && state_ptr->result_flag) {
                                        output_result();
                                        output_valid(QSIM_ZERO);
                                }
                                else {
                                        output_valid(QSIM_ONE);
                                } break;
                        case 2: if (state_ptr->hold_flag && state_ptr->result_flag) {
                                        output_result();
                                        /* reset valid_flag */
                                        state_ptr->valid_flag = FALSE;
                                        output_valid(QSIM_ONE);
                                }
                                break;
                        case 3: if (state_ptr->hotd_flag && state_ptr->result_flag) {
                                        output-result();
                                        output_valid(QSIM_ZERO);
                                }
                                break;
                        case 4: output_result();
                                output_valid(QSIM_ZERO);
                                break;
                }
                /* increment clock_counter */
                state_ptr->clock_counter++;
                if (state_ptr->clock-counter > 4) {
                        state-ptr->clock_counter = 0;
                }
        }
        return;
}


void input_multiplicand()
{
        short bit;
        unsigned long data_bit, multiplier_value;

        /* input data from multiplicand input */
        multiplier_value = 0;
        for (bit = 0; bit < 32; bit++) {
                data_bit = qsim_con_value[(*(qsim_instance_ptr
                                        ->multiplier_I_multiplicand))
                                        ->bits[bit]];
                multiplier_value += data_bit << bit;
        }
        /* save input internally */
        state_ptr->multiplicand = multiplier_value;
        /* set multiplicand_flag */
        state-ptr->multiplicand_flag = TRUE;

        return;
}
```

```
void input_multiplier()
{
        short bit;
        unsigned long data_bit, multiplier_value;

        /* input data from multiplier input */
        multiplier_value = 0;
        for (bit = 0; bit < 32; bit++) {
                data_bit = qsum_con_value[(*(qsim_instance_ptr
                                        ->multiplier_l_multiplier))
                                        ->bits[bit]];
                multiplier_value += data_bit << bit;
        }
        /* save input internally */
        state_ptr->multiplier = multiplier_value;
        /* set multiplier_flag */
        state_ptr->multiplier_flag = TRUE;

        return
}


void multiply_calculate()
{
        short shifted, m_s_bit_position, shift_pointer, no_bits;
        unsigned long multiplier_position, bit_position, bit_multiplier, carry_test;

        shifted = 0;
        shift_pointer = 1;
        multiplier_position = state_ptr->multiplier;

        /* find m_s_bit position to optimise multiply time */
        while ((!(multiplier_position & M_S_BIT_MASK)) && shifted < 32) {
                multiplier_position <<= 1;
                shifted++;
        }
        m_s_bit_position = 32-shifted;
        /* do (32x32=64)bit multiply */
        state_ptr->result.lsw = 0;
        state_ptr->result.msw = 0;
        for (no_bits = 0; no_bits < m_s_bit_position; no_bits++) {
                bit_position = shift_pointer << no_bits;
                bit_multiplier = state_ptr->multiplier & bit_position;
                if (bit_multiplier) {
                        /* do least significant long word */
                        carry_test = state_ptr->result.lsw+(state_ptr->multiplicand<<no_bits);
                        /* if carry_test < both operands then carry condition */
                        if ((carry_test < state_ptr->result.lsw) &&
                                (carry_test < state_ptr->multiplicand << no_bits)) {
                                state_ptr->result.msw++;
                        }
                        state_ptr->result.lsw = carry_test;
                        /* do most significant long word */
                        state_ptr->result.msw += state_ptr->multiplicand >> (32 - no_bits);
                }
        }
        /* set result_flag */
        state_ptr->result_flag = TRUE;
        state_ptr->multiplicand_flag = FALSE;
        state_ptr->multiplier_flag = FALSE;
        return;
}
```

```
void output_result()
{
        short bit;
        unsigned long bit_pointer, lsw_bit, msw_bit;
        qsim_bit_string_t output_vlw, output_lsw, output_msw;

        /* output result */
        bit_pointer = 1;
        for (bit = 0; bit < 32; bit++) {
                lsw_bit = ((state_ptr->result.lsw) & (bit_pointer << bit)) >> bit;
                msw_bit = ((state_ptr->result.msw) & (bit-pointer << bit )) >> bit;
                output_vlw[bit] = qsim_con_state[lsw_bit][QSIM_STRONG];
                output_vlw[bit + 32] = qsim_con_state[msw_bit][QSIM_STRONG];
        }
        qsim_drive_delay_output(&qsum_instance_ptr->multiplier_O_result, output_vlw);
        /* set valid_flag */
        state_ptr->valid_flag = TRUE;
        return;
}
```

4.4. Results

The waveforms for simulation of the multiplier example are given in Figures 5 to 8. These show inputs and outputs from the multiplier with an arbitrary time scale. The main objectives of this example are to show the simplicity with which a BLM can be written — it requires only basic knowledge of C, the flexibility available — simulation of a faster multiplier can be achieved by reordering the function calls with in the clock routines, and how timing of the program is independent of the simulation timing. Output can be scheduled for any time the user requires.

Many other options could have been included in the simulation, including extensive error checking with message reporting, using multiple instances of the multiplier on one sheet to demonstrate the independence of each block, or even creating further blocks within the multiplier block to perform the simulation at a lower level, without resorting to a gate-level model.

5. Conclusion

The work carried out with graphics VLSI design using the strategy presented and Behavioural Language Modelling at Sussex has been found to give us a consistent environment in which to work and a more complete method for partitioning, simulating and evaluating stages of a design. We have found this to be practical when a relatively straightforward design is to be evaluated or the level of internal detail is kept minimal. The results obtained can be used to collate statistical information about function usage, pin usage, etc., to detect, for example, bottlenecks in a design. For complex designs, the effort of software writing and debugging a BLM can be considerable, although if this route is chosen from the outset of a design, integration of stages can minimise this.

This work has been undertaken as part of a collaborative project with GEC Hirst Research Centre and Singer Link-Miles, funded by the Alvey Directorate. The contribution of all members of this consortium is acknowledged.

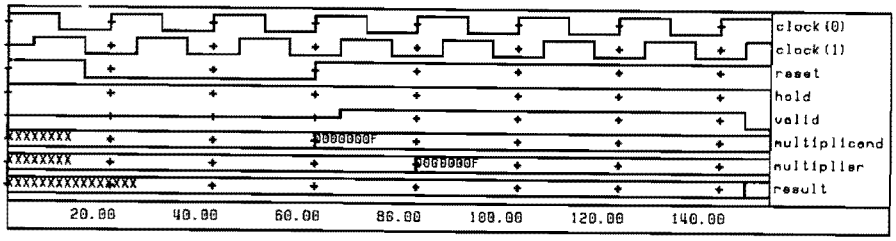

Figure 5: Simulation Trace 0 - 150.

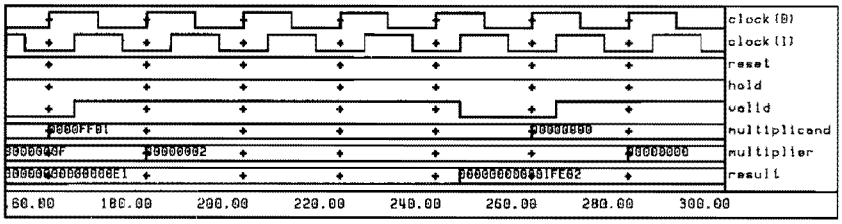

Figure 6: Simulation Trace 151 - 300.

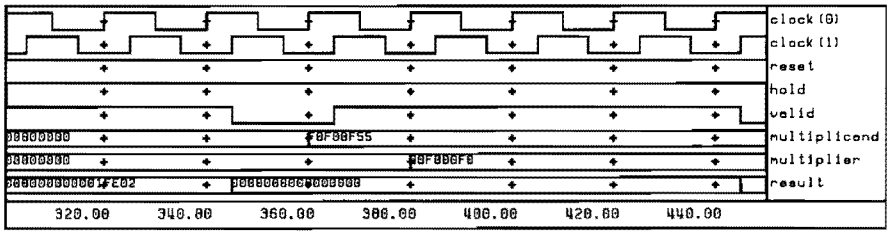

Figure 7: Simulation Trace 301 - 450.

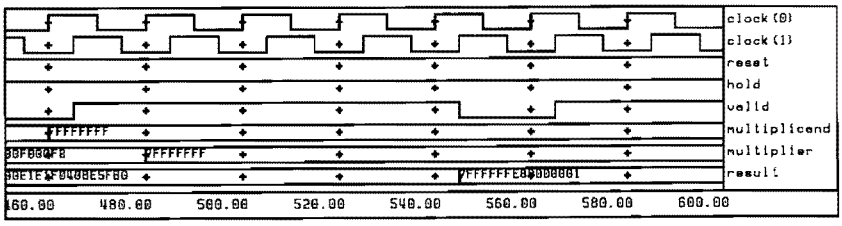

Figure 8: Simulation Trace 451 - 600.

6. References

1. Asal, M., Short, G., Preston, T., Simpson, R., Roskell, D., and Guttag, K., "The Texas Instruments 34010 Graphics System Processor", *IEEE CG&A*, October 1986.
2. Nickel, R., (Silicon Graphics, Inc.) "The IRIS Workstation", *IEEE CG&A*, August 1984.
3. Clark, J.H., "The Geometry Engine: A VLSI Geometry System for Graphics", *Computer Graphics*, July 1982.
4. Ikedo, T., (Seillac Co., Ltd) "High Speed techniques for a 3-D Color Graphics Terminal", *IEEE CG&A*, May 1984.
5. Smith, D., "The Integration of Graphics and Imaging Systems", *VLSI Systems Design*, February 1988.
6. Jamieson, L.H., "Characterising Parallel Algorithms", *The Characteristics of Parallel Algorithms*, The MIT Press, 1987.
7. Rogers, D.F., "Procedural Elements for Computer Graphics", McGraw-Hill, 1985.
8. Robison, A.D., "Illinois Functional Programming: A Tutorial", *BYTE*, February 1987.
9. Pountain, D., May, D., "A Tutorial Introduction to OCCAM Programming", BSP Professional Books, March 1988.
10. Organick, E.I., Carter, T.M., Maloney, M.P., Davis, A., Hayes, A.B., Klass, D., Lindstrom, G., Nelson, B.E., and Smith, K.F., "Transforming an Ada Program Unit to Silicon and Verifying Its Behavior in an Ada Environment: A FIRST EXPERIMENT", *IEEE Software*, January 1984.
11. David, N., (Xerox Corporation) "Using Silicon Compilation in a Commercial Product Development Project", *COMPCON 86 IEEE Computer Society* (Order Number 692).
12. Lloyd, L., "Systems Design using BLM's", British Mentor Graphics User Group, 12-13 July 1988.